Kapitel 2
Container und Algorithmen

2.1 Einleitung

Als Programmierer wird man nur in wenigen Fällen alles neu von Grund auf entwickeln und selbst schreiben. Meistens greift man auf fertigen Code zurück, der Basisdienste zur Lösung der eigenen Aufgaben bereitstellt und oft nur durch eigenen Code ergänzt, seltener auch für die eigenen Bedürfnisse modifiziert wird. Fertigen Code in Form vorübersetzter Module oder als echten Quellkode erhält man in mehreren Formen:

- Als Basisbibliotheken, die den Entwicklungssystemen immer beigefügt sind und auch fast immer verwendet werden.
- Als Spezialbibliotheken, die nur bei recht speziellen Problemen zum Einsatz kommen und meist separat von der Entwicklungsumgebung organisiert werden müssen.
- Als fertige Anwendungen, aus denen man Teile für den eigenen Bedarf ausgliedert.

C++ Entwicklungssystemen sind standardmäßig die C-Bibliotheken und ihre Pendants in C++ beigefügt. Eine weitere mächtige Universalbibliothek ist die boost++ Bibliothek, eine spezialisiertere beispielsweise die blitz++ Bibliothek für bestimmte nummerische Anwendungen. Wir werden uns in diesem Kapitel vorzugsweise mit der C++ Standardbibliothek, genauer mit der Standard Template Library STL, beschäftigen.

> **Aufgabe.** Suchen Sie ein wenig im Internet, welche Bibliotheken dort frei verfügbar sind, und laden Sie sie probeweise herunter. Vermutlich werden Sie zunächst relativ wenig damit anfangen können, aber bei der weiteren Arbeit kann ein Seitenblick, ob ähnliche Funktionen auch in den Bibliotheken angeboten werden und wie die Probleme dort gelöst sind, manchmal ganz hilfreich sein.

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_2, © Springer-Verlag Berlin Heidelberg 2010

In vielen Anwendungen hat man mehrere Objekte einer Sorte oder einer Klassenfamilie zu verwalten. Der Oberbegriff für derartige Verwaltungsaufgaben ist der des Containers, wobei die einfachste Form des Containers ein Feld ist:[1]

```
MyType* array;
array = new MyType[field_len];
...
delete[] array;
```

Nun existieren verschiedene Containerformen, wie wir sehen werden, und es ist nicht egal, welcher Container zum Einsatz kommt. Um den optimalen Container für eine Anwendung zu finden, müssen viele Randbedingungen betrachtet werden, zum Beispiel:

- Wird auf die Elemente nur nacheinander (seriell) oder auch zufällig (indiziert) zugegriffen?
- Werden neue Elemente hinten, vorne oder auch in der Mitte eingefügt (oder entfernt)?
- Werden Suchvorgänge nach bestimmten Elementen auf dem Container ausgeführt?
- Ist der Container sortiert, teilsortiert oder unsortiert?

Je nach Eigenschaften eines Containers und Art der Aufgabe wird man bestimmte Algorithmen ausführen müssen. Eine Klassifizierung der Qualität eines Containers zusammen mit einem passenden Algorithmus hinsichtlich einer bestimmten Anforderung wird mittels der Komplexitätsordnung durchgeführt. Hierunter versteht man den funktionellen Zusammenhang zwischen Laufzeit und Elementanzahl im Container, reduziert auf das führende Glied. Liegt beispielsweise folgender Algorithmus vor

```
for(i=0;i<n;i++){
   for(j=i+1;j<n;j++){....}}
for(i=0;i<n;i++){...}
```

so wäre der funktionelle Zusammenhang durch

$$t(n) = \frac{k_1}{2} n^2 + k_2 * n$$

gegeben und die Laufzeitordnung, da nur das führende Glied gezählt und Konstanten fortgelassen werden, durch $O(n^2)$.[2]

---

[1] Beachten Sie die etwas abgewandelte Syntax beim Erzeugen und Vernichten von Feldern gegenüber Einzelvariablen!

[2] Die Anzahl der Schachtelungen der for-Schleifen gibt die Potenz der Laufzeitordnung an, wobei es unerheblich ist, ob eine Schleife immer vollständig durchlaufen wird. Dies spielt erst dann eine

Wir werden in diesem Kapitel eine Reihe von Containertypen und auf ihnen ablaufende Algorithmen nach diesen Gesichtspunkten untersuchen (*es handelt sich vorzugsweise um diejenigen, die Gegenstand von Vorlesungen über Algorithmen und Datenstrukturen sind, d.h. Sie können dieses Buch auch für diese Studieneinheit benutzen*), wobei wir glücklicherweise auf komplette Implementationen verzichten können, denn Container und Algorithmen sind in der C++ Standard Template Library (STL) vorhanden. Wie der Name schon sagt, basiert die Bibliothek auf der intensiven Verwendung von Templates, und diese wurden noch nicht so weit vertieft, wie das wünschenswert wäre. Das holen wir jetzt nach, allerdings auch noch nicht vollständig, sondern nur so weit, dass wir die Techniken dieses Kapitels erklären können. Die anderen Kapitel des Buches halten weitere anwendungsbezogene Programmierdetails der Templatetechniken bereit.

2.2 Template-Klassen, inline-Funktionen

2.2.1 Template-Klassen und Template-Funktionen

Templates können sich auf Klassen oder Funktionen beziehen. Die Klassen der STL sind durchweg als Vorlagenklassen (*templates*) implementiert, das heißt die Datentypen bestimmter Attribute sind nicht festgelegt, sondern durch Platzhalter vertreten.

```
// Implementierung:
// ================
template <class T> class A {
    ...
    T t;
    ...
};//end class
```

In Templatefunktionen betrifft dies auf die Übergabe- und Rückgabeparameter sowie in der Implementation verwendete temporäre Variable.

```
template <class T> T square(const T& t){
     return t*t;
}//end function
```

Bei der Implementation der Anwendung werden die Template-Klassen mit den erforderlichen speziellen Datentypen bei der Deklaration der Arbeitsvariablen verwendet. Beim Einsatz der Template-Funktionen ist meist klar, was zu verwenden ist, jedoch kann hier in zweifelhaften Fällen auch eine direkte Spezifikation angegeben werden.

Rolle, wenn ordnungsmäßig gleichwertige Algorithmen verglichen werden, aber da misst man in der Regel das Laufzeitverhalten besser praktisch, als sich allzu sehr auf die Theorie zu verlassen.

```
// Aufrufe in der Anwendung
// ========================
A<double> a; A<long> b;
double r,s, int i,j;
r=square(r);
i=square(i);
s=square<double>(i);
```

Der Übersetzer erkennt bei der Deklaration der Variablen `a` und `b`, für welche Spezialtypen eine Implementierung zu erstellen ist und generiert zwei Versionen der Klasse `A` sowie zwei Versionen der Funktion `square` für die Bedienung der unterschiedlichen Parametertypen, in diesem Fall also eine Version mit einer ganzzahligen Multiplikation und eine zweite mit Fließkommamultiplikation.

Wenn bei Funktionen klar ist, für welche Datentypen die Compilerarbeit durchzuführen ist, müssen diese im Funktionsaufruf nicht genannt werden. Der dritte Funktionsaufruf zeigt die Methodik, wenn der Programmierer die Compilerautomatik überschreiben will. Trotz des int-Parameters wird eine Methode für double-Typen vom Compiler implementiert (und der int-Typ automatisch beim Aufruf nach double konvertiert). Diese direkte Spezifikation ermöglicht auch Funktionsversionen, bei denen der Compiler aus dem Aufruf nicht alle notwendigen Details entnahmen kann:

```
template <class T, class U> void f(T& t){
   U u; ...

int k;
f<int,double>(k);
...
```

Konstruktionen dieser Art sollte man allerdings in der Regel meiden, da die Typautomatik von C++ hierdurch teilweise ausgeschaltet und die Verantwortung dem Programmierer übertragen wird, die Fehlermöglichkeit also steigt.

Templates ähneln somit in gewisser Weise den bekannten Funktionsdefinitionen, wobei die Templateparameter die Funktion der Übergabenvariablen einer Funktion übernehmen. Templates werden zur Compilezeit ausgewertet, und Datentypen übernehmen die Rolle der Werte bei der Laufzeitauswertung von Funktionen. Wie Funktionsrückgabewerte über Templates realisiert werden (*und dass Templates im Gegensatz zu Funktionen sogar beliebig viele Rückgabewerte erlauben*) und was man damit anfangen kann, werden wir in den verschiedenen Kapiteln dieses Buches noch sehen.

Als Templateparameter können alle Datentypen/Klassen eingesetzt werden, für die die in Funktionen und Methoden verwendeten Operatoren oder Methodenaufrufe implementiert sind. Dabei leistet der Compiler eine recht beeindruckende Arbeit, denn die Parameter können ihrerseits wieder Templatetypen sein, die rekursiv über mehrere Stufen aufzulösen sind. Bei geschachtelten Templateparametern ist auf eine

Besonderheit beim editieren zu achten. Um Verwechslungsprobleme mit dem Operator >> beim Parsen zu vermeiden, sind in Schachtelungen die „>“ - Klammern durch ein Leerzeichen zu trennen:

```
A<B<int> > a
```

Der Vorteil dieser Programmierart ist, dass einmal erstellter und geprüfter Code ohne weiteren Programmieraufwand auch mit Datentypen einsetzbar ist, die die gleichen Eigenschaften aufweisen. Die Abstraktionsweise der Mathematik wird so auf elegante Weise auf die Programmiertechnik übertragen.

Beispiel. In allen euklidischen Ringen, das heißt algebraischen Mengen, in denen die Division mit Rest definiert ist, besteht die Möglichkeit, einen „größter gemeinsamen Teiler ggT“ zweier Elemente festzustellen. Solche Mengen sind unter anderem die Menge der ganzen Zahlen oder die Menge der Polynome über den reellen Zahlen bzw. beliebigen anderen Körpern wie komplexen Zahlen oder Modulkörpern. Der `ggT(a,b)` ist durch den Euklidischen Algorithmus berechenbar:

$$\begin{aligned} a &= q_1 * b + r_1 \\ b &= q_2 * r_1 + r_2 \\ &\dots \\ r_{n-1} &= q_{n+1} * r_n + r_{n+1} \end{aligned}$$

Nach den Regeln der Schulmathematik muss ein gemeinsamer Teiler von a und b auch den Rest teilen, d.h. wir können an Stelle von a und b auch nach dem gemeinsamen Teiler von b und r fragen. Da r kleiner ist als a, kann diese Rekursion nicht endlos fortgesetzt werden. Der Algorithmus bricht ab, wenn der Rest Null wird, und der `ggT` ist das letzte $r \neq 0$.

Eine Implementation des Algorithmus für ganze Zahlen kann nun sehr einfach durch einen Rechnung auf dem Papier auf Korrektheit kontrolliert werden. Anstatt in der Implementation für ganze Zahlen mit dem Datentyp `int` zu arbeiten und den Algorithmus dann auch nur für ganze Zahlen einsetzen zu können, wird der Algorithmus typunabhängig als `template`–Funktion implementiert

```
template <class T> T ggT(const T& t1, const T& t2){
    T a=Abs(t1);
    T b=Abs(t2);
    while(true){
        a%=b;
        if(a==0) return b;
        b%=a;
        if(b==0) return a;
    }//endwhile
}//end function
```

Nachdem festgestellt ist, dass der Algorithmus mit ganzen Zahlen fehlerfrei funktioniert, lässt er sich unmittelbar auch mit Polynomen als Vorlagenparameter einsetzen, da Polynome ähnliche Eigenschaften wie die ganzen Zahl aufweisen. Oder um es

etwas überspitzt zu formulieren: wenn wir über ähnlich einfach zu überprüfende Template-Klassen für Polynome, rationale Zahlen und komplexe Zahlen verfügen, erzeugt der Compiler auch für den Datentyp

```
Polynom<Rational<Complex<double>>>
```

ein mit dem `ggT`-Algorithmus korrekte Ergebnisse produzierendes Programm, auch wenn man das auf dem Papier nur noch mit einem ausgeprägten Hang zum Masochismus überprüfen würde.

Aufgabe. Polynome werden wir erst später implementieren, weshalb Sie dazu noch keine Aufgabe bekommen. Implementieren Sie zunächst den Euklidischen Algorithmus, wobei Sie die Absolutfunktion ebenfalls als `template` implementieren, das im einfachsten Fall nichts macht.

Anschließend versuchen Sie den Algorithmus folgendermaßen zu erweitern: Wenn man die Entwicklungsformel des Euklidischen Algorithmus nach r_{n+1} auflöst und rückwärts wieder einsetzt, kann man nachweisen, dass zwischen teilerfremden (a, b) die Beziehung

$$1 = u * a + v * b$$

mit ganzen Zahlen (u, v) gilt. Der erweiterte Algorithmus soll neben dem `ggT` auch die Zahlen (u, v) bestimmen. Versuchen Sie, das Rekursionsschema für (u, v) in jedem Schritt des Algorithmus zu ermitteln und zu implementieren.

2.2.2 Spezialisierungen

Sind für einen bestimmten Datentyp effizientere Algorithmen bekannt, so lässt sich eine Methode oder Klasse auch spezialisieren (*damit steigt natürlich wieder die Anzahl der Implementierungen an*). Ist beispielsweise für einen Datentyp `B` eine effizientere Art der Quadratbildung bekannt (*wir werden solche Fälle noch behandeln*) oder ist für einen speziellen Datentyp eine effizientere Implementierung einer Vorlagenklasse möglich, so kann dies folgendermaßen implementiert werden:

```
template <> class A<B> {... };
template <> B square(const B& t) { ... };
```

Um die für einen spezielle Anwendung beste Implementierung zu finden, durchsucht der Compiler bei Auftreten einer `square`–Anweisung seine Methodenliste, wobei er beim letzten Eintrag beginnt, und verwendet den ersten passenden Eintrag. Es ist deshalb wichtig, mehrstufige Spezialisierungen in der richtigen Reihenfolge zu definieren. Mit den Bezeichnungen von oben beispielsweise:

```
template <class T> T square(const T& t) { ... };
template <class T> A<T> square(const A<T>& t){...}
template <> double square(const double& t) { ... };
```

Bei dieser Vorgehensweise ermittelt der Übersetzer bei den Anweisungen

```
A<double> a;
double r;
int i;
...
a = square(a);
r = square(r);
i = square(i);
```

für die Variable `r` die untere Methode, für `a` die mittlere, da es sich bei `a` um eine Variable einer Templateklasse handelt, und für `i` die obere Methode, da die beiden anderen nicht passen.

Das Konzept kann nicht nur für spezielle Implementationen eingesetzt werden, sondern auch zur Verhinderung der Nutzung von Methoden durch bestimmte Datentypen. Bei der Spezialisierung

```
template <class T> void f(const T& t){...};
template <class T> void f(const complex<T>& t);
```

wird der Compiler bei der Nutzung von `f` durch eine Variable des Typs `complex<..>` mit beliebigem Typparameter auf die zweite Templatedefinition stoßen. Ist nun wie hier zu diesem Typ keine Implementation angegeben, endet die ganze Angelegenheit mit einem Compilerfehler, während mit allen anderen Datentypen, die nicht mit `complex<..>` zu tun haben, die Übersetzung problemlos erfolgt.

Das Spezialisierungskonzept gilt nicht nur für Funktionen, sondern auch Klassen können in dieser Weise spezialisiert werden. Die zu diskutierenden Anwendungen enthalten hinreichend Beispiele dieser Art, so dass hier diese Bemerkung genügt.

Aufgabe. Schreiben Sie einige komplexere Spezialisierungen von Klassen und Funktionen und prüfen Sie durch Bildschirmausgaben das Compilerverhalten.

2.2.3 Offener Code

Ein Nachteil der Verwendung von `templates` kann sein, dass der Code immer als Quelle ausgeliefert werden muss, wenn es sich um Entwicklungsumgebungen und nicht um fertige Anwendungen handelt, da der Übersetzer ja die passenden Typen zur Übersetzungszeit einsetzen muss und mit vorübersetzten Modulen nicht viel anfangen kann. Die `template` -Implementationen bestehen ausschließlich aus

Headerdateien, die Definition und Implementation enthalten.[3] Geheimhaltung von Algorithmen ist dann nicht möglich. Allerdings trifft das Problem vermutlich nur auf wenige Fälle zu, und die meisten Softwareunternehmen haben inzwischen den Standpunkt eingenommen, dass es ohnehin besser ist, dem Kunden gegenüber mit offenen Karten zu spielen.

2.2.4 Partielle Übersetzung

Für den Fall, dass man selbst `template` –Code entwickelt (*und im weiteren Verlauf dieses Buches werden eine Reihe von Beispielen auftauchen*), muss man allerdings einen entscheidenden Unterschied im Umgang des Compilers mit `template` –Code gegenüber Standardkode berücksichtigen. Wie bereits oben bei der Begründung der Vorteile, die man sich durch die Nutzung von `templates` verschafft, ausgeführt, sind nicht alle formal zur Verfügung gestellten Operationen sinnvoll oder technisch möglich. `operator%(..)` macht beispielsweise keinen Sinn bei Fließkommazahlen und ist auch nicht implementiert. Trotzdem kann er in einer Klasse, die sowohl mit ganzen Zahlen als auch mit Fließkommazahlen instanziiert wird, in einigen Methoden, die nur ganze Zahlen betreffen, auftreten. Der Aufruf einer solchen Methode für eine Fließkommazahl führt dann zu einem Compilerfehler

```
template <class T> struct A {
    T a,b;
...
    void f() { ... a%b; ... }
};//end struct

A<int> a; A<double> x;
...
a.f();        // ok
x.f();        // Fehler !
```

Um dem Entwickler trotzdem das Arbeiten mit beiden Datentypen als Template-Parameter zu ermöglichen, setzt der Compiler nicht nur die Datentypen in die Vorlagen ein, sondern analysiert auch, welche Methoden mit einem bestimmten Typ in der Anwendung aufgerufen werden, und implementiert auch nur diese Methoden. Wird also `f()` mit der Variablen `x` nicht verwendet, versucht er auch gar nicht erst, `f()` für `x` zu konstruieren und übersetzt das Programm fehlerfrei. Allerdings hat das auch einen Pferdefuß: Syntaxfehler fallen erst dann auf, wenn die Methoden in einer Anwendung auch eingesetzt werden. Dies ist beim Testen von Template-Code zu berücksichtigen. Speziell ist nach Ergänzungen/Erweiterungen der Code auch mit Datentypen erneut zu testen, die für die bearbeitete Anwendung gar nicht

[3] Bei den wenigen Ausnahmen, bei denen neben der .h-Datei auch eine .cpp-Datei vorhanden ist, enthält letztere keinen Vorlagenkode.

zur Disposition stehen, aber für die Prüfung bestimmter Methoden benötigt werden (*vergleiche die Bemerkungen zu Testumgebungen*).

2.2.5 Default–Parameter und template-template-Parameter

In vielen Anwendungen wird man universelle Strategien für bestimmte Probleme einsetzen, möchte aber vielleicht dem Anwendungsprogrammierer den Weg offen halten, auch eigene Strategien einzusetzen. Ein Beispiel dafür ist die bereits an anderer Stelle schon erwähnte Freispeicherverwaltung, deren generellen Strategien ja nicht unbedingt besonders effizient sind. Um nun nicht bei jeder Variablendeklaration alle Strategien im Parametersatz angeben zu müssen, können bei der Template-Vereinbarung Standardwerte angegeben werden:

```
template <class T, class A = MyStandard>
struct Strategies { ... };
...
Strategies<double> myStrat;
```

Standardparameter sind allerdings nur in Template-Klassen zulässig, nicht aber in Template-Funktionen. Bei der Implementation müssen Klassen für Parameter mit Standardtyp nur dann angegeben werden, wenn tatsächlich ein anderer gewollt ist:

```
Strategies<double,MyOwnStandard> myStrat2;
```

Sind in einer Template-Vereinbarung mehrere Parameter angegeben, so kann der Fall eintreten, dass die hinteren Parameter selbst wieder Templates sind, die von einem der vorderen Parameter abhängen und `template-template-`Parameter genannt werden. Diese Eigenschaft ist in der Parameterdeklaration zu hinterlegen:

```
template<class T> class myarray { /* ... */ };

template<class K, class V,
        template<class> class C = myarray>
class Map {
    C<K>    key;
    C<V>    value;
    C<int>  counter;
 ...
};
```

Der Parameter C kann innerhalb der `Map`-Klasse nach Bedarf mit beliebigen Parametern deklariert werden.

2.2.6 Rückgabe von Typen

Ein damit zusammenhängendes Problem ist, wie man von Parametern, die selbst Templates sind, deren Parametertyp ermittelt. Beispielsweise könnte man eine Sortierfunktion konstruieren, die neben anderen Typen auch mit dem Typ `myarray` arbeiten können soll:

```
template <typename T> void sort(T& cont){...}
...
myarray<double>ma;
sort(ma);
```

Für den Sortiervorgang muss sie auf die Elemente im Feld zurückgreifen und Größenvergleiche anstellen. Um dieses Problem zu lösen, definiert man in der Feldklasse einen Datentyp, der den Template-Parametertyp widerspiegelt:

```
template <typename T> class myarray {
public:
   typedef T value_type;
...

};//end class

template <typename T> void sort(T& cont){
   typename T::value_type obj;
...
}//end function
```

Die Variable `obj` besitzt nun den Datentyp, mit dem die im Aufruf verwendete `myarray`-Variable deklariert wurde, also hier der Typ `double`.

Wir haben hier eine der Möglichkeiten von Templates vor uns, Rückgabewerte zu erzeugen, wobei die Rückgabewerte Datentypen sind, die dem Compiler für seine weitere Arbeit zur Compilezeit zur Verfügung stehen. Eine Templateklasse darf beliebig viele Ausgabetypen deklarieren, deren Ermittlung sehr komplex sein kann, wie spätere Kapitel zeigen.

Zu beachten ist dabei das Syntaxwort `typename` in der Variablendeklaration. Der Compiler prüft den Code nämlich in zwei Durchläufen. Zunächst prüft er nur die C-Syntax, konstruiert aber die Template-Klassen und -Funktionen noch nicht, anschließend führt er die Konstruktionen durch und prüft nun, ob auch alle Methoden und Attribute definiert sind (*und im abschließenden dritten Linkerdurchlauf wird nun auch geprüft, ob alle Implementationen vorhanden sind*).

Im ersten Durchlauf stellt sich bei `T::value _type` in der Funktion nun das Problem, ob es sich hierbei um einen Attributaufruf oder eine Typdefinition handelt. Da die Template-Typen noch nicht konstruiert werden, kann die Frage hier nicht geklärt werden, so dass der Compiler gemäß C++-Standard von der Attribut-Vermutung ausgeht. `T::value _type obj` macht aber syntaxmäßig keinen Sinn, so dass ein Compilerfehler die Folge wäre. Das vorangestellte `typename` sorgt nun dafür, dass der Compiler die Anweisung als Typvorgabe interpretiert

und die Syntax akzeptiert (*wenn das falsch war, meckert er natürlich im nächsten Durchlauf*).

2.2.7 Zahlen als Templateparameter

Außer Klassen können in Template-Parameterlisten auch Werte auftreten, die zur Kennzeichnung von Klassen zur Laufzeit durch Zahlen dienen:

```
template <int i> struct WertParameter {
   enum { value = i }; };
   int a[i];
   ...
```

Der Wertparameter wird hier in Form eines `enum`–Wertes hinterlegt und zur Festlegung einer Feldgröße eingesetzt. Wertparameter können aber auch zur Implementation von Compileralgorithmen einsetzt werden:

```
template <int i> struct Sum {
   enum { value = i + Sum<i-1>::value };
};//end struct

template <int> struct Sum<1> {
   enum { value = 1 };
};//end struct
```

Die Befehlszeile

```
cout << Sum<50>::value << endl;
```

gibt dann die Summe aller ganzen Zahlen zwischen 1 und 50 aus. Die Berechnung wird vom Compiler und nicht etwa zur Laufzeit durchgeführt. Für die Konstruktion von `Sum<50>` benötigt dieser nämlich `Sum<49>`, was er sofort zu konstruieren versucht, und so fort, bis er bei `Sum<1>` auf eine Spezialisierung stößt, die keine Fortsetzung erfordert, so dass die Rekursion abbricht. Wir werden später noch spezielle Beispiele solcher Compileralgorithmen kennenlernen.

Zahlen als Templateparameter stellen somit einerseits eine weitere Möglichkeit zur Rückgabe „berechneter“ Werte dar. Daneben können sie eingesetzt werden, um mehrere verschiedene Datentypen mit der gleichen Wertdarstellung zu erzeugen oder Compileralgorithmen zu kontrollieren.

In einigen rekursiven Algorithmen kann es notwendig sein, die rekursiven Schritte durch eigene Datentypen voneinander zu trennen. Der Datentyp

```
template <int i> struct int2type {
    enum {value = i};
};
```

ermöglicht dies, da z.B. `int2type<1>` und `int2type<2>` für den Compiler unterschiedliche Typen darstellen und er unterschiedliche Implementationen erzeugt,

obwohl die Typen selbst leer sind, also nichts Wesentliches zur Implementation beitragen. Wie dies genutzt werden kann, um in Compilerrekursionen auf bestimmte Objekte zugreifen zu können, werden wir noch sehen.
Ähnlich erlaubt

```
template <class T> struct type2type {
    typename T original_type;
};
```

eine leere Umtypisierung des Typs `T`, wenn dieser nicht direkt verwendet werden soll. Beispielsweise würde ohne ein solches Hilfskonstrukt die Funktion

```
template <class T,class U>
T* make(U const& u, type2type<T>) { ...
```

in der Form

```
template <class T,class U>
T* make(U const& u, T) { ...
```

definiert werden müssen, was jeweils die (möglicherweise gar nicht erlaubte) Konstruktion eines temporären Objektes des Typs `T` erfordert.

2.2.8 Effizienz und inline -Code

Nun kann (*und sollte*) man sich an dieser Stelle fragen, ob bei der ganzen objektorientierten Vorgehensweise überhaupt noch effizienter Code zustande kommt und der Entwickler die Übersicht über das System behalten kann. Immerhin ist die Kapselung, von der wir hier noch gar nicht gesprochen haben, immer mit Methodenaufrufen verbunden, und ein Methodenaufruf von `operator()(int zeile,int spalte)` zum Zugriff auf ein Matrixelement bedeutet ja einigen Verwaltungsaufwand. Die Antwort ist in beiden Fällen JA! Bei Einhaltung der Regeln für die Implementation von Spezialisierungen kann die Auflösung bei einem Aufruf dem Compiler überlassen werden. Der Anwendungsprogrammierer ruft nur einen Funktionsnamen mit einem für ihn eindeutigen Arbeitsergebnis auf und kann sicher sein, dass die optimale Version genutzt wird – womit zunächst die Frage nach der Übersichtlichkeit beantwortet ist.[4]

[4] Das bedeutet allerdings nicht, dass man sich nun leicht eine Übersicht über das verschaffen könnte, was wirklich in der Bibliothek passiert. Wie wir noch sehen werden, sind die Containerklassen aus gutem Grund komplexer gestrickt, und entsprechend komplex fallen die Spezialisierungen aus. Zusammen mit dem zweiten Aspekt „Geschwindigkeit“ führt das dazu, dass man sowohl bei der Untersuchung des Quellkodes als auch beim Debuggen einer Anwendung sehr viel Zeit und Aufwand investieren muss, um nicht die Übersicht zu verlieren. Da das meist nur auf intellektuelle Erkenntnis hinausläuft, die man sonst nur schlecht verwerten kann, kann man auf diesen Aufwand verzichten.

Die Antwort auf die Frage nach der Effizienz heißt inline -Funktion. Den Elementzugriff auf Matrixelemente wird man folgendermaßen in der Header-Datei implementieren:

```
inline double& operator()(int zeile,int spalte){
    return a[zeile*dim+spalte];
};//end function
```

Im Debugmodus kann man zunächst keinen Unterschied feststellen, wenn die `inline`–Deklaration fehlt. In die Operatormethode wird wie zuvor hineingesprungen, die Rückgabewerte auf dem Stack abgelegt, und das Programm ist nicht übertrieben schnell. Das ändert sich schlagartig, wenn der Optimierer aktiviert wird. Sofern dieser richtig funktioniert, wird nun anstelle eines Funktionsaufrufes der Code direkt eingebunden und im besten Fall sogar die Indexarithmetik als Adressrechnung erkannt und registeroptimiert ausgeführt. Das Ergebnis ist Code, der hinsichtlich der Ausführungsgeschwindigkeit nur unwesentlich hinter Code der Programmiersprache FORTRAN zurücksteht, in der Indexzugriffe der Form `a[i,j]` direkt im Sprachstandard definiert sind. Damit der Optimierer so arbeiten kann, sind natürlich alle `inline`-Funktionen in den Headerdateien zu implementieren. Da kein Eintrag in eine Funktionstabelle erfolgt, treten auch keine Namenskonflikte auf, die entstehen, wenn das Wort `inline` fortgelassen und die Header-Datei in mehrere Quellen eingebunden wird. In diesem Fall findet der Linker mehrere Funktionen des gleichen Namens in den Objektmodulen und weigert sich, hier eine Entscheidung zu treffen. Das `inline`–Konzept führt natürlich zu größerem ausführbaren Code, da die gleiche Methode mehrfach vorhanden ist.

> **Aufgabe.** Schreiben Sie ein einfaches Programm mit einer `inline`–Funktion, beispielsweise den oben beschriebenen Zugriff auf ein Matrixelement. Rufen Sie diese Funktion in einer Schleife so häufig auf auf, dass die Laufzeit messbar wird. Compilieren Sie anschließend mit Optimierung und messen Sie erneut.
>
> Die gleiche Messung können Sie mit einer Klasse durchführen, in der eine Methode wahlweise normal oder virtuell definiert wird. Sie erhalten dann eine Abschätzung des Aufwands der Verwaltung von virtuellen Methoden.

2.3 Zugriffe auf Daten: Verallgemeinerte Zeiger

2.3.1 Iteratoren

Für die Verwaltung größerer Objektmengen existieren eine ganze Reihe verschiedener Strategien, denen wir uns in den folgenden Teilkapiteln zuwenden. Unabhängig von der Speicherstrategie müssen alle Container die Möglichkeit bieten, auf alle gespeicherten Objekte nacheinander zugreifen zu können. Denken wir an Datenzugriffe in C, dann ist die universellste Möglichkeit wohl die Verwendung von Zeigern. Die folgenden beiden Programmkodes erledigen die gleiche Aufgabe:

```
int a[size]; int i;
for(i=0;i<size;i++) a[i]=i;

// alternative Implementation mit Zeigern;

int* i;
for(i=a;i!=a+size;i++) *i=(i-a);
```

Nun kann man Zeiger immer verwenden, um sich in einer Liste von Objekten zu bewegen, Indizes aber nicht unbedingt. Beispielsweise könnten die Elemente unterschiedliche Größen besitzen, was es immer noch erlauben würde, einen Zeigerwert von einem Objekt auf das nächste vorzurücken (wenn auch nicht durch eine einfache `for`-Schleife wie in dem Beispiel), während der Index sich nur auf das erste Element bezieht und in solchen Fällen zu anderen Positionen gelangt. Es liegt daher nahe, den Zeigerbegriff zu verallgemeinern und auf einem Container

```
template <class T> class Container { ... }
```
[5]

ein generalisiertes Zeigerobjekt zu definieren, das in einem bestimmten Umfang wie ein normaler C-Zeiger auf einem Feld zu bedienen ist. Ein generalisiertes Zeigerobjekt ist eine Template-Klasse, deren Operatoren so überschrieben sind, das ein Anwendungsentwickler Objekte der Klasse beim Einsatz nicht von einfachen Zeigern, beispielsweise vom Typ `char*` auf einem String oder einem Zeiger auf eine einfache Struktur, unterscheiden kann (*die Schnittstelle ist „transparent“*) und intern den einfachen Vorgängen äquivalente Abläufe auf beliebig komplexen Datenstrukturen abgewickelt werden.

Eine solche Klasse erhält den Namen (*oder Namensbestandteil*) `iterator`. Wenn wir uns vor Augen halten, welche Operationen mit oder auf C–Zeigern möglich sind, erhalten wir folgende Methodenliste (*vergewissern Sie sich von der Korrektheit und Vollständigkeit, in dem Sie* `int*pi` *als Iterator auf* `int ic[100]` *interpretieren und die Möglichkeiten der Verwendung von* `pi` *einmal durchspielen*):

```
template <class T> class Iterator{
public:
// Konstruktoren und Destruktoren
    Iterator();
    Iterator (const Iterator & it)
    ~Iterator();

// Elementzugriffe
    // Zugriff auf den Wert
    T& operator * ();
    // Zugriff auf Strukturen
    T* operator-> ();
```

[5] „Container” ist ein Synonym für eine beliebige Containerklasse. Eine Klasse „Container” existiert in der STL nicht.

```
    // indizierter Zugriff auf Feldwerte
    T& operator[] (int n);
    T const& operator * () const ;
    T const* operator-> () const ;
    T const& operator[] (int n)const ;
// Zeigerarithmetik
    // Post- und Präfixinkrementoren
    Iterator&  operator++ ();
    Iterator   operator++ (int) ;
    Iterator&  operator-- () ;
    Iterator   operator-- (int) ;
    // beliebige Inkrementierung
    Iterator&  operator+= (int n);
    Iterator&  operator-= (int n);
// Vergleiche von Iteratoren
   bool  operator==(const Iterator& it) const
};//end class
```

Es sei jedoch sofort angemerkt, dass dieser Ansatz nicht in allen STL-Klassen zur Anwendung kommt. Je nach Datenstruktur, die sich hinter dem Iterator verbirgt, sind nicht alle Operatoren (*sinnvoll*) implementierbar und treten dann in den entsprechenden Container-Klassen auch nicht in Erscheinung (*siehe Kapitel Iteratorkategorien*).

Aufgabe. Angenommen, `T& operator*() const` kann für einen Iterator nicht sinnvoll eingesetzt werden. Wie können Sie eine Verwendung im Code ausschließen?

2.3.2 Einsatz von Iteratoren

Eine normale Schleife mit Indexzugriff in einem Feld besitzt das Aussehen

```
for(i=0;i<n;i++)
   a[i]= ... ;
```

Den Einsatz von Iteratoren organisieren wir in der gleichen Art (s.o.):

```
for(it=a,it!=&a[n];it++)
   *it= ... ;
```

Der Durchlauf eines Iterators durch einen Container beginnt beim ersten Eintrag und wird in jedem Schleifendurchlauf inkrementiert. Der Inkrementoperator muss daher von allen Iteratoren unterstützt werden. Abgebrochen wird die Schleife hinter dem letzten Eintrag, d.h. der Enditerator zeigt auf eine Speicherstelle, die im Container nicht existiert. Ein Container ist also genau dann leer, wenn der Startiterator gleich dem Enditerator ist

Wichtig ist, dass der Abbruchvergleich nicht mit „<“ wie in einer Schleife mit Zahlen, sondern mit „!=“ durchgeführt wird. Bereits bei normalen Arrays besteht nämlich das Problem, dass nicht festgelegt ist, wie die Adressen, auf die eine Zeigervariable verweist, ausgewertet werden. Ist der auswertungstyp beispielsweise 32-Bit-Integer auf einer 32-Bit-Maschine, dann kehrt sich formal die Reihenfolge der Zeiger um, sobald sie hoch genug im Speicher angelegt werden. Wenn man das mit einem Cast auf einen geeigneten Zahlentyp möglicherweise noch fixen könnte, verliert der „<“-Operator endgültig seinen Sinn, wenn Iteratorklassen zum Einsatz kommen und die Daten im Container gar nicht in einer bestimmter Speicherreihenfolge abgelegt werden.

2.3.3 Spezialisierungen für Container

Zwischen den Iteratoren der verschiedenen Containerklassen existieren zwar Gemeinsamkeiten, die sich aber erst nach einem Studium der Container selbst erschließen, so dass wir hier noch nicht viel dazu sagen können. Ein Iterator wird daher immer eine für einen bestimmten Containertyp spezialisierte Vorlagenklasse sein. Für den Anwendungsentwickler sollte das jedoch transparent sein, das heißt unabhängig vom Container ist der Handgriff für die Deklaration eines Iterators immer die gleiche. Das lässt sich durch folgenden Klassenaufbau des Containers erreichen:

```
template <class T> class container {
public:
    class iterator: public container_iterator<T>;
    ...
};//end class

// oder

template <class T> class container {
public:
    typedef container_iterator<T> iterator;
    ...
};//end class
...
template <class T>
class container_iterator: public Iterator<T> { ... } ;
```

Im ersten Fall wird die Iteratorklasse als interne Klasse des Containers definiert, wobei gegebenenfalls von einer allgemeinen Basisklasse für Iteratoren geerbt wird. Im zweiten – gebräuchlicheren – Fall ist die Iteratorklasse eine selbständige Klasse, für die in der Containerklasse ein interner Typverweis angelegt wird. Für alle Container werden Iteratorobjekte in einer Standardnotation in allen Anwendungen folgendermaßen deklariert:

```
void fu(){
    container<T>::iterator it
    ...
```

Trotz des einheitlichen Aufrufschemas verbergen sich dahinter hochspezialisierte Implementationen.

Neben dem Iteratortyp stellt der Container noch einige weitere Typdefinitionen zur Verfügung

```
typedef T*        pointer;
typedef T const*  const_pointer;
typedef T&        reference;
typedef T const&  reference;
typedef T         value_type;
...
```

Die Typdeklaration fällt in den Containerklassen zwar noch um einiges komplexer aus, jedoch müssen wir für das Verständnis erst noch einige Voraussetzungen legen. Für die Anwendung ist das jedoch nicht weiter interessant, geht es doch darum, die internen Datentypen in anderen Template-Umgebungen zugänglich zu machen, wenn die Template-Parameter selbst Template-Klassen sind (*siehe letztes Teilkapitel*).

2.3.4 Iteratorkategorien

Da nicht alle Operationen auf jedem Containertyp sinnvoll beziehungsweise erwünscht sind, weil das spezielle Containerdesign beispielsweise nur eine alles andere als effiziente Ausführung ermöglichen würde, führt die STL eine Reihe von Iteratorkategorien ein. Iteratoren eines bestimmten Containertyps gehören zu einer bestimmten Kategorie und erben von einer entsprechenden Basisklasse. Für den Fall, dass ein Anwender selbst einmal eine Iteratorimplementation verfassen möchte/muss, ist damit für ein sinnvolles Grundgerüst gesorgt. Die Kategorien sind:

(a) Iteratoren mit wahlfreier Zugriff : Alle definierten Zugriffsmethoden sind erlaubt.
(b) Bidirektionaler Iteratoren : Der Iterator kann nur noch an der Objektkette des Containers „entlanggleiten“, aber keine größeren Sprünge mehr machen (*die arithmetischen Operatoren $\pm$ fehlen*).
(c) Unidirektionale Iteratoren : Der Iterator kann nur noch in einer Richtung bewegt werden (*wobei die Definition zweier unidirektionaler Operatoren für das Abschreiten in beiden Richtungen möglich ist*).

Je nach Kategorie werden eine Reihe von Methoden in der oben angegebenen Methodenliste für den allgemeinen Iteratortyp nicht mehr zur Verfügung gestellt, weil keine effiziente Implementierung möglich ist.

Welcher Kategorie ein Iterator angehört, wird durch eine Typdefinition in der Iteratorklasse spezifiziert. Die Kategorien werden durch die leeren Klassen

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag :
       public input_iterator_tag {};
struct bidirectional_iterator_tag :
       public forward_iterator_tag {};
struct random_access_iterator_tag :
       public bidirectional_iterator_tag {};
```

angelegt. Beachten Sie, dass Kategorien, die freiere Zugriffsmöglichkeiten erlauben, Erweiterungen der einfacheren Kategorien darstellen und von diesen erben. Ist eine Iteratorklasse als `random _access _iterator` deklariert, so müssen nur die für diesen Iteratortyp neuen und zusätzlichen Eigenschaften freigeschaltet werden, während die Eigenschaften eines `bidirectional _iterator` per Vererbung übernommen werden. Die Kategoriezuordnung erfolgt durch

```
template <class T> Iterator {
public:
   typedef random_access_iterator_tag category;
    ...
```

Auch hier ist das konkrete Geschehen in der STL komplizierter. Neben der Kategorie enthält die Iteratorklasse auch Typdefinitionen für den Zugriff auf den internen Datentyp und weitere Spezifikationen, die für die Auswahl der richtigen Algorithmen notwendig sind. Solche Spezifikationen werden als Eigenschaften oder „Traits“ bezeichnet. Wie solche Konzepte eingesetzt werden, diskutieren wir weiter unten.

2.3.5 Iteratoren und konstante Iteratoren

Als weitere Unterkategorien können schreibende oder lesende Zugriffe auf die Objekte im Container ausgeschlossen oder zugelassen werden (*siehe Kap. 3.5*). In der allgemeinen Klasse haben wir verschiedene Elementzugriffe hierzu deklariert, jedoch ist diese Unterscheidung nur dazu gedacht, dem Compiler eine widerspruchsfreie Verwendung in const–Methoden zu ermöglichen. Nicht möglich ist auf diese Weise die Kontrolle, ob in einer Umgebung, in der der Container (*oder besser sein Inhalt*) konstant bleiben soll, auch nur Iteratorfunktionen aufgerufen werden, die dies garantieren.

Iteratoren auf einem Container können nur sinnvoll durch den Container selbst initialisiert werden. Container stellen daher zwei Funktionen zur Verfügung, die Iteratoren auf das erste und hinter das letzte Element liefern. Von diesen Funktionen sind zwei Sätze notwendig, einer für Umgebungen, in denen der Containerinhalt unverändert bleiben soll, und ein zweiter für solche, die eine Änderung der Inhalte

erlauben. Die beiden Funktionssätze geben Objekte zweier verschiedener Iteratorklassen zurück, die nun ihrerseits eine Änderung der Inhalt ermöglichen oder verhindern:

```
template <class T> class Container{
   ...
   iterator       begin();
   const_iterator begin() const;
   iterator       end();
   const_iterator end() const;
```

Die vielleicht naheliegende Vermutung, dass `iterator` von `const_iterator` erbt, ist allerdings falsch (*machen Sie sich klar, dass das Vererbungsschema*

```
template <class T>
class iterator: public const_iterator<T>{...
```

im Prinzip die Kontrolle der Konstanz erlauben würde). Statt dessen wird in der allgemeinen Definition eines Iterators folgende Implementation gewählt:

```
template <class T> class Container {
public:
   typedef T                  value_type;
   typedef Iterator<T*>       iterator;
   typedef Iterator<T const*> const_iterator;
```

Zwischen den Iteratoren ist daher so ohne Weiteres keine Konvertierung möglich, da es sich um vollkommen eigenständige Datentypen handelt, zwischen denen noch nicht einmal Standard-Konvertierungsmechanismen existieren. Deshalb werden auch `cast`–Versuche auf Iteratorebene vom Compiler abgeblockt,das heißt folgende Anweisungen werden nicht akzeptiert:

```
const_iterator ci;
...
iterator it(xxxx_cast<iterator>(ci));
```

Auf Objektebene ist aber eine Konvertierung durch `T* -> T const*` gegeben. Wir können diese vom Compiler ohne Nebenbedingungen akzeptierte Standardkonvertierung ausnutzen, um aus Iteratoren bei Bedarf konstante Iteratoren zu machen.

```
template <class T> classe iterator {
   T* t;
   ...
   template <class U>
   iterator(iterator<U>const& it){ t=it.t ;}
   ...
```

Die Rückgabetypen der Zugriffsoperatoren `operator*` oder `operator->` sind hierdurch je nach verwendetem Iterator wie gewünscht garantiert unveränderbar oder variabel.

2.3.6 Iteratorabstand und Iteratorvorschub

Für den Vorschub eines Iterators um eine bestimmte Anzahl von Positionen bzw. für die Bestimmung des Abstands zweier Iteratoren existieren zwei Funktionen, die mit jedem beliebigen Iteratortyp funktionieren:

```
template<class InIt, class Dist>
    void advance(InIt& it, Dist n);
template<class Init>
    Init::distance_type distance(Init first, Init last);
```

Der Abstand zwischen zwei Iteratoren wird hierbei nicht einfach durch eine Zahl dargestellt, sondern durch einen iteratorspezifischen Typ `distance_type`, der ähnlich den Zeiger- und Datentypen in der Iteratorklasse definiert ist. In den meisten Fällen ist der Abstand natürlich eine ganze Zahl, aber je nach Notwendigkeit kann sich auch etwas anderes dahinter verbergen. Im einfachsten Fall arbeitet `distance` mit der folgenden Implementation:

```
template<class It>
It::distance_type distance(It first, It last){
   typename It::distance_type i = 0;
   for(;first!=last;first++,i++);
   return i;
}//end function
```

Die Methode ist natürlich alles andere als optimal, wenn eine Zeigerarithmetik zur Verfügung steht, denn die würde den Code auf

```
return last - first;
```

reduzieren. Um die jeweils optimale Implementation zu generieren, kommt nun der `category`-Typ ins Spiel, der indirekt aufgerufen wird:

```
template <class It> inline
It::distance_type distance(It first, It last){
   return _d(first,last,typename It::category());
}//end function
```

Die Funktion `_d(..)` existiert in verschiedenen Versionen für die unterschiedlichen Kategorien, von denen ein drittes Objekt im Aufruf erzeugt wird. Da dieses Objekt nirgendwo benötigt wird und alle Funktionen als `inline`-Funktionen deklariert sind, dient die Schreibarbeit nur dem Compiler zur Auswahl der optimalen Methode und verursacht keinen einzigen Assemblerbefehl im fertigen Programm.

Auf ähnliche Art ist auch die Methode `advance` implementiert. Neben den beiden Standardvorgehensweisen – Inkrementierung in Einzelschritten oder direkte Offsetaddition – können auch noch weitere effiziente Methoden existieren, die nur der Bibliotheksentwickler kennt und als Spezialisierung für einen bestimmten Containertyp implementiert.

Aufgabe. Begründen Sie, warum die Zeilen 1 und 2 einen Compilerfehler liefern, wärend Zeile 3 den Abstand zwischen den beiden Iteratoren ausgibt.

```
vector<int>::iterator it;
vector<int>::const_iterator jt;
distance(it,jt);                          // 1
distance<vector<int>::iterator>(it,jt); // 2
distance<vector<int>::const_iterator>(it,jt);// 3
```

Als Anwendungsbeispiel diene die Methode `reverse`, die die Reihenfolge der Elemente eines Containers umkehrt. Der Aufruf erfolgt mit dem Start- und dem Enditerator, wobei der erste nach jedem Schritt inkrementiert, der zweite vor jedem Schritt dekrementiert wird (*er steht ja zu Beginn auf eine ungültigen Position, weshalb erst eine Dekrementierung stattfinden muss*). Das Problem ist, das Ende der Operation zu erkennen, denn je nach Anzahl der Elemente im Container müssen die beiden Iteratoren nicht gleich werden. Mit Hilfe der Methode `distance` ist die Lösung leicht möglich:

```
template <class IT>
void reverse(IT first, IT last){
     int n;
     for(n=distance(first,last);n>0;n-=2){
        --last;
        swap(*first,*last);
        ++first;
     }//endfor
}//end function
```

Die Implementation setzt natürlich voraus, dass die Iteratoren bidirektional sind und der Abstandstyp vom Compiler automatisch nach `int` gecastet werden kann. Abweichungen von diesen Voraussetzungen sind aber nur bei so exotischen Containertypen denkbar, dass eine Invertierung der Objektreihenfolge auf ihnen ohnehin wenig Sinn macht.

2.3.7 Iteratorgültigkeit

Wichtig für die Arbeit mit Iteratoren ist auch die Kenntnis, welche Operationen im Container zulässig sind, damit ein Iterator gültig bleibt (*zeigt der Iterator nach einem Löschen oder Einfügen noch auf eine gültige Position*?). Die Regeln der

STL hierfür entsprechen den Regeln für C–Zeiger, die ja beispielsweise nach einer `Resize`–Operation nicht mehr gültig sind. Konkret: Während Datenzugriffe natürlich problemlos möglich sind, ist nach dem Einfügen neuer (`insert`) oder dem Löschen nicht mehr benötigter Objekte aus dem Container (`remove/erase`) generell davon ausgehen, dass keiner der vorher initialisierten Iteratoren noch auf irgendeine sinnvolle und nutzbare Information verweist und alle Iteratoren zunächst reinitialisiert werden müssen.

Wie immer bei der Arbeit mit Zeigern kann die Gültigkeit aber erst zur Laufzeit erkannt werden, das heißt Unterstützung bei der Implementation gibt es durch den Compiler nicht und die Sorgfalt und Disziplin des Programmierers ist gefragt. Konkret: das Programm

```
ita=cont.begin();
ite=cont.end();
for(it=ita;it!=ite;++it){
   if(*it==15)
     insert(it,16);
   if(*it==20)
     insert(it,21);
}//endfor
```

kann laufzeitfehlerfrei ablaufen, wird aber mit Sicherheit nicht bis zum Ende des Containers laufen. Es kann ablaufen, wenn das Einfügen der zusätzlichen Elemente keine neue Speicherorganisation notwendig macht, also vorzugsweise bei Tests mit wenigen Daten (*weshalb dann von Anfängern wieder das übliche „ Wieso? Es läuft doch!“ kommt*). Es wird nicht mehr ablaufen, wenn der Speicher neu organisiert wird, also beim Kunden.

In der Regel liefern die iteratorändernden Methoden einen neuen gültigen Zeiger zurück. Die folgende Modifikation funktioniert daher immer:

```
it=cont.begin();
while(it!=cont.end()){
   if(*it==15)
      it=insert(it,16);
   if(*it==20)
      it=insert(it,21);
   it++;
}//endfor
```

2.3.8 Spezielle Attributtypen

Im Normalfall verweist ein Iterator eines Containers wie ein C–Zeiger auf eine Variable des Vorlagentyps T. Im Vorgriff auf die Containerklassen sei hier aber noch ein spezieller Datentyp vorgestellt. In einigen Containern werden die Objekte nach

Schlüsseln sortiert, wobei Objekt und Schlüssel ein Paar bilden, konsequenterweise vereinigt in

```
template <class KEY, class OBJ> struct pair{
    KEY first;
    OBJ second;
    pair(){};
    pair(const KEY& k, const OBJ& o):
                          first(k),second(o){};
```

Als Beispiel denke man an eine Liste von Kundenadressen mit dem Namen als Zugriffsschlüssel. Iteratoren auf solchen Containern besitzen den Datentyp `pair` und es ist durch `first` oder `second` weiter zu spezifizieren, worauf man wirklich zugreifen will. Bestimmte Verallgemeinerungen sind bei diesen Containertypen nicht mehr ohne weiteres möglich. Der Leser denke beispielsweise an die elementweise Addition der Inhalte zweier Container, die bei Implementation eines Additionsoperators für die Datenklasse und einem Iterator, der nur auf das Datenelement verweist, problemlos generalisierbar ist.

```
container<T> a,b;
a += b;

// bedeutet sinngemäß
for(it1=a.begin(),it2=b.begin();it1!=a.end();
                                  ++it1,++it2)
 *it1 += *it2;
```

Bei einem Iterator mit zwei Attributen ist es jedoch nur noch schwer möglich, eindeutig in allgemein gültiger Form festzulegen, wie eine Addition durchzuführen ist (*man könnte beispielsweise*

```
if(it1->first == it2->first)
    it1->second += it2->second;
```

festlegen und eine Spezialisierung dafür implementieren. Wir werden Beispiele kennen lernen, die in dieser Form organisiert sind. Das lässt sich jedoch nicht mehr durchhalten, wenn `first` *aufgrund der Neuberechnung von* `second` *ebenfalls neu bewertet werden muss*) .

Vergleiche zwischen verschiedenen Paaren betreffen konsequenterweise beide Partner:

```
inline bool operator==(const <T1,T2>& x,
                       const pair<T1,T2>& y){
    return x.first==y.first && x.second==y.second;
}//end function
```

Hier nicht nur die Schlüssel zu vergleichen hat seinen Grund: Bei Vergleich der Inhalte verschiedener Container müssen Objekte mit gleichen Schlüsselwerten nicht zwangsweise übereinstimmen, und bei Containern, die mehrfach den

gleichen Schlüssel zulassen, muss ausgeschlossen werden, dass es sich nicht um Objektverdopplungen handelt.

Aufgabe. Macht es Sinn, `bool operator<(..)` für den Datentyp `pair<..>` zu definieren? Falls ja, implementieren Sie einen solchen.

2.3.9 Rückwärtsiteratoren

Bei der Programmierung von Schleifen kann die Laufvariable sowohl aufwärts wie abwärts zählen. Um diese Funktionalität auch auf Containern zu gewährleisten, sind in der STL zwei weitere Iteratorklassen definiert:

```
container<T> a;
container<T>:: reverse_iterator  ri;
container<T>::const_reverse_iterator  rit;
for(ri=a.rbegin();ri!=a.rend();++ri) ...
```

Rückwärtsiteratoren werden genauso gehandhabt wie Vorwärtsiteratoren, durchlaufen den Container aber in umgekehrter Richtung (*der Leser beachte: Um ein Element im Container zurückzugehen, wird der Rückwärtsiterator inkrementiert*). Sinngemäß gilt

```
a.rbegin() = a.end()-1
a.rend() = a.begin()-1
```

Die meisten Algorithmen setzen allerdings die Verwendung eines Vorwärtsiterators voraus und kommen mit Rückwärtsiteratoren nicht zurecht, da vielfach wieder Klassen dahinter stecken, die wenig miteinander zu tun haben. Um trotzdem einen Einsatz zu ermöglichen, bieten Rückwärtsiteratoren eine Methode zur Erzeugung eines passenden Vorwärtsiterators an:

```
container<T>::iterator it = ri. base();
```

Aber auch das hat Tücken, denn `it`zeigt nicht auf das gleiche Objekt wie `ri` ! Die Abbildungsfunktion lautet nämlich

```
ri -> obj[k]   ===>   it -> obj[(k+1) mod n]
```

Hierbei ist `k` der Laufindex der Objekte im Container, `n` die Anzahl der Objekte. `rbegin().base()` verweist somit auf `begin()`, und alle weiteren Verweise sind jeweils um ein Element versetzt, das heißt `(++ rbegin()).base()` auf `(end()-1)` usw., bis `(rend()-1)` auf `(begin()+1)` verweist (*machen Sie sich das an einem Beispiel klar und testen Sie das*). Soll beispielsweise ein Objekt, auf das ein Rückwärtsiterator verweist, gelöscht werden, so lautet die dazu gehörende Anweisung:

```
a.erase((++ri).base());
// oder
a.erase(--(ri.base()));
```

Der Inkrementoperator in der ersten Anweisung verändert natürlich den Iterator, was bei Folgeoperationen zu berücksichtigen ist. Für dieses Beispiel ist das relativ gleichgültig, da nach einer Löschoperation die Iteratoren ohnehin nach der allgemeinen Regel nicht mehr brauchbar sind.

2.4 Verwaltung des Objektspeichers

2.4.1 Einführung

Bevor wir uns der Diskussion der verschiedenen Containertypen widmen, klären wir zunächst, in welcher Form die Objekte im Container gespeichert werden, und bei näherer Betrachtung erweist sich das als nicht ganz so trivial, wie es sich anhört. Wir beginnen mit einigen allgemeinen Überlegungen zu den *möglichen* Vorgehensweisen:

1. Die zur Speicherung übergebenen Objekte werden auf lokal im Container vorhandene Variable kopiert. Ein dazu notwendiges lineare Feld wird durch

   ```
   T* ar = new T()[..]
   ```

   erzeugt. Für die Objekte muss ein Zuweisungsoperator definiert sein, um die Daten in den Container zu überführen (*Daten im und außerhalb des Containers sind unabhängig voneinander*).
2. Der Container speichert Zeiger auf Objekte, die

   2.1 wie im ersten Fall Kopien der Objekten außerhalb des Containers sind und dynamisch mittels des `new`–Operators und des Kopierkonstruktors erzeugt werden.
   2.2 von der Anwendung an den Container übergeben werden, das heißt der Container erhält die Besitzrechte an den Objekten und gibt den Speicherplatz beim Löschen der Objekte oder am Ende seiner Gültigkeit frei. In der Anwendung darf nach Ungültigwerden des Containers kein Gebrauch mehr von den Zeigern gemacht werden.
   2.3 im Besitz der Anwendung verbleiben, das heißt die Lebensdauer des Containers muss mit dem der entsprechenden Anwendungsteile abgestimmt werden,
   2.4 Mehrfachreferenzen auf Zeigerobjekte sind (*siehe unten*), das heißt Anwendung und Container teilen sich die Besitzrechte auf ein Zeigerobjekt, das erst mit dem Verschwinden des letzte Eigentümers selbst gelöscht wird..

   In den Fällen 2.2–2.4 sind Änderungen an den Objekten gewissermaßen global. Änderungen an Zeigerobjekten in der Anwendung sind später auch an anderen

Stellen sichtbar, wenn auf das entsprechende Objekt im Container zurückgegriffen wird. Anstelle des in 1. im linearen Fall angelegten Felds wird ein nun Zeigerfeld erzeugt:

```
T** ar = new T*[..];
for(i=0;i<..;++i) ar[i]=0;
```

Den einzelnen Elementen werden bei Belegung mit Objekten nach unterschiedlichen Methoden Zeigerwerte zugewiesen.

3. Es existieren für spezielle Klassen bessere Strategien als 1. oder 2. Die Speicherverwaltung wird deshalb proprietär organisiert, das heißt die Standardfunktionen `malloc/free` und `new/delete` kommen nicht zur Anwendung, sondern beispielsweise

```
T* ar = myAllocFunc();
...
myDeleteFunc(ar);
```

Wir werden die verschiedenen Modelle im Verlauf der Studien in diesem Buch kennen lernen. Die Standardbibliothek unterstützt nur Methode 1, das aber ziemlich geschickt.

2.4.2 Allokator–Klassen

Die unterschiedlichen Variablen- und Containermodelle setzen auch unterschiedliche Speichermodelle voraus, die dem Container in Form eines zweiten `template`–Parameters mitgeteilt werden. Durch die Containerdefinition

```
template<class T, class A = allocator<T>>
    class container {
```

wird eine STL–Standardstrategie als *default*–Strategie vordefiniert. Dabei handelt es sich natürlich nicht bei jedem Containertyp um die gleiche Strategie, wie Sie sich anhand der Diskussion der verschiedenen Typen wohl vorstellen können. Wir werden uns hier in der Diskussion auch zunächst auf den Fall der linearen Felder beschränken. Die STL–Entwickler haben jedoch einige Mühe in die Standardstrategien gesteckt, und man ist in der Regel gut beraten, sich daran zu halten.

Wie kann nun die Speicherstrategie für eine große Menge an C++–Objekten optimiert werden? Die Basisüberlegung besteht darin, zunächst einmal die C–und die C++–Anteile an der Speicherung eines Objektes zu trennen. Im Klartext: Die Bereitstellung des Speicherplatzes eines Objektes wird von der Initialisierung des Speicherplatzes entkoppelt. Die Klasse `allocator` stellt daher zwei Funktionssätze für die Datenverwaltung im Container zur Verfügung, denen zwei Funktionen des Containers entsprechen:

(a) **Kapazität eines Containers**. Die erste Funktionengruppe umfasst die Methoden

```
capacity(..)            // Container-Methode
allocate(..)            // Allocator-Methode
deallocate()            // Allocator-Methode
```

Die Methode `allocate` fordert einen zusammenhängenden Speicherplatz vom Betriebssystem für die Aufnahme einer bestimmten Anzahl von Objekten vom Betriebssystem an. Der Speicherplatz wird nicht initialisiert.

```
T* _buf = (T*) malloc(n*sizeof(T));
```

`deallocate` gibt den Speicherplatz wieder frei.

```
free(_buf);
```

Bei Aufruf der Methode `reserve(..)` können beide Methoden zusammen mit Kopierfunktionen genutzt werden:

```
h=allocate(n_neu);
memcpy(h,_buf,n*sizeof(T));[6]

deallocate(_buf);
_buf=h; n=n_neu;
```

Der so bereit gestellte Speicher kann nun eine bestimmte Anzahl von Objekten aufnehmen, ohne dass neuer Speicherplatz vom Betriebssystem angefordert werden muss. Für die Aufnahme eines Objektes muss nur ein Stück des Speichers initialisiert werden. Der Speicher definiert damit tatsächlich die Aufnahmekapazität eines Containers. Eine Änderung der Kapazität ist mit größerem Aufwand verbunden. So weit möglich, sollte eine entsprechende Planung stattfinden, um weder Speicherplatz zu verschwenden noch großen Aufwand durch ständige Kapazitätsausweitung zu verursachen.

(b) **Größe eines Containers**. Die zweite Funktionengruppe umfasst die Methoden

```
resize ..)              // Container-Methode
construct(..)           // Allocator-Methode
deconstruct()           // Allocator-Methode
```

Die Methode `construct(..)` initialisiert Teile des durch `allocate(..)` bereitgestellten Speicherplatzes. Das geschieht erst dann, wenn neue Variable aus der Reserve zum Beispiel durch Aufruf der Methode `resize(..)` in den genutzten Bereich gezogen werden

```
for(i=k;i<l;++i)
    new (&_buf[i]) T();
```

[6] Da man nicht so genau weiß, was in den Objekten passiert, kann die Methode **memcpy(..)**im allgemeinen Fall eingesetzt werden. **Aufgabe**. Was ist im allgemeinen Fall zu tun und warum?

`deconstruct()` ruft für die nicht mehr benötigten Objekte den Destruktor auf, entfernt aber nicht den Speicherplatz.

```
T* _tp;
for(_tp=&_buf[k];_tp!=&_buf[l];++_tp)
    _tp->~T();
```

Beachten Sie die speziellen Aufrufformen! Beim `new`–Operator bietet C++ die Möglichkeit, die Adresse anzugeben, an der ein Objekt erzeugt wird. Der Konstruktor wird dann an die angegebene Adresse umgeleitet, ohne dass Speicherplatz reserviert wird. Der Destruktor, der ohnehin wesentlich mehr mit normalen Methoden zu tun hat als die Konstruktoren, wird wie eine Methode direkt aufgerufen. Beides dürfte in „normalen“ Anwendungen recht selten auftreten.

Der Sinn der Trennung von Speicherplatzanforderung und Speicherinitialisierung erschließt sich bei der Betrachtung komplexerer Klassen. Deren Objekte haben oft ein recht komplexes Innenleben, das beispielsweise zusätzlichen Speicherplatz in Form von Zeigerattributen und/oder langwierige Initialisierungsvorgänge erfordert. Die Beschränkung auf die benötigte Objektanzahl schont so Ressourcen und Rechenzeit.

2.4.3 Eigene Allokatorklassen

Die Methoden `allocate/construct` sowie `deconstruct/deallocate` werden vom Container immer im oben beschriebenen Sinn ausgeführt. Wenn Sie schon auf den Gedanken kommen sollten, eigene Allokatoren zu implementieren, Sie aber nicht auch noch eine neue Containerimplementation verfassen möchten, müssen sich die eigenen Allokatoren an diese Philosophie halten.

Zum Funktionieren der Container beziehungsweise des Standard–Allokators ist die Implementation und freie Zugänglichkeit von Standard- und Kopierkonstruktor sowie des Zuweisungsoperators der als Template-Parameter verwendeten Klassen notwendig. Umgekehrt ist vom Klassendesigner aber auch zu berücksichtigen, dass der Compiler selbständig Standardkonstruktoren und Zuweisungsoperatoren konstruiert, wenn diese nicht implementiert. Hat die Klasse ein komplizierteres Innenleben, so müssen zumindest die Konstruktoren und Operatoren in der Klassenschnittstelle definiert sein, um bei Fehlen der Implementation einen Compilerfehler auszulösen.[7] Soll ein Zugriff auf diese Methoden gar nicht möglich sein – auch diese Einsatzfälle werden wir kennen lernen – so sind sie im geschützten oder privaten Bereich zu definieren.

---

[7] Möglicherweise sind diese Schnittstellen in der normalen Verwendung der Klassen gar nicht vorgesehen und werden bei der Klassendefinition „vergessen“.

Container mit dieser „Standardausrüstung“ haben außerdem die Eigenschaft, typgenau zu sein und über Vererbung definierte Klassenhierarchien nicht zu unterstützen. Ist ist beispielsweise nicht möglich, eine allgemeine Klasse `Fensterelement` zu definieren und später Schaltknöpfe oder Textfelder, die davon erben, in einem Standardcontainer zu speichern. Für die Nutzung der Vererbung dürfen keine fest definierten Objekte erzeugt werden, sondern man muss mit Zeigern oder Referenzen auf Objekte arbeiten.

Man könnte sich nun Gedanken darüber machen, Allokator-Klassen für Container mit Referenzen oder Zeigern zu konstruieren, um diese Lücke zu schließen. Hieraus resultieren allerdings einige, allgemein nicht völlig zweifelsfrei zu klärende Probleme, wie ein Blick auf die Einleitung zu diesem Kapitel lehrt. Ist eine Referenz noch gültig, wenn man sie verwenden möchte? Wem gehört ein Zeigerobjekt, speziell wer kümmert sich um den Aufruf des Destruktors? Und eine Kopie von sich selbst herstellen kann nur ein Objekt zur Laufzeit selbst, und dazu muss es vom Programmierer auch erst in die Lage versetzt werden.

Natürlich lassen sich auch diese Probleme lösen, und wir werden später elegantere Methoden kennen lernen, die dies tun, jedoch nicht an den Containerallokatoren ansetzen, um allgemeiner einsetzbar zu sein.

> **Aufgabe.** Die Diskussion hat sich auf Containertypen mit linearen Feldern von Objekten beschränkt, d.h. es werden jeweils mehrere Objekte gleichen Typs verwaltet. Wie sind beispielsweise verkettete Listen oder Bäume zu implementieren, in denen die Objekte einzeln verwaltet werden? Macht hier eine Reserve Sinn? Entwerfen Sie Modelle.[8]

2.5 Feld- oder Listencontainer

Die Strategie zur Speicherung von Objekten hängt davon ab, wie Daten in eine vorhandene Menge eingefügt oder aus ihr entfernt werden sollen und wie auf sie zugegriffen werden soll. Beispielsweise kann es notwendig sein, Daten

- an einem Ende des Speichers,
- an beiden Enden des Speichers,
- inmitten der vorhandenen Sequenz

anzufügen oder zu löschen beziehungsweise für Anfügen und Löschen unterschiedliche Strategien zu verfolgen. Der Zugriff auf die Daten kann

sequentiell vorwärts oder rückwärts,

[8] Die Aufgabe stellt gewissermaßen einen Vorgriff auf das noch Kommende dar, weil Sie ja offiziell noch nicht wissen, was eine verkettete Liste oder ein Baum ist. Schauen Sie sich ggf. die Einführungen zu diesen Datenstrukturen an oder merken Sie sich die Aufgabe zur nachträglichen Bearbeitung nach Durcharbeiten dieser Kapitel vor.

- indexorientiert,
- nach einem Schlüsselkriterium

erfolgen. Für (fast) alle angesprochenen Möglichkeiten hält die STL Klassen mit optimaler Strategie bereit. Wir werden in diesem Kapitel zunächst die Speicherstrategien vorstellen.

Darüber hinaus hängt eine optimale Strategie für eine Speicherverwaltung natürlich auch davon ab, wie die Objekte im Inneren strukturiert sind und welche Zugriffsarten besonders häufig auftreten. Da die Objektklassen selbst aber Template-Parameter sind und nur der Anwender weiß, welche Operationen er vorzugsweise benötigt, kann hier nur noch bedingt allgemeine Unterstützung angeboten werden, und der Anwender muss im Bedarfsfall selbst unterstützende Algorithmen liefern. Die STL stellt auch hierzu gewisse Möglichkeiten zur Verfügung.

2.5.1 Felder(STL-Klasse vector)

Der einfachste Containertyp verwaltet ein dynamisches lineares Feld, wie es auch in C deklariert werden. Er entledigt den Anwender zunächst von der Aufgabe, für Erzeugung und Freigabe des dynamischen Feldes sorgen zu müssen. Da sich die Anzahl der Elemente im Container ändern kann, wird eine Aufteilung in genutzte Daten und Reserveplätze vorgenommen:

```
template <class T> struct _vector{
   T* _ar;
   int size;
   int reserve;
};
```

Für die Zeigervariable `_ar` wird Speicherplatz für `reserve` Elemente reserviert, von denen jedoch nur `size` Objekte tatsächlich mittels ihres Konstruktors erzeugt werden.[9] Verringert sich die Anzahl der Elemente, werden die Destruktoren ausgeführt, jedoch ohne den Speicherplatz freizugeben (*siehe letztes Kapitel*). Sind die Reserven ausgeschöpft, so muss beim Hinzufügen weiterer Elemente ein neuer größerer Bereich alloziert, die Objekte umkopiert und der alte Bereich freigegeben werden.

> **Aufgabe.** Schreiben Sie Methoden, die Objekte hinzufügen und auch bei Überschreiten der Größe `reserve` neuen Speicherplatz organisieren. Zum Testen entwerfen Sie eine Klasse, die im Konstruktor und Destruktor dem belegten Speicherplatz charakteristische Werte zuweist.

[9] Bei Standardtypen wie `int` spielt diese Unterscheidung natürlich keine Rolle, da mit dem Speicherplatz bereits alles für die Objekte getan ist. Bei Stringvariablen oder Polynomen ist aber andererseits noch einiges zu tun, bis tatsächlich ein gebrauchsfertiges Objekt vorliegt.

Da dieser Containertyp theoretisch kaum Probleme bereitet, diskutieren wir die Containerklasse `vector`, die zunächst eine Reihe von Typisierungen enthält, die in den weiteren Containerklassen ebenfalls wieder auftauchen und daher hier stellvertretend präsentiert seien.

```
template<class T, class A = allocator<T>>
   class vector {
public:
    typedef A allocator_type;
    typedef A::size_type size_type;
    typedef A::difference_type difference_type;
    typedef A::reference reference;
    typedef A::const_reference const_reference;
    typedef A::value_type value_type;
    typedef __iterator<reference, vector_type>
                         iterator;
    typedef __iterator<const_reference,
              vector_type>  const_iterator;
    typedef reverse_iterator<iterator, value_type,
       reference, A::pointer, difference_type>
                        reverse_iterator;
    typedef reverse_iterator<const_iterator,
       value_type, const_reference,
       A::const_pointer, difference_type>
                         const_reverse_iterator;
```

Die Rolle der Allokatorklasse `allocator` haben wir im letzten Kapitel bereits diskutiert.

Die Größe des belegten und belegbaren Speicherplatzes wird durch eine Reihe von Methoden ermittelt, die ebenfalls in anderen Containertypen wieder auftauchen:

```
void reserve(size_type n);
void resize(size_type n, T x = T());
void clear();

// Abfragemethoden:
size_type capacity() const;
size_type size() const;
size_type max_size() const;
bool empty() const;

// Allokator-Ermittlung
A get_allocator() const;
```

Auch die Iteratoren sind eigenständige Typen, da sie einerseits einer Klassifizierung unterliegen, andererseits vom Allokator das Speichermodell in Erfahrung bringen müssen, um korrekt auf die Daten weisen und auf ein anderes Objekt vorrücken zu können.

Die Konstruktorenliste enthält Erzeugungs- und Kopierkonstruktoren, wobei unter Verwendung von Iteratoren Vektorobjekte aus beliebigen anderen Containern erzeugt werden können.

```
explicit vector(const A& al = A());
explicit vector(size_type n,
                    const T& v = T(),
                    const A& al = A());
vector(const vector& x);
vector(const_iterator first, const_iterator last,
       const A& al = A());
```

Aufgabe. Implementieren Sie den letzten Konstruktor der Liste.

Objektzugriffe sind über Indexfunktionen

```
reference at(size_type pos);
const_reference at(size_type pos) const;
reference operator[](size_type pos);
const_reference operator[](size_type pos);
reference front();
const_reference front() const;
reference back();
const_reference back() const;
```

oder Iteratoren möglich. Start- und Endwerte für Iteratoren werden vom Containerobjekt durch die Methoden

```
iterator begin();
const_iterator begin() const;
iterator end();
iterator end() const;
reverse_iterator rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator rend();
const_reverse_iterator rend() const;
```

gegeben. Das automatische Anfügen oder Löschen von Objekten am Ende des Vektors erfolgt durch die Methoden

```
void push_back(const T& x);
void pop_back();
```

Zu beachten ist, dass sich bei Änderung der Anzahl der gespeicherten Elemente in jedem Fall der auf das Ende zeigende Iterator ändert, bei Änderungen, die auch den Reservebereich betreffen, alle Iteratoren ihre Gültigkeit verlieren. In einer korrekten Implementation sind daher nach allen Operationen, die mit einer Größenänderung des Containers verbunden sind, sämtliche in Gebrauch befindliche Iteratoren neu zu initialisieren.

Aufgabe. Implementieren und Testen Sie einige der Methoden für Ihren Übungsvektor. Implementieren Sie einige Testbeispiele mit dem STL-Container, um sich mit den Funktionen vertraut zu machen.

Führen Sie dies im Weiteren auch für andere Methoden und Container aus, ohne dass ich dies jedes Mal als Aufgabe formuliere. Die eine oder andere Feinheit zeigt sich erst beim Probieren und nicht schon beim Drüberweglesen, weshalb ein wenig Testen vor dem ersten Realeinsatz meist recht sinnvoll ist.

Eine Reihe von Methoden erlauben den Austausch der Inhalte mehrerer Variabler gleichzeitig. Die Methode

```
void assign(const_iterator first,
            const_iterator last);
```

tauscht beginnend ab dem ersten Element die Inhalte der Feldvariablen gegen die durch die Iteratorsequenz gegebenen Inhalte aus, entspricht also einer Zuweisung, die Methode

```
void assign(size_type n, const T& x = T());
```

ersetzt **n** Elemente durch den angegebenen Wert. Der Austausch der Inhalte zweier Vektoren wird durch

```
void swap(vector x);
```

durchgeführt. Die folgenden Methoden erlauben das Einfügen oder Löschen von Sequenzen ab der ersten Iteratorposition, das heißt die folgenden Elemente werden auf dem Vektor entsprechen verschoben

```
iterator insert(iterator it, const T& x = T());
void insert(iterator it, size_type n, const T& x);
void insert(iterator it,
       const_iterator first, const_iterator last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

Viele der Methoden, die die Containergröße verändern, liefern als Rückgabewert einen neuen Iterator zurück, der dem entspricht, mit dem die Methode aufgerufen wurde. Mit ihm kann beispielsweise eine Schleife fortgesetzt werden, wobei aber auch darauf zu achten ist, dass die Abbruchbedingung durch Aufruf der Methode `end()` bearbeitet wird.

Betrachten wir abschließend die Laufzeiteigenschaften eines Vektors bei grundlegenden Operationen:

(a) Indizierter Zugriff: O(1)
(b) Einfügen/Löschen am Ende: O(1)
(c) Einfügen/Löschen am Anfang oder im Inneren: O(n).

Um Platz zu schaffen, müssen nämlich alle Daten jenseits der Einfügeposition um ein Feld nach oben geschoben werden; beim Löschen verhält es sich umgekehrt.

Ein Vektor eignet sich somit für Algorithmen, die auf Elemente in beliebiger Reihenfolge zugreifen müssen, sowie für dynamische Container, deren Elemente nach dem LIFO (last in-first out)-Mechanismus ausgetauscht werden.

2.5.2 Segmentierte Felder (STL-Klasse deque)

Für dynamische Container, die auch am Anfang oder gar im Inneren häufig Elemente hinzufügen oder löschen müssen, ist eine Vektorstruktur wenig geeignet. Wenn auch der indizierte Zugriff weiterhin gewährleistet werden soll, behilft man sich durch einen segmentierten Vektor. Jedes Segment besitzt eine vorgegebene Kapazität, so dass Lösch- und Einfügeoperationen berechenbar sind; bedarfsweise werden weitere Segmente reserviert oder überflüssige gelöscht.

Für die Segmentierung sind zwei Strategien möglich. Eine Datenstruktur für beliebige Entnahmen oder Hinzufügungen ist

```
template <class T> class _deque{
    struct segment{
        T* beg, *dat, *end};
    segment* beg, *dat, *end;
```

Jedes Segment ist ein eigenständiger kleiner Vektor (und kann auch als Instanz eines Vektors implementiert werden), und je nach Position des Einfügens oder Löschens müssen innerhalb des Segmentes die Elemente wie beim Vektor nach Rechts verschoben werden. Die Zeiger `beg, dat` und `end` zeigen jeweils auf die Startadresse, hinter das letzte belegte und hinter das letzte reservierte Speicherelement (Iteratorkonvention). Ist ein Segment durch Anfügen von Daten gefüllt, wird ein weiteres Segment eingefügt und in der Segmentverwaltung eingetragen. Das Segment kann an beliebiger Stelle in der Gesamtkette eingefügt werden und enthält das beim Füllen nach Rechts hinausgeschobene Element.

Zum Entfernen eines Elementes wird es lediglich aus dem Segment entfernt, in dem es sich befindet. Wie beim Vektor wird der Rest der Elemente nach Links aufgerückt. Ist ein Segment vollständig geleert, wird es aus der übergeordneten Liste entfernt.

Aus dieser Beschreibung folgen die Laufzeitordnungen

(a) Einfügen/Löschen: O(1). Die Segmentgröße ist eine festgelegte Konstante, woraus eine konstante Laufzeit resultiert, die bei zu größen Segmentgrößen aber nicht sonderlich effektiv ist.
(b) Indizierter Zugriff: O(n). Da jedes Segment eine andere Elementanzahl enthalten kann, müssen beim indizierten Zugriff sämtliche Segmente in der Segmenteliste durchgezählt werden, bis das Segment mit dem gewünschten Elemente gefunden ist. Formal resultiert daraus die Ordnung O(n), die jedoch,

da die Anzahl der Summanden von der Anzahl der Segmente abhängt, nicht sonderlich störend wirkt.

Aufgabe. Implementieren Sie eine solche segmentierte Liste mit Hilfe der Vektorklasse. Die Segmente sind vom Typ `vector<T>`, die Segmentliste vom Typ `vector<vector<T> >`. Implementieren Sie den Indexoperator.[10]

Neben dieser Strategie kann man auch eine weitere anwenden, bei der alle mittleren Elemente immer vollständig gefüllt sind und das erste von Links leerläuft. Ein Rückgriff auf die Struktur Vektor ist bei der Implementation nicht mehr möglich, da sich die Segmente nicht mehr wie Vektoren verhalten. Diese Strategie eignet sich für einen reinen FIFO-Mechanismus und weist folgende Laufzeitordnungen auf:

(a) Indizierter Zugriff: O(1), da nicht mehr über die mittleren Segmente summiert werden muss. Nach Untersuchung des ersten Segmentes kann die Position des gesuchten Elementes durch eine ganzzahlige Division mit Rest ermittelt werden.
(b) Einfügen/Löschen am Beginn/Ende: O(1). Die Operation ist effektiver als beider ersten Strategie, da auch Schiebeoperationen auf den Segmenten entfallen.
(c) Einfügen/Löschen im Inneren: O(n). Die Operation ist weniger effektiv als beim Vektor, da jeweils die Segmentgrenzen überwunden werden müssen und der Aufwand daher dem des einfachen Vektors entspricht.

Aufgabe. Entwerfen Sie eine Datenstruktur für einen Container diesen Typs. Entwerfen Sie einen Algorithmus für den Indexoperator auf dem Container.

Eine grundsätzliche Änderung gegenüber dem ersten Vektormodell ist der nun nicht mehr zusammenhängende Datenbereich. Als Konsequenz sind Iteratoren auf diesem Container im Gegensatz zur Vektorklasse keine einfachen Zeiger mehr, sondern echte Klassenobjekte, die sich um einiges kümmern müssen. Ein denkbarer Aufbau für den ersten Typ (unter Berücksichtigung der Verwendung der Vektorklasse) besteht in

```
template <class T> class DequeInterator {
   vector<T>::iterator            ist, seg_ende;
   vector<vector<T> >::iterator   it_liste, list_ende;
   ...
T const& operator*() const { return *ist; }
DequeIterator<T>& operator++(int){
   ++ist;
```

[10] Ich hätte statt der Datenstruktur auf der Basis von Zeigern auch gleich diesen Implementationsweg wählen können, zumal damit auch viele Funktionen, um die man sich sonst selbst kümmern muss, entfallen. Die Darstellung alternativer Möglichkeiten passt jedoch gut zum Charakter eines Lehrbuches. **Aufgabe**. Welche Funktionen fallen fort, da in Vektor vorhanden, und wie hätten sie ausgesehen.

```
    if(ist==seg_ende){
      ++it_liste;
      if(it_liste != list_ende){
        ist=it_liste->begin();
        seg_ende=it_liste->end();
      }
    }
    return *this;
}
```

Der Zugriff auf das Element wird an das Iteratorattribut `ist` durchgereicht. Beim Inkrementieren muss überprüft werden, ob das Ende des aktuellen Segments erreicht ist, um auf das nächste Segment (soweit noch vorhanden) zu verzweigen. Noch komplexer sind Differenzen von Iteratoren oder Sprünge um mehrere Einheiten nach vorne oder hinten.

Die Schnittstelle der Klasse `deque` stimmt mit der von `vector` überein und implementiert nur weitere Methoden zur Bedienung des Feldbeginns:

```
void push_front(const T& x);
void pop_front();
```

Die Methoden `reserve(..)` und `capacity()` sind nicht mehr notwendig, da das Feld segmentweise verwaltet und erweitert wird, und deshalb in der Klassendefinition auch nicht mehr vorhanden.

Aufgabe. Implementieren Sie die Klasse mit den Methoden `push _back/push _front, pop _back/pop _front, insert, erase` sowie Iteratoren mit Hilfe von Vektorobjekten nach dem ersten Implementationsmuster.

2.5.3 Warteschlangen (STL-Klassen Stack und Queue)

Auf der Grundlage von `vector<..>` und `deque<..>` sind in der STL weitere Containerklassen für spezielle Warteschlangen oder Datentypen realisiert, die an dieser Stelle ebenfalls vorgestellt werden sollen. Dabei wird teilweise die vorhandene Schnittstelle auf den Umfang reduziert, der für die spezielle Warteschlange notwendig ist, teilweise allgemeine Algorithmen als Spezialisierungen in die Klasse übernommen.

Die Klasse stack stellt eine Schnittstelle für eine ausschließliche LIFO–Speicherliste zur Verfügung:

```
template<class T,class Cont = deque<T>>
    class stack {
public:
    ...
     bool empty() const;
```

```
    size_type size() const;
    value_type& top();
    const value_type& top() const;
    void push(const value_type& x);

    void pop();
};//end class
```

Es können nur an einer Seite Objekte hinzugefügt oder entfernt werden, wobei der Zugriff auf das letzte Objekt mit der Methode `top()` möglich ist.

Das Gegenstück mit gleicher Schnittstelle, aber Entnahme der Objekte am vorderen Ende der Schlange (FIFO–Warteschlange) heißt `queue`.

Anmerkung. Die beiden Warteschlangen assoziiert man in der Regel mit den Speicherstrukturen „verkettete Liste“ oder „doppelt verkettete Liste“. Die Listencontainer verursachen jedoch einen größeren Aufwand bei der Speicherverwaltung. Wir werden uns in einem späteren Kapitel auch um solche Probleme kümmern. Technisch werden Warteschlangen häufig in „engen Umgebungen“ wie Betriebssystemen oder Mikrocomputeranlagen benötigt, in denen man sich eine aufwändige Freispeicherverwaltung nicht leisten kann oder will. Mit den festen Speicherstrukturen von Feldern oder segmentierten Feldern ist man hier deutlich besser bedient..

2.5.4 Bitfelder

Bei der Verwaltung von (*logischen*) Schaltern, die nur die Werte `true` oder `false` annahmen können, genügt ein Bit für die Informationsspeicherung. Als weitere Spezialisierung von `vector` enthält die STL dafür die Klasse bit_vector, die funktionsmäßig etwa `vector<bool>` entspricht, aber für bitweise Behandlung des Feldes ausgelegt ist. Die Schnittstellen entsprechen weitgehend den Hauptklassen.

> **Aufgabe.** Eine Liste von Primzahlen kann man so anlegen, dass eine Zahl als Index in einem Bitfeld verwendet wird und der Inhalt 0 oder 1 signalisiert, ob die Zahl keine Primzahl ist oder doch. Die Liste wird so initialisiert, dass sie nur ungerade Zahlen enthält, d.h. zur Prüfung, ob 5.129 eine Primzahl ist, ist der Index [5.129/2]=2.564 zu verwenden.
>
> Zum Erzeugen der Liste wird das Feld mit 1 gefüllt. In zwei geschachtelten Schleifen werden nun bestimmte Positionen gelöscht. Ist beispielsweise an der Position k eine 1 eingetragen, so entspricht dies der Zahl (2^*k+1), und diese ist eine Primzahl. Alle Zahlen im Abstand (2^*k+1) sind durch diese Zahl teilbar, d.h. die Bits sind an diesen Positionen zu löschen. Diese Methode wird „Sieb des Erathostenes“ genannt. Implementieren Sie eine solche Liste.

2.5.5 Zeichenketten Strings

Ebenfalls eine Spezialisierung der Klasse `vector` ist die Klasse `basic_string`, die in Anwendungen meist in der Spezialisierung `string` mit dem Datentyp `char` auftritt, aber für anderen Schriftsysteme, die wie das chinesische nicht mit ca. 100 Zeichen auskommen, auch mit anderen Typen realisiert werden kann.

In dieser Klasse werden eine Reihe weiterer Methoden implementiert, die im Zusammenhang mit einer Stringverarbeitung benötigt werden, das heißt es werden nicht individuelle Eigenschaften der gespeicherten Objekte untersucht, sondern ganze Objektfolgen ausgewertet. Neben den Einfüge- und Löschoperationen sind Methoden zum Aneinanderfügen von Zeichenketten, Finden bestimmter Zeichenketten ab vorgegebenen Positionen, Austauschen und Ketten unterschiedlicher Längen usw. definiert:[11]

```
// Verketten von Zeichenketten (ZK)
   basic_ string& operator+=(
                  const basic_ string& rhs);
   basic_ string& append(const basic_ string& str);
// Austausch von n0 Zeichen ab p0 durch andere ZK
   basic_ string& replace(size_ type p0,
                          size_ type n0,
                          const basic_ string& str);
// Identifizieren der Position bestimmter ZK
   size_ type find(const basic_ string& str,
                   size_ type pos = 0) const;
   size_ type rfind(const basic_ string& str,
                    size_ type pos = npos) const;
// Identifizieren der Position bestimmter Zeichen
// in einer ZK aus einer vorgegebenen Menge
   size_ type find_ first_ of(const basic_ string& str,
                     size_ type pos = 0) const;
   size_ type find_ last_ of(const basic_ string& str,
                     size_ type pos = npos) const;
   size_ type find_ first_ not_ of(
                     const basic_ string& str,
                     size_ type pos = 0) const;
   size_ type find_ last_ not_ of(
                     const basic_ string& str,
                     size_ type pos = npos) const;
```

[11] Wer sich mit „regulären Ausdrücken“, d.h. den Stringverarbeitungsanweisungen von Skriptsprachen wie Perl auskennt, wird sich vermutlich ein Schmunzeln ob diesen Funktionsumfangs nicht verkneifen können. Entsprechende Bibliotheken existieren natürlich auch in C, jedoch darf man wohl anmerken, dass bei einem nur gelegentlicher Bedarf von komplexeren Stringverarbeitungen ein Zusammenstellen der notwendigen Funktionalität auf Basis dieser Funktionen schneller zu realisieren ist als eine Einarbeitung in reguläre Ausdrücke und ihre Verwendung.

```
// Teil-ZK und Vergleiche von ZK
   basic_ string substr(size_ type pos = 0,
                        size_ type n = npos) const;
   int compare(const basic_ string& str) const;
```

Die Methoden sind jeweils in mehreren Versionen implementiert, die als Argumente andere Objekte des Typs `basic _string`, Felder, C–Strings (*sofern sinnvoll*) oder Iteratoren erlauben.

```
template<class E,class T=char_ traits<E>,
            class A=allocator<T>>
class basic_ string {
public:
    ...
    basic_ string& append(const basic_ string& str);
    basic_ string& append(const basic_ string& str,
        size_ type pos, size_ type n);
    basic_ string& append(const E *s, size_ type n);
    basic_ string& append(const E *s);
    basic_ string& append(size_ type n, E c);
    basic_ string& append(const_ iterator first,
                          const_ iterator last);
```

Die Stringklasse besitzt wie ein `char`-Array in C einen ambivalenten Charakter. `basic_string` entspricht nicht einem C–String, das heißt ein Nullelement bedeutet nicht das Ende der Sequenz, sondern es können beliebige Zeichenketten verwaltet werden, die auch Nullelemente enthalten dürfen (*Start- und Endezeiger sind dafür natürlich notwendig*). Die Beschränkung auf C–typische Strings mit $'\backslash 0'$ als Endekennung wird nur in stringtypischen Methoden durchgeführt. So haben die beiden Methoden

```
const E* c_str();
const E* data();
```

als Rückgabewert beide einen Zeiger auf den Beginn der Zeichenkette, wobei aber `c_str()` dafür sorgt, dass die Kette durch eine Null abgeschlossen ist. Bei allgemeinen Zeichenketten muss das bei `data()` nicht der Fall sein, d.h. die Abschluss-Null des Strings ist im Datenpuffer gar möglicherweise nicht vorhanden bzw. nach der Null sind weitere Datenvorhanden, die von den Stringfunktionen nicht erfasst werden. Dazu passend liefern

```
int length() const;
int size() const;
```

die Längen von Strings bzw. die Anzahl der Datenworte im String.

Nun könnte man meinen, dass damit bereits alles zu Strings gesagt ist, aber das gilt nur, wenn wir über ASCII-Strings sprechen. Schon deutsche Umlaute sorgen für gewisse Probleme beim Sortieren und Vergleichen von Strings, von chinesischen oder anderssprachigen Zeichen, deren Anzahl den Zahlenumfang von Char wesentlich übersteigen, ganz zu schweigen.[12] Für die korrekte und sprachrichtige Zeichenkettenauswertung wesentlich ist daher der Vorlagenparameter char_traits<..>, in dem festgelegt wird, welchen Eigenschaften die Zeichen einer Kette haben und wie eine Zeichenkette auszuwerten ist. Genauer: Der Templateparameter von `char_traits<..>` gibt den Datentyp der Zeichen in der Kette an, in `char_traits<..>` werden für die Arbeit in der Stringklasse notwendige Zugriffsmethoden und Algorithmen hinterlegt.

```
struct char_traits<E> {
    ...
    static void assign(E& x, const E& y);
    static E *assign(E *x, size_ t n, const E& y);
    static bool eq(const E& x, const E& y);
    static bool lt(const E& x, const E& y);
    static int compare(const E *x,
                       const E *y, size_ t n);
    static size_ t length(const E *x);
    static E *copy(E *x, const E *y, size_ t n);
    static E *move(E *x, const E *y, size_ t n);
    static const E *find(const E *x, size_ t n,
                         const E& y);
    static E to_ char_ type(const int_ type& ch);
    static int_ type to_ int_ type(const E& c);
    static bool eq_ int_ type(const int_ type& ch1,
                      const int_ type& ch2);
    static int_ type eof();
    static int_ type not_ eof(const int_ type& ch);
};//end class
```

Die Methoden enthalten das, was man auf den ersten Blick vermuten darf, also in Bezug auf den Datentyp `char` ziemliche Trivialitäten, aber auch die Konventionen für C–Strings. Spezielle Festlegungen sind aber beispielsweise notwendig für

[12] Die Kodierungssysteme in Anwendungen sind meist nicht einheitlicher Natur und können auch nicht mit diesem Eigenschaftssystem verwaltet werden. Beispielsweise kann der Umlaut „ä“ in HTML-Texten durch „ä“ kodiert werden, d.h. die dargestellten Zeichen besitzen interne Darstellungen unterschiedlicher Längen (und es gibt noch weitere Kodierungsnormen). Bitte verwechseln Sie diese Kodierungsvorschriften nicht mit den Eigenschaftssystem, das hier diskutiert wird.

erweiterte Alphabete wie chinesische/japanische Schrift, Hieroglyphen und sonstige Schriften, die Silben oder ganze Worte oder mehr als 255 Zeichen umfassen.
`traits<..>` oder Eigenschaften werden für viele Datentypen definiert und enthalten allgemeine Informationen, mit denen der Compiler automatisch korrekte Entscheidungen in Algorithmen treffen kann. Wir werden solche Eigenschaftssammlungen später auch unter anderen Namen kennenlernen. Hier nur einige Beispiele

- `traits<Typ>::is _integral` unterscheidet Datentypen, mit denen exakt gerechnet werden kann (*beispielsweise ganze Zahlen*), von solchen mit Rundungsfehlern (`double`).
- `traits<Typ>::sum _type` legt fest, wie bei Summation zu verfahren ist. Beispielsweise führen Summen über mehrere Elemente bei den Datentypen `char, unsigned char, short, unsigned short` relativ schnell zu Überlauffehlern, was durch den Summentyp `long int` oder `long long int` verhindert werden kann.
- `traits<Typ>::has _sign` unterscheidet Datentypen mit und ohne Vorzeichen.
- ...

`trait`-Klassen bieten also die Möglichkeit, individuelle Typeigenschaften und typspezifische Funktionen für beliebige Fälle zu definieren. Sie können anstelle allgemeiner Template-Klassen oder zur Unterstützung der Auswahl von Template-Klassen eingesetzt werden. Die Zuteilung individueller Eigenschaften zu Datentypen zur korrekten Bearbeiten bestimmter Aufgaben ist allerdings auch mit einem Preis verbunden: implementiert man eine Funktion mit einer bestimmten Klasse und definiert dazu ein neues Attribut, so verweigert sich der Compiler beim dem Versuch, die Methode auch mit alten Klassen, die das Attribut noch nicht kennen, zu implementieren. Um das zu Umgehen, müssen die neuen Attribute in die Eigenschaftslisten der alten Klassen, teilweise sogar der Systembibliotheken, eingepflegt werden. Bei größeren Klassenbibliotheken wird man das eher bedarfsweise als generell machen. Ein bedarfsweises Einpflegen ist aber oft zeitlich deutlich von der Primärimplementation getrennt, so dass oft die Klasse, die Anlass zu der Spezialität gegeben hat, nicht mehr bekannt ist und sich der Sinn erst aus dem Studium der Implementation der Methode ergibt. Das Anschauen fertigen Kodes gehört aber nun gerade zu den Tätigkeiten, die man vermeiden sollte. `trait`-Klassen sollten also eher vorsichtig definiert werden.

2.5.6 Objekte und Zeiger in Containern

Als weitere Rahmenbedingungen von Feldspeichern der diskutierten Typen mit Reserve sind nach dem bisherigen Entwicklungsstand der Theorie zu notieren:

(a) Alle Objekte sind vom gleichen Typ, das heißt es ist nicht möglich, Vererbungshierarchien zu berücksichtigen und polymorphe Objekte dynamisch zur Laufzeit zu verwalten.

```
class A { ... };
class B: public A { ... };
container<A> a;
B b; a.insert(b); // geht nicht, nur Typ a vorhanden
```

(b) Das Einfügen in mittlere Positionen ist nach wie vor recht aufwendig, wenn die Objekte viel Speicherplatzbedarf haben, der beim Verschieben kopiert werden muss.

Wie aus den technischen Details zur Vererbung bekannt ist, kann das Problem (a) durch Verwendung von Zeigerobjekten beseitigt werden.

```
container<A*> a;
a.insert(new b());
```

Bei Einfügen oder Löschen im mittleren Bereich müssen nun auch nur Zeigerwerte anstelle kompletter Objekte kopiert werden.

Das Problem einer solchen Vorgehensweise sind allerdings die Eigentumsverhältnisse. Kann ein Zeigerobjekt in einer Anwendung gelöscht werden oder ist es noch in einem Container vorhanden und muss folglich existent bleiben? Umgekehrt ist die gleiche Frage möglich. Außerdem benötigen wir eine spezielle Version des Containers oder besser der Allokatorklasse, die weiß, dass sie den `delete`–Operator verwenden muss.

Aufgrund dieser Probleme, die innerhalb der Containerklassen nicht sicher gelöst werden können, muss von Deklarationen wie

```
vector<MyClass*>mv;
```

dringend abgeraten werden. Wir werden später Techniken entwickeln, die es uns erlaubt, auf spezielle Containerversionen für Zeigervariable oder besondere Verwaltungseinrichtungen zu verzichten. Trotzdem sei zu Übungszwecken ein Aufgabe dazu gestellt:

> **Aufgabe.** Entwickeln Sie eine Allokatorklasse, die Zeigerobjekte aus dem Container entfernt (alle gespeicherten Objekte gehen in den Besitz des Containers über). Wie können Zeigerobjekte in Container ausgetauscht werden, ohne dass es zu Speicherproblemen kommt?

2.5.7 Verkette Listen (STL-Klasse list)

Eine andere Speicherstrategie, bei der Einfügen und Löschen an beliebiger Position mit der Laufzeitordnung O(1) durchführbar ist, das Auffinden eines bestimmten

Datenobjekts über einen Index aber nur von der Ordnung O(n) ist,[13] ist die verketteten Liste.

```
template <class T> struct Node {
   T object;
   Node<T>* next;
};

template <class T> struct List {
   ...
   Node<T>* anchor;
   ...
```

Die Daten werden in einem Attribut der Struktur `Node` gespeichert, die als Zeigervariable erzeugt wird und einen Zeiger auf das nächste gleichartige Objekt besitzt. Der Container selbst besitzt eine „Ankervariable “, die auf das erste Glied der Kette zeigt. Soll auf ein anderes als das erste Objekt der Kette zugegriffen werden, ist ein sequentielles Durchgehen notwendig, das heißt der Aufwand ist höher als in den bisherigen Containertypen. Das Speicherschema eignet sich daher vorzugsweise für Anwendungen, in denen Objekte nach dem Schema „last in–first out “ gespeichert werden (`stack`–*Strukturen*). Die Objekte können als Typattribute eines Listenelementes oder als Zeigerobjekte implementiert werden.

Listen sind ebenfalls geeignete Speicherstrukturen, wenn die Objekte relativ große sind und häufig neu angeordnet werden. Bei Änderung der Ordnung oder Zusammenlegen von Listen müssen dann nicht komplette Objekte erzeugt und vernichtet, sondern nur Zeiger an eine andere Stelle verschoben werden.

Durch einen zweiten Zeiger in der Struktur `Node` lässt sich die Liste in eine doppelt verkettete Liste erweitern, die von beiden Seiten gleichmäßig bedient werden kann. Die Liste wird im Container an beiden Seiten verankert, innerhalb der Liste sind Bewegungen in beide Richtungen möglich. Der Aufwand zum Finden eines bestimmten Elementes steigt linear mit der Anzahl der Glieder an.

Die beiden Typen sehen zwar recht ähnlich aus, die doppelte Verkettung bewirkt aber doch recht unterschiedliche Eigenschaften. Soll beispielsweise ein Element M vor ein Objekt N in eine Liste eingefügt werden, so haben die Algorithmen folgendes Aussehen:

Einfache Verkettung	Doppelte Verkettung
`for(i=c.begin();i!=N;++i);`	`M->last=N->last;`
`M->next=N;`	`M->last->next=M;`
`i->next=M;`	`N->last=M;`
	`M->next=N;`

[13] Woraus die Gesamtordnung O(n) resultiert, wenn die Position erst noch gesucht werden muss.

Da der Aufwand doppelt verketteter Listen gegenüber einfach verketteter Listen gering ist, die Vorteile bei der Bedienung aber erheblich, werden einfach verkettete Listen in der Regel nicht implementiert. Grundlage der Implementation einer doppelt verketteten Liste ist das folgende Klassengerüst:

```
template <typename T> class Dlist {
public:
   Dlist():front(0),back(0){}
   ~Dlist(){ if(front) delete front;}
private:
   struct Node {
      T obj;
      Node *prev, *next;
      Node(T const& t): obj(t),prev(0),next(0){}
      ~Node(){ if(next) delete next; }
   } *front, *back;
};//end class
```

Aufgabe. Implementieren Sie auf dieser Grundlage eine doppelt verkettete Liste mit einigen Zugriffsfunktionen (`size()`, `push _back(T const&)`, `push _front(T const&)`, `insert(iterator)`, `delete(iterator)`) und Vorwärts- und Rückwärtsiteratoren.

Eine Stärke von Listen ist das Zusammenlegen mehrerer Listen. Das Anfügen einer Liste an eine andere, bei den Containern `vector` und `deque` von der Laufzeitordnung O(n), ist hier O(1), da lediglich der Anfang der einen Liste an das Ende der anderen gehängt werden muss. Auch das sortiere Zusammenfügen ist einfach: es muss in jedem Schritt nur das kleinere (oder größere) Element der beiden Listen an das Ergebnis angefügt werden, was einen Laufzeitaufwand O(n) ergibt. Dies funktioniert zwar auch bei den anderen Containertypen, ist wegen die Kopieraufwandes aber in der Regel trotzdem zeitaufwändiger.

In der STL definiert ist die doppelt verkettete Liste `list<..>`. Die Schnittstelle hält wenig Überraschungen bereit. Außer den Indexzugriffen finden wir alle Iteratoren sowie Zugriffsmöglichkeiten auf die Enden der Kette wie bei `deque` wieder. Auch die Methoden

```
void resize(size_type n, T x = T());
size_type size() const;
bool empty() const;
```

sind implementiert und erzeugen wie gewohnt eine Liste der angegebenen Größe (*beziehungsweise hängen entsprechend viele Elemente an das Ende der vorhandenen Liste an*) beziehungsweise ermitteln die Größe. Das Innenleben der Methoden kann allerdings kaum noch auf Formeln zurückgreifen, sondern basiert auf Abzählungen. Einfügen und Löschen von Elementen oder Elementsequenzen erfolgt mit den bekannten Methoden

```
iterator insert(iterator it, const T& x = T());
void insert(iterator it,
            size_ type n, const T& x);
void insert(iterator it,
            const_ iterator first,
            const_ iterator last);
void insert(iterator it,
            const T *first, const T *last);
iterator erase(iterator it);
iterator erase(iterator first, iterator last);
```

Der Transfer von Elementen einer anderen Liste in einer gegebene erfolgt mit der Methode

```
void splice(iterator it, list& x, iterator first);
void splice(iterator it, list& x, iterator first,
               iterator last);
```

„Transfer“ bedeutet, dass die Elemente gleichzeitig aus der Quellliste entfernt werden. Bei allen diesen Methoden wird ähnlich wie bei den vorher diskutierten Containern keine Sortierung vorgenommen, sondern es werden nur die angegebenen Iteratorpositionen verwendet. Weitere Methoden untersuchen allerdings auch den Inhalt der Objekte.

```
void remove(const T& x);
void remove_if(binder2nd<not_equal_to<T>> pr);
```

löscht alle Objekte, die mit `x` übereinstimmen oder die in der zweiten Methode angegebenen Bedingung erfüllen. Wie die Bedingung anzugeben ist, wird in Kap. 4.6 bei den Algorithmen erläutert. Die Methoden

```
void unique();
void unique(not_equal_to<T> pr);
```

löschen Mehrfacheinträge in der Liste (*es bleibt aber ein Element erhalten*). Das Sortieren einer Liste oder das sortierte Zusammenfügen zweier Listen übernehmen[14]

```
void sort();
template<class Pred> void sort(greater<T>pr);
void merging(list& x);
void merge(list& x, greater<T>pr);
```

Aufgabe. Implementieren Sie einen Algorithmus zum sortierten Zusammenführen von Listen.

[14] Wie Sortieralgorithmen funktionieren, werden wir weiter unten diskutieren.

Trotz ihrer Beliebheit in der Theorie und insbesondere bei Programmierübungen von Anfängern sind Listen recht isolierte Containertypen innerhalb der STL und besitzen in der Praxis eigentlich nur beim sortierten Zusammenführen von Containern eine größere Bedeutung.

> **Aufgabe.** Die Segmentverwaltung in segmentierten Feldern des Typs 1 kann auch mit einer List realisiert werden, d.h.
>
> ```
> vector<vector<T> >
> ```
>
> wird durch
>
> ```
> list<vector<T> >
> ```
>
> ersetzt. Realisieren Sie dies.

2.6 Bäume

2.6.1 Teilordnung und Vollordnung

Listencontainer, als sowohl Felder als auch verkettete Listen, lassen sich zwar sortieren, und in sortieren Feldcontainern lässt sich ein bestimmtes Objekt auch sehr effektiv lokalisieren, wie wir noch sehen werden, das Einfügen und Löschen eines Elementes lässt sich aber nicht besser als mit der Laufzeitordnung O(n) realisieren, weil alle Elemente jenseits der Einfüge- oder Löschposition um eine Einheit verschoben werden müssen.[15] Baumcontainer beseitigen diesen Mangel, jedoch machen Baumcontainer, wie wir sehen werden, unsortiert wenig Sinn, serielle Zugriffe sind programmtechnisch mit mehr Aufwand verbunden, indizierte Zugriffe in der Regel nur in O(n) umzusetzen. Halten wir daher fest:

> Ein Baumcontainer ist nur für Objekte einsetzbar, für die eine Ordnungsrelation $<$ definiert ist (die Ordnungsrelation $=$ existiert immer, wie man sich leicht überzeugen kann).

Ein Element eines Baumcontainer ähnelt dem einer verketteten Liste: jedes Element besitzt (mindestens) zwei Zeiger, die auf Kindknoten verweisen, die mit ihren Zeigern jedoch nicht zurück-, sondern wiederum auf zwei weitere Elemente verweisen. Der Anker im Container ist dabei der höchste Knoten.

```
template <class T> struct Node {
    T object;
    Node<T>* left_child, *right_child;
};
```

[15] Nochmal zum Verständnis: Suchen + Einfügen = O(log(n)) + O(n) = O(n), da die größte Ordnung letztendlich den Aufwand bestimmt.

Ausgehend vom Anker (oder der Wurzel) besitzt ein Baum somit Ebenen, wobei sich die Anzahl der Objekte in einer in n Schritten vom Anker erreichbaren Ebene 2^n ist. Knoten, die keine weiteren Nachfolger besitzen, heißen Blätter, und bei optimalem Aufbau ist ein Blatt eines Baums mit n Objekten in $O(\log(n))$ Schritten erreichbar.

Dabei existieren zwei verschiedene Relationen zwischen Eltern-, also höheren, und Kindknoten:

- Im **Heap** ist der Elternknoten (genauer: das auf dem Knoten gespeicherte Objekt; wir werden uns im Weiteren aber der Sprachvereinfachung bedienen) immer größer (oder bei umgekehrter Sortierrichtung kleiner) als seine beiden Kinder. Zwischen den Kindern bestehen keine Relationen. Ein solcher Baum ist **teilsortiert**.
- Im **binären Baum** sind alle Objekte im linken Teilbaum stets kleiner als das Objekt im gerade betrachteten Knoten, alle Objekte im rechten Teilbaum größer. Ein binärer Baum ist daher **vollsortiert**.

Das folgende Beispiel zeigt, wir die Zahlen 1 .. 7 in solchen Speicherstrukturen verteilt werden. Zunächst ein Heap:

7						
4				6		
2		3		5		1

Anschließend ein binärer Baum:

4						
2				6		
1		3		5		7

Die Beispieldiagramme sind „balanciert“, was bedeutet, dass von der Wurzel ausgehend die Weglängen zu jedem beliebigen Endknoten maximal um eine Einheit differieren. Man kann sich auch leicht Bäume vorstellen, die nicht balanciert sind.

Aufgabe. Erstellen Sie nicht balancierte Bäume aus den Beispielen.

Es müssen jedoch nicht immer beide Kindzeiger gleichmäßig belegt sein, und ein unbalancierter Baum, in dem beispielsweise das linke Kind immer ein Nullzeiger ist, artet zu einer verketteten Listen aus, so dass die gewünschten Eigenschaften verloren gehen. Ein Großteil des Aufwands bei der Konstruktion von Algorithmen auf Bäumen entsteht also durch die Wahrung der Balance eines Baumes.

2.6.2 Heap (STL-Klasse priority_queue)

Möglicherweise haben Sie sich bei dem Zahlenbeispiel bereits die Frage gestellt, was man mit einem Heap eigentlich anfangen soll, denn zum Zugriff auf ein bestimmtes Element scheint es auf Anhieb keinen schnellen Algorithmus zu geben. Will man feststellen, ob der Wert 2 im Heap vorhanden ist, läuft der Suchalgorithmus mit O(n). Dieser Anschein trügt nicht, und das einzige Element, für das eine sichere Aussage getroffen werden kann, ist das erste, denn es ist gemäß Ordnungsrelation das größte.

Wenn es also dieses Element ist, um das sich alles beim Heap dreht, ist zu erwarten, dass effiziente Algorithmen existieren, die bei einem Hinzufügen eines neuen Elementes oder beim Entfernen des ersten dafür sorgen, dass anschließend wiederum das größte Element vorne steht. Sehen wir uns zunächst diese Algorithmen an.

Bei binären Bäumen ist die Wahrung einer Balance nicht ganz unkompliziert, bei Heaps aber glücklicherweise sehr einfach zu erreichen. Um auf einfache Art feststellen zu können, wie viele Ebenen belegt sind, verweist man in der Praxis das Knoten-Zeigermodell in das Reich der Theorie und legt einen Heap in Form einer linearen Liste (eines Vektors) an. Hier ergibt sich nämlich eine sehr einfache Eltern-Kind-Beziehung, wie man an einem Beispiel leicht nachrechnen kann:

(a) Der Wurzelknoten erhält die Indexnummer Eins.[16]
(b) Hat ein Elternknoten die Indexnummer K , so haben die Kinder die Indexnummern (2^*K) und (2^*K+1)
(c) In einem maximal balancierten Baum mit N Elementen werden die ersten N Speicherstellen dicht belegt.

Ein Heap ist also immer maximal balanciert, d.h. das Feld wird „dicht“ belegt und es existieren keine Löcher. Einen Heap mit Daten zu füllen wird unter diesen Bedingungen zu einer sehr einfachen, rekursiv zu erledigenden Arbeit:

(a) Füge das neue Element an die letzte belegte Position an. Sei N nun die neue Größe des möglicherweise an der letzten Position falsch belegten Heaps.
(b) Der Elternknoten besitzt den Index $E = N/2$, wobei die Division ganzzahlig unter Verwerfung des Restes erfolgt.
(c) Ist $a[E] < a[N]$, so tausche Eltern- und Kindknoten, setze $N = E$ und fahre fort bei (b), sofern $N > 1$ gilt.

Als Kodebeispiel sieht das folgendermaßen aus:

```
template <class T> class Heap{
private:
   vector<T> v;
```

[16] Abweichend vom C-Indexschema 0 .. (n-1) werden hier die Feldelemente mit dem Indexschema 1 .. n durchnumeriert. Bei einer Implementation müssen Sie das natürlich wieder rückübersetzen!

```
public:
   void push_heap(T const& t){
      v.push_back(t);
      for(int i=v.size();i>0;i/=2){
         if(v[i-1]>v[i/2-1])
            swap(v[i-1],v[i/2-1]);
         else
            break;
      ...
```

Das Einfügen eines neuen Elementes in einen Heap besitzt somit die Laufzeitordnung O(log(n)).

Aufgabe. Begründen Sie, weshalb gerade die Speicherstruktur eines Feldes diesen einfachen Algorithmus ermöglicht. Welche Probleme treten auf, wenn ein Heap in Form verketteter Zeiger realisiert wird?

Aufgabe. Sei ein Vektor mit unsortierten Elementen gegeben. Wie kann aus dem Vektor ein Heap erzeugt werden, welche Laufzeitordnung ergibt sich für diese Operation?

Wenn aus einem Heap ein Element entfernt werden soll, macht nur die Entfernung des ersten Elementes wirklich Sinn, denn von diesem wissen wir, dass es das größte Element im Heap ist. Um es aus einer Vektorstruktur effektiv entfernen zu können, tauschen wir es mit dem letzten Element des Heaps und entfernen es anschließend aus dem Feld, was in $O(1)$ abläuft. Hierdurch ist der Heap nun im ersten Element falsch belegt. Durch folgenden Algorithmus wird die Ordnung wiederhergestellt:

(a) Falls $E > (K_1,K_2)$ gilt, ist das Element korrekt einsortiert und der Algorithmus wird abgebrochen. Sonst wird
(b) E mit dem größeren der beiden Kinder getauscht, dessen Index nun das neue E bezeichnet.

Der Algorithmus wird bei (a) fortgesetzt, bis keine Kinder mehr vorhanden sind.

Die entsprechende Kodeergänzung zur `push`-Methode sähe dann folgendermaßen aus (vielleicht nicht gerade in elegantester Form, aber dafür vermutlich leicht nachvollziehbar):

```
void pop_heap(){
    swap(v[0],v[v.size()-1]);
    v.pop_back();
    int i=1,j;
    while(true){
        if(2*i < v.size())
            j=2*i-(int)(v[2*i]<v[2*i-1]);
        else if(2*i-1 < v.size())
            j=2*i-1;
```

```
        else
            return;
        if(v[i-1]>v[j])
            return;
        else{
            swap(v[i-1],v[j]);
            i=j+1;
        }
```

Auch dieser Algorithmus besitzt eine Laufzeitordnung von O(log(*n*)).

Aufgabe. Machen Sie sich klar, dass ein Heap ausschließlich für den Anwendungsfall, jeweils leicht auf das größte Element eines Containers zugreifen zu können, geeignet ist. Begründen Sie, weshalb sich die Suche nach einem bestimmten Objekt im Heap aufwandsmäßig kaum von der in einem unsortierten Container oder einer verketteten Liste unterscheidet.

Neben der LIFO– und der FIFO–Warteschlange stellt ein Heap eine Prioritäts-Warteschlange als weitere Form der Warteschlangen dar. Solche Warteschlangen werden in der Technik sehr häufig benötigt, allerdings ist damit auch schon weitgehend der Verwendungsbereich eines Heaps abgedeckt (auf eine Verwendung in Sortieralgorithmen zur Erzeugung eines sortieren Feldes kommen wir später zu sprechen). Dafür ist Heap-Sortierung mit äußerst geringem Aufwand herzustellen. In der STL ist die Warteschlange durch die Klasse `priority_queue` realisiert. Sie besitzt die gleiche Schnittstelle wie `stack`, hat aber als zusätzlichen Vorlagenparameter eine Struktur `less<..>`, die die Vergleichsrelation definiert:

```
template<class T,
    class Cont = vector<T>,
    class Pred = less<Cont::value_type> >
    class priority_queue {
```

Im Normalfall ist `less<..>` durch einen einfachen `operator<(..)`–Vergleich implementiert.

Die Art der Sortierung hat die weitere Konsequenz, dass einmal in der Liste befindliche Objekte ihre relative Position zueinander nicht mehr ändern können. Enthält eine Warteschlange beispielsweise Objekte mit Bewertungen <50, neu hinzukommende Objekte im Schnitt die Bewertung ∼90 und entspricht die Zugangsrate der Abgangsrate, so haben die Objekte mit kleinen Bewertungen keine Chance, die Warteschlange zu verlassen. Umgehen ließe sich dies, indem die Bewertung jedes bereits vorhandenen Elementes beim Hinzufügen eines weiteren vergrößert wird; allerdings muss die Erhöhung für alle Elemente gleich sein, da die relative Position zueinander in den meisten Zweigen des Baumes nicht mehr angepasst wird, und zusätzlich würde durch einen solchen Schritt die Effizienz sinken, da jedes Element angefasst werden muss, was zur Ordnung O(*n*).

2.6.3 Binärer (Rot-Schwarz)-Baum

Bei binären (*und anderen*) Bäumen liegt immer eine Vollsortierung vor (*siehe oben*), was die Einhaltung einer Balance sehr viel schwieriger macht als bei einem Heap. Entsprechend existiert eine Vielzahl unterschiedlicher Strategien, dies zu erreichen. Führen wir uns zunächst noch einmal die Ziele vor Augen: das Suchen, Einfügen und Löschen in einem Baum soll in der Regel mit der Ordnung $O(\log(n))$ erfolgen. Das Umsortieren von Elementen wird aufwändiger als beim Heap werden, und um durch solche Vorgänge nicht die Effizienz zu gefährden, legen wir uns bei der hier diskutierten Strategie auf folgende Randbedingungen fest:

(a) Die Speicherstruktur ist kein Feld, sondern besteht aus verketteten Zeigerelementen gemäß theoretischer Einführung.
(b) Die Balance muss nicht optimal (d.h. Weglängenunterschiede maximal Eins) sein, sondern darf um einen definierten Grad davon abweichen.

Eine Strategie, die dies gewährleistet, ist die des Rot-Schwarz-Baumes.

2.6.3.1 Konstruktionsprinzip

Die Grundstruktur für einen rot-schwarz Baum erhalten wir durch einige Modifikationen der Basisstruktur:

```
template <typename T> class RSTree {
public:
   RSTree():root(0){}
   ~RSTree(){ if(root) delete root;}
private:
   enum Color {black, red};
   struct Node {
      T obj;
      Node *less, *greater, *parent;
      Color color;
      Node(T const& t, Node* p): obj(t), color(red),
                        less(0),greater(0), parent(p){}
      ~Node(){
         if(less) delete less;
         if(greater) delete greater;
      }
   } *root;
};//end class
```

Der gesamte Baum wird am Ankerattribut `root` aufgehängt. Jeder Knoten erhält als zusätzliche Eigenschaft eine „Farbe“, die aber nur die Werte „Rot“ oder „Schwarz“

annehmen kann, und damit wir uns im Baum bei Umsortierungen etwas eleganter bewegen können, definieren wir außer den Zeigen auf die Kinder auch einen Zeiger auf den Elternknoten. Bevor wir uns mit den Hintergründen auseinandersetzen, zunächst noch eine Übung:

> **Aufgabe.** Ignorieren Sie zunächst die Farbe und sämtliche Überlegungen an eine Balancierung und implementieren Sie Einfügeoperationen für einen binären Baum, wobei der Baum durchaus zu einer verketteten Liste ausarten darf. Wenn das zu einfach gewesen ist, implementieren Sie auch die Löschoperation, die etwas komplizierter ist, da hier auch Knoten in der Mitte gelöscht werden.
> Beides werden wir später als Grundalgorithmen benötigen und die Balancierungsalgorithmen darauf aufsetzen.

Nun zur Theorie: der Baum soll folgende Regeln erfüllen:

(1) Alle Null-Pointer sind per Definition Schwarz (meist erhält auch die Wurzel die Eigenschaft Schwarz).
(2) Besitzt ein roter Knoten einen Vater, so ist dieser Schwarz.

    Es dürfen also nicht zwei rote Knoten direkt miteinander verkettet sein, wohl aber beliebig viele schwarze.
(3) Alle Wege von einem beliebigen Knoten zu einem Null-Pointer besitzen die gleiche Anzahl von schwarzen Knoten. Dies wird als Schwarzhöhe des Baumes bezeichnet und stellt die Balancebedingung dar.

Nach diesen Regeln sind Wege, die nur aus schwarzen Knoten, und solche, die alternierend aus roten und schwarzen gebildet werden, möglich. Die Balancierung ist damit nicht optimal, aber die Absolutlängen zwischen zwei beliebigen von einem Knoten ausgehenden Wegen können sich allenfalls um den Faktor Zwei unterscheiden, wenn auf einem Weg nur schwarze Knoten aufeinander folgen, auf einem anderen schwarze und rote alternieren. Dies ist der Kompromiss, den wir eingehen, um den Aufwand der Neusortierung eines Baumes beim Einfügen und Löschen zu begrenzen.

2.6.3.2 Einfügen eines Knotens

Zum Einfügen eines Knotens (*was zunächst nach der gleichen Methode geschieht, die Sie in der Basisaufgabe gelöst haben*), ergänzen wir die Regeln um

(4) Jeder neue Knoten ist rot.

Der erste Schritt sieht (etwas verkürzt) somit folgendermaßen aus.

```
template <typename T>
bool RSTree<T>::insert(T const& obj){
    if(root==0){
```

```
        root=new Node(obj,0);
        return true;
    }//endif

    Node *n=root;
    while(true){
        if(obj==n->obj) return false;
        if(obj<n->obj){
            if(n->less){
                n=n->less;
            }else{
                n->less=new Node(obj,n);
                balance_insert(n->less);
                return true;
            }//endif
        }else{
            ...
        }//endif
    }//endwhile
}//end function
```

Die Nachsortierung übernimmt die Methode `balance_insert(..)`. Da jeder neue Knoten die Farbe Rot erhält (und per Definition schwarze Kinder besitzt) muss diese dann tätig werden, wenn der Elternknoten ebenfalls rot ist. Hierzu untersucht die Methode folgende Knoten:

- den Problemknoten, der in der ersten Prüfrunde der neu eingefügte Knoten ist, an dessen Stelle aber infolge rekursiver Ordnungsvorgänge andere Knoten treten können,
- dessen Elternknoten des Problemknotens,
- den Großvater, also den Elternknoten des Elternknotens, und
- den Onkel, d.h. den anderen Kindknoten des Großvaters.

Die Untersuchung soll nur rekursiv in Richtung der Wurzel geführt werden, sich also nicht in anderen Zweigen des Baumes in Richtung der Blätter erstrecken, da nur so die Forderung, dass die Vorgänge insgesamt den Aufwand $O(\log(n))$ nicht überschreiten, gewährleistet werden. Per Voraussetzung sind immer sämtliche Kindteilbäume, die von den vier untersuchten Knoten ausgehen, im Sinne der Rot-Schwarzbaum-Regeln korrekt aufgebaut, und diese Ordnung darf durch den Sortiervorgang auf keinen Fall zerstört werden. Der Algorithmus muss im Problemfall nun die Regel, dass keine zwei roten Knoten aufeinander folgend dürfen, wiederherstellen, und zwar so, dass ausgehend vom Großvater

- auf beiden Seiten gleiche Schwarztiefe besteht,
- jeder Teilbaum korrekt aufgebaut ist,
- sich die Schwarztiefe im Teilbaum des Großvaters insgesamt nicht geändert hat.

Die letzte Regel mag im ersten Augenblick etwas befremden, macht aber Sinn, wenn man bedenkt, dass sich durch das Einfügen eines neuen Knotens aufgrund der roten Farbe ebenfalls nichts an der Schwarztiefe ändert. Der einzige Grund für eine Neusortierung ist der Regelverstoß gegen Regel (2), und wenn die Methode rekursiv arbeiten soll, darf sich daran zunächst nichts ändern. Natürlich muss sich beim fortlaufenden Einfügen von Elementen die Schwarztiefe des Baumes insgesamt zwangsweise ändern, jedoch ist diese Änderung nur am Ende der Rekursion, also an der Wurzel, möglich.

Um den Teilbaum zu reparieren, kann es notwendig werden, den Großvater, der zwangsläufig schwarz ist (der Problemfall tritt ja nur bei einem roten Vater auf, und zuvor war der Baum ja regelkonform), nach Rot umzufärben, womit dieser ein neuer Problemknoten wird und die Rekursion mit ihm fortgesetzt werden kann. Ist das nicht mehr möglich, weil der Elternknoten bereits der (rote) Wurzelknoten ist, bleibt nur, die Wurzel nach Schwarz umzufärben, was immer möglich ist, das Problem behebt und gleichzeitig die Schwarztiefe des Baumes erhöht. Damit haben wir schon die Abbruchbedingung ermittelt. Die Auswertung der Farbe eines Knotens wird dabei so konstruiert, dass auch die Nullknoten eine Farbe besitzen. Der folgende Algorithmenteil bricht ab, wenn kein Konflikt vorliegt oder der Elternknoten die Wurzel ist:

```
template <typename T>
typename Color RSTree<T>::node_color(Node* n){
    if(n) return n->color;
    else  return black;
}
template <typename T>
void RSTree<T>::balance_insert(Node* n){
    if(node_color(n->parent) == black) return;
    if(n->parent->parent==0){
        n->parent->color=black;
        return;
    }
    ...
```

Bei weiter bestehendem Konflikt können nun zwei Fälle eintreten: der Onkel kann ebenfalls Rot oder Schwarz sein.

> **Aufgabe.** Implementieren Sie eine Methode `get _oncle(..)`, die den Onkel ermittelt und den Zeiger darauf zurückgibt (der ein Nullzeiger sein kann). Hinweis: Sie müssen prüfen, ob der Elternknoten den `greater`- oder `less`-Zweig darstellt, falls ein Großvater existiert.

Einfach zu lösen ist der Fall eines (ebenfalls) roten Onkels. Man weist dem Eltern- und dem Onkelknoten eine schwarze und dem Großvater eine rote Farbe zu:

```
Node* o = get_oncle(n);
if(node_color(o)==red){
   set_black(n->parent);
   set_black(o);
   set_red(n->parent->parent);
   balance_insert(n->parent->parent);
   return;
}
```

Aufgabe. Implementieren Sie die beiden Methoden `set _black(..)` und `set _red(..)`. Diese müssen so konstruiert werden, dass sie auch bei Übergabe eines Nullzeigers keinen Unfug machen (siehe vorhergehende Aufgabe).

Überzeugen Sie sich davon, dass dieser Fall nur eintreten kann, wenn der Großvater existiert (die Zeigerverkettungen somit gültig sind und keine Laufzeitfehler verursachen können), dass durch diesen Farbentausch die Schwarztiefe im gesamten Teilbaum des Großvaters korrekt und unverändert ist und der Teilbaum nun regelgerecht konstruiert ist. Nach dem Tausch kann nun allerdings ein Konflikt zwischen dem Großvater und dem Urgroßvater auftreten, so dass der Algorithmus rekursiv fortgesetzt werden muss.

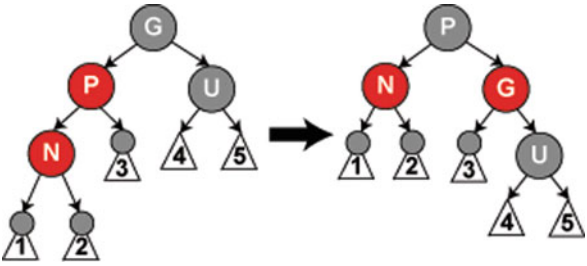
Der verbleibende Fall eines schwarzen Onkels ist komplizierter. Der Onkel kann nicht nach Rot umgefärbt werden, weil dadurch möglicherweise Konflikte weiter unten in seinem Teilbaum entstehen, was wir oben als Regel ausgeschlossen haben. Vater und Großvater können ebenfalls nicht einfach die Farben tauschen, weil dadurch die Schwarztiefe geändert wird (überzeugen Sie sich durch ein Beispiel davon). Wenn Farben geändert werden, was ja notwendig ist, kann dies nur bei einem gleichzeitigen Rollentausch der vier Knoten erfolgen, wobei aber wiederum deren Kindknoten nach Möglichkeit ihren Ort behalten müssen, um die Baumstruktur zu wahren. Bei einem Rollentausch ist außerdem zu berücksichtigen, dass

(a) Der Problemknoten rechtes oder linkes Kind des Vaters und
(b) der Vater wiederum rechtes oder linkes Kind des Großvaters

sein kann, was auf insgesamt vier unterschiedliche Konstellationen führt. Wir lösen des Problem einmal exemplarisch für den Fall, dass die Knoten jeweil linke, als `less`-Knoten ihrer Eltern sind. Die Lösung besteht in einer sogenannten Rotation der Knoten, wobei

- der Problemknoten an die Position des Elternknotens rückt und seine rote Farbe behält,
- der Elternknoten an die Stelle des Großvaters rückt und dabei Schwarz wird,
- der Großvater die Position des Onkels übernimmt und zu Rot wechselt,
- der Onkel nicht betroffen ist, aber formal an die Position des rechten Kindes des neuen Onkels rückt.

Die Kinder des Problemknotens und des Onkels sind nicht betroffen und behalten ihre Bindungen bei, lediglich das `greater`-Kind des Elternknotens muss verschoben werden, da der Elternknoten ja den Problemknoten und den alten Großvaterknoten als Zeigerverweise übernimmt. Da dieser Zweig sämtliche Objekte enthält, die größer als der alte Elternknoten, aber kleiner als der alte Großvater sind, rückt der Zweig in die `less`-Position des neuen Onkels, die freigeworden ist. Die folgende Grafik verdeutlicht die Operation[17]



Aufgabe. Überzeugen Sie sich durch ein Beispiel davon, dass dieser Rollentausch tatsächlich das Problem löst, die Schwarztiefe insgesamt erhalten bleibt und sämtliche Relationen im Baum erfüllt sind.

Der Code besteht aus einem Umhängen der Zeiger, wobei man ein wenig auf die Reihenfolge achten muss, um keine Referenz zu verlieren. Wir beginnen mit der Änderung des Urgroßvaters (sofern existent), der ja anstelle des Großvaters nun einen Zeiger auf den Vater erhalten muss. Hierbei ist wieder zu berücksichtigen, ob es sich um das rechte oder linke Kind handelt. Nach der Änderung der Farben wird der `greater`-Zweig des Vaters mit dem `less`-Zweig des Großvaters vertauscht, was nach unten bereits korrekte Relationen liefert. Abschließend werden noch die Elternrelationen angepasst.

```
if(left_child(n)+left_child(n->parent)){
   if(n->parent->parent->parent){
      if(left_child(n->parent->parent)
         n->parent->parent->parent->less=n->parent;
      else
         n->parent->parent->parent->greater=n->parent;
   }
   n->parent->color=black;
   n->parent->parent->color=red;
```

[17] Diese Grafik und weitere sind aus Wikipedia als öffentliche und beliebig frei verwendbare Grafiken übernommen.

```
    swap(n->parent->greater,n->parent->parent->less);
    n->parent->parent=n->parent->parent->parent;
    n->parent->parent=n->parent;
    return;
}
```

Da bei dieser Aktion die Farbe des alten Großvaters mit der des neuen übereinstimmt und Schwarz ist, ist mit dem Tausch die Reparatur bereits beendet.

> **Aufgabe.** Implementieren Sie ebenfalls die anderen Fälle. Sie können dies individuell machen oder versuchen, einen Fall zunächst durch eine andere Rotation auf einen bereits gelösten zurückzuführen und dann diesen anzuwenden.
>
> Ist beispielsweise der Problemknoten rechtes Kind seines Vaters, kann man ihn zum Vaterknoten und den Vaterknoten zum neuen linken Problemknoten machen, ohne die Farben zu ändern. Zusätzlich ist das linke Kind des Problemknotens auf die rechte Position des Vaterknotens umzuhängen, womit der bereits untersuchte Fall eingestellt ist.
>
> Solche Rückführungen werden in manchen Literaturstellen bevorzugt, bezüglich des Gesamtaufwands ist es aber letztendlich egal, welche Lösung man implementiert.

Wenn Sie alles korrekt implementiert haben, können Sie nun mit der Methode

```
void print_out(Node* n, string s){
    char const co[2][10] = {"rot","schwarz"};
    if(n){
        print_out(n->greater,s+"    ");
        cout << s << n->obj << " " << co[n->color] << endl;
        print_out(n->less,s+"    ");
    }
}
```

überprüfen, ob Ihr Baum bei Einfügeoperationen korrekt konstruiert wird.

2.6.3.3 Löschen von Knoten

Das Löschen eines Knotens erscheint dem Betrachter zunächst kompliziert, wenn im allgemeinen Fall ein Knoten mitten im Baum betroffen ist. Durch eine einfache Operation lässt sich dieser Fall jedoch auf das Löschen eines Blattes oder zumindest eines Knotens, der nur ein Kind besitzt, zurückführen. Dazu wird entweder das Objekt des größten Knotens des `less`-Zweiges oder der des kleinsten Knotens des `greater`-Zweiges auf den zu löschenden Knoten kopiert. Anschließend ist der Knoten, von dem kopiert wurde, zu löschen.

```
Node* c=n->greater;
while(c->less != 0) c=c->less;
n->obj=c->obj;
n=c;
```

> **Aufgabe.** Das Kopieren ist die einfachste und in der Regel auch die effektivste Operation. Lediglich bei sehr aufwändigen Objekten könnte ein Umhängen der Knoten günstiger sein. Untersuchen Sie, wie die Knoten selbst ausgetauscht werden können (implementieren Sie das aber nicht, sondern belassen Sie es bei der Kopieroperation).

Der zu löschende Knoten `n` kann nun Rot oder Schwarz sein. Beginnen wir mit dem Fall Rot. Sein Vaterknoten ist damit auf jeden Fall Schwarz und beide Kindknoten sind Null (*da sonst die Schwarztiefe nicht konstant sein könnte*). In diesem Fall kann er entfernt werden, denn die Schwarztiefe bleibt konstant.

```
if(n->color==red){
   n->parent->less=0;
   delete n;
   return;
}
```

Im Fall Schwarz können ebenfalls wieder zwei Fälle auftreten: beide Kinder sind Null oder das `greater`-Kind ist vorhanden und Rot (und besitzt seinerseits keine weiteren Kinder). Machen Sie sich an einem Beispiel klar, dass an dieser Stelle andere Möglichkeiten nicht vorhanden sind, wenn es sich um einen korrekt konstruierten Baum handelt.

Behandeln wir zunächst den Fall des roten Kindes. Nach Kopieren von dessem Wert auf `n` kann der `less`-Knoten gelöscht werden und wir sind wiederum fertig.

```
if(n->black && n->less!=0){
    n->obj=n->greater->obj;
    delete n->greater;
    n->greater=0;
    return;
}
```

Im verbleibendem Fall ist `n` ist Schwarz und besitzt keinen `greater`-Zeiger. Wir entfernen den Knoten, müssen jetzt aber weiteren Maßnahmen treffen, da die Schwarztiefe vermindert worden ist. Der Problemknoten ist der Vaterknoten des gelöschten Knotens, der nun Zweige mit unterschiedlichen Schwarztiefen aufweist. Da die Beseitigung des Konflikts wieder zu rekursiven Maßnahmen führen kann, bereiten wir diese etwas ausführlicher vor.

```
Node* parent, *brother;
parent=n->parent;
if(parent==0){
   delete n;
```

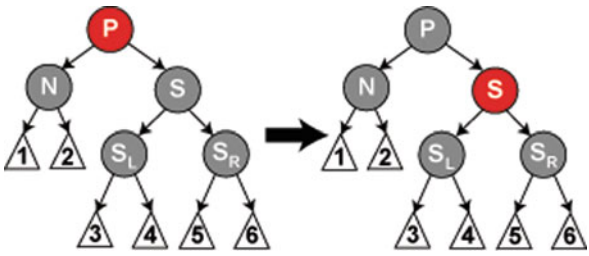
```
    root=0;
    return;
}
brother=parent->greater;
delete parent->less;
n=parent->less=0;
repair(n,parent,brother);
```

Der erste Teil des Algorithmus berücksichtigt das Löschen des letzten Elementes des Baumes, also der Wurzel. Im allgemeinen Fall werden berücksichtigt

- der den Konflikt auslösende Knoten `n`, zu Beginn der Rekursion nach dem Löschen eines Knotens also ein Nullknoten, im Laufe der Rekursion ein tatsächlich existierender Knoten,
- dessen Elternknoten,
- der Bruderknoten des auslösenden Knotens, der auf jeden Fall vorhanden ist,
- die Kinder des Bruderknotens, die im ersten Rekursionsschritt auch schwarze Nullzeiger sein können (oder Rot sind).

Zu bemerken ist, dass wir erst mit der Methode `repair(..)` in die Terminologie „Konfliktknoten“ einsteigen und zuvor schon einiges an Arbeit und Fällen erledigt wurde. Insbesondere ist mit Einstieg in die Methode bereits der zu entfernende Knoten gelöscht worden, und die weiteren Operationen bestehen nur noch aus Umsortiervorgängen innerhalb des Baumes. Die Rolle des Konfliktknotens im Rekursionsfall übernimmt jeweils der aktuelle Elternknoten nach Ausführen des Teilalgorithmus, so dass die Rekursion nach O(log(n)) Schritten abgeschlossen ist. Die Strategie muss nun wieder sein, in den beiden vom Elternknoten ausgehenden Teilbäumen die gleiche Schwarztiefe herzustellen, wobei maximal auf die Enkel zurückgegriffen werden darf. Hierzu stehen uns als Werkzeuge wieder geschickter Umfärben oder Umgruppieren zur Verfügung.

Beginnen wir mit dem Fall, dass der Bruder (S) ebenfalls Schwarz und der Vater (P) Rot ist. Der Bruder besitze außerdem zwei schwarze Kinder (S_L und S_R). Wir tauschen in diesem Fall nur die Farben von Vater und Bruder aus.

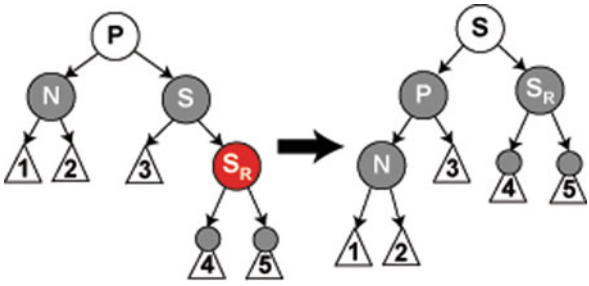


Durch diese Operation ist die Schwarzhöhe des Konfliktknotens (N) um eine Einheit erhöht worden, während alle anderen Knoten immer noch dieselbe Höhe besitzen. Der Baum ist also wieder ausbalanciert, und wir sind fertig.

```
void repair(Node* n, Node* p, Node* b){
   ...
   if(node_color(p)==red &&
      node_color(b->less)==black &&
      node_color(b->greater)==black){
      p->color=black;
      b->color=red;
     return;
 }
```

> **Aufgabe.** Wir untersuchen die Reparaturmechanismen jeweils für die Ausgangssituation, d.h. der Konfliktknoten (N) ist immer der linke Knoten des Vaters (und ein Nullknoten). Untersuchen sie für die Kodeteile jeweils, ob Fallunterscheidungen für andere Konstellationen notwendig sind, und notieren Sie dies im Code. Untersuchen Sie im Rekursionsfall, ob dadurch andere Konstellationen erzeugt werden können, und implementieren Sie die jeweils fehlenden Fälle.

Das funktioniert nicht mehr, wenn der Bruderknoten keine schwarzen Kinder besitzt. Wir betrachten den Fall, dass (mindestens) das rechte Bruderkind Rot ist. Wir können nun wieder eine Rotation durchführen, die der beim Einfügen beschriebenen gleicht:



Auch hier lässt sich durch Abzählen leicht verifizieren, dass sich die Höhe für (N) und alle seine Nachfolger um eine Einheit erhöht hat, womit ein Ausgleich für den gelöschten Knoten geschaffen ist, während sich für alle anderen Knoten nichts geändert hat, d.h. auch hier sind wir bereits fertig. Die Farbe des Vaterknotens hat keinen Einfluss auf das Ergebnis, muss aber konstant bleiben, wie man leicht verifiziert.

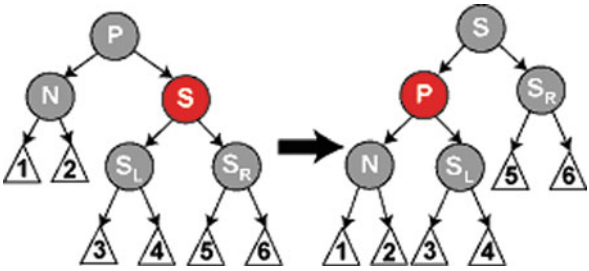
Aufgabe. Implementieren Sie den Code hierfür. Da wir das nun bereits mehrfach geübt haben, überlasse ich das komplett Ihnen und gebe nur einige Hilfestellungen.

Beachten Sie, dass anstelle des rechten auch das linke Kind der rote Übertäter sein kann. Sie können dies wie beim Einfügen durch Fallunterscheidungen oder durch Zwischenrotationen im Teilbaum des Bruders und Rückführen auf den beschriebenen Fall lösen. Beachten Sie auch die anderen Möglichkeiten gemäß der zuvor gestellten Aufgabe.

Beachten Sie auch, dass auch die Zeiger der Großvaters umgehängt werden müssen, sofern der Elternknoten nicht bereits die Wurzel ist.

Bei den Operationen können an verschiedenen Stellen Nullzeiger auftreten, die das Leben zusätzlich schwer machen, wenn formal auf deren Kinder zugegriffen werden muss. Empfehlenswert sind an dieser Stelle spezielle Methoden, die zwischen Nullzeiger und Objekten sowie zwischen linken und rechten Kindern unterscheiden können, da sie die Tiefe der Fallunterscheidungen im Code begrenzen.[18]

Besitzt der Bruderknoten (S) die Farbe Rot, so hat er notwendigerweise schwarze Kinder und einen schwarzen Vater. Diesen Fall können wir durch eine Rotation in einen der bereits untersuchten überführen:



Der Vater (P) bleibt Vater des Konfliktknotens, der Bruder (S) wird zum Großvater, während einer der Neffen (S_L) zum neuen Bruder, der andere zum neuen Onkel (S_R) wird.Wie man unschwer erkennt , hat sich nun für keinen der Knoten etwas an der Schwarzhöhe geändert und der Konfliktknoten (N) ist weiterhin der Konfliktknoten. Er besitzt jedoch nun einen neuen schwarzen Bruder und einen roten Vater, kann also mit einer der beiden zuvor ermittelten Operationen weiterbearbeitet werden.

Aufgabe. Implementieren Sie diesen Fall ebenfalls. Rein formal könnte S_L auch ein Nullknoten sein. Kann dieser Fall eintreten, und falls ja, wie ist darauf zu reagieren?

Bringen Sie nun die verschiedenen Fälle in eine sinnvolle Reihenfolge.

---

[18] In der Literatur werden von manchen Algorithmen anstelle der Nullzeiger NIL-Objekte verwendet, d.h. man stößt bei der Abarbeitung der Algorithmen nirgends auf echte Nullzeiger. Man handelt sich damit jedoch an anderen Stellen Ärger ein, so dass von dieser Lösung abgeraten sei.

Bis jetzt sind wir noch nicht auf eine Rekursion gestoßen. Da so etwas aber zwangsweise auch geschehen muss, kann dies nur im allerletzten noch nicht berücksichtigten Fall, dass sämtliche Knoten schwarz sind, erfolgen. Verifizieren Sie, dass dieser Fall primär dadurch eintreten kann, dass ein schwarzes Blatt ohne Kinder und einem schwarzen Elternknoten sowie einem schwarzem Onkel (ebenfalls ohne Kinder) gelöscht wird. Dieser Fall ist lokal einfach dadurch zu lösen, dass der Bruderknoten Rot gefärbt wird.

Hierdurch ist zwar nun der komplette, beim Elternknoten beginnende Teilbaum wieder korrekt, jedoch ist die Schwarztiefe gegenüber dem Ausgangszustand um eine Einheit vermindert. Sofern der Elternknoten nicht die Wurzel des Baumes ist, wird er damit neuer Problemknoten, und die Rekursion ist mit seinem Eltern- und Bruderknoten fortzuführen. Auch hierbei kann wieder der Fall auftreten, dass N, P, S, S_L und S_R schwarz sind, der in gleicher Weise wie der Startfall gelöst wird und die Rekursion fortsetzt.

2.6.3.4 Iteratoren

In binären Bäumen lassen sich sequentielle Operatoren implementieren, die ein Durchlaufen der Elemente vom größten zum kleinsten erlauben. Ein Iteratorobjekt enthält einen Zeiger auf einen aktiven Knoten, dessen Inhalt mittels des Dereferenzierungsoperators (`operator*`) abgerufen werden kann. Beim Wechsel zum nächsten aktiven Knoten müssen allerdings einige Feinheiten beachtet werden. Die Schnittstelle der Iteratorklasse erhält folgendes Aussehen:[19]

```
template <class T> class RSTree;

template <class T> class Iterator {
public:
   ...
   Iterator<T>& operator=(Iterator<T> const&);
   Iterator<T>  operator++();
   T const& operator*() const;
   bool operator==(Iterator<T> const&) const);
protected:
   typename RSTree<T>::Node* node;
   int status;
   template <class S> friend class RSTree;
};
```

[19] Die Iteratorklasse wird vor der Containerklasse implementiert. Die Implementation erfordert zusätzlich eine friend-Deklatiotion der Iteratorklasse in der Containerklasse, um auf Node zugreifen zu können.

Aufgabe. Wenn Sie sich die Schnittstelle ansehen, stellen Sie fest, dass lediglich eine konstante Referenz auf den Knoteninhalt abgeliefert wird, dieser also nicht verändert werden kann. Warum?

Beginnen wir mit der Initialisierung einer Iteratorvariablen durch die `begin()`-Methode des Baumobjektes. Sinnvollerweise erzeugt diese Methode ein Interatorobjekt, dass auf das kleinste Element im Baum verweist.

```
Iterator<T> RSTree<T>::begin(){
  Iterator<T>it;
  it.node=root;
  it.status=1;
  while(it.node->less!=0)
    it.node=it.node->less;
  }//endif
  return it;
}
```

Bei Aufruf des Inkrementierungsoperators können nun mehrere Zustände auftreten:

(a) Der aktive Knoten ist `less`-Knoten seines Elternknotens und besitzt keine Nachfolger. Der Elternknoten wird nun aktiver Nachfolger.
(b) Der aktive Knoten besitzt einen `greater`-Knoten. Nachfolger wird der kleinste Knoten des `greater`-Zweiges.
(c) Der aktive Knoten ist `greater`-Knoten seines Elternknotens. Nachfolger wird der Elternknoten, der in einer rekursiven Kette den aktiven Knoten in einem `less`-Zweig aufweist.
(d) Nach Rekursion (c) ist kein unbearbeiteter Elternknoten mehr zu finden, und die Rekursion endet im Wurzelknoten.

Dies lässt sich relativ einfach in Code umwandeln.[20] Zustand (d) kennzeichnen wir mit `status=0` und machen nichts, weil der Enditerator erreicht ist, der auch so von der `end()`-Methode des Baumobjektes eingerichtet wird.

```
Iterator<T> operator++(int){
   if(status==0) return *this;
```

Im zweiten Schritt prüfen wir, ob der Knoten einen `greater`-Nachfolger besitzt. Falls ja, gehen wir zum kleinsten Element dieses Zweiges.

```
 if(node->greater){
   node=node->greater;
   while(node->less) node=node->less;
   return *this;
 }
```

[20] Der Einfachheit halber implementieren wir den Präfixoperator.

Im dritten Schritt prüfen wir, ob es sich um den `less`-Eintrag des Elternknotens handelt. In diesem Fall gehen wir einen Schritt zurück.

```
if(node==node->parent->less){
  node=node->parent;
  return *this;
}
```

Andernfalls gehen wir bis zur Erfüllung von Bedingung (c) oder (d) zurück.

```
  while(node->parent){
    if(node==node->parent->less){
      node=node->parent;
      return *this;
    }
    node=node->parent;
  }
  status=0;
  return *this;
}//end function
```

Aufgabe. Die Darstellung ist etwas vereinfacht. Implementieren Sie die Iteratorklasse und binden Sie sie in die Baumklasse ein.

2.6.4 STL-Klassen set und map/Hashsortierung

2.6.4.1 Objekte, Schlüssel und Daten

Auf der Basis des Rot-Schwarz-Baums stellt die STL ebenfalls eine Reihe von Containern zur Verfügung, so dass eine Optimierung der Übungsimplementation nicht notwendig ist. Vor den Details der STL-Container sind jedoch noch einige grundsätzliche Bemerkungen notwendig.

Sortierte Listen und Bäume setzen die Existenz eines Ordnungskriteriums $<$ auf der Objektmenge voraus. Bislang haben wir unterstellt, dass das Ordnungskriterium auf den Objekten selbst definiert ist. Das muss jedoch nicht so sein, und wir formulieren allgemeiner, dass für die Verwendung sortierter Listen oder Bäume ein Schlüssel definiert sein muss, für den eine Ordnungsrelation $<$ existiert. Das Verhältnis zwischen Schlüssel und Objekt kann nun folgendermaßen aussehen:

(a) Der Schlüssel ist mit dem Objekt identisch oder implizit im Objekt enthalten (*das war der bisherige Standardfall, den wir als behandelt betrachten können*) oder
(b) der Schlüssel ist eine vom eigentlichen Objekt unterschiedene Größe.

Diese Unterscheidung ist insofern wichtig, als Schlüssel nicht geändert werden dürfen, da ansonsten die Baumstruktur verloren geht. Container des Typs (a) können daher nur dahingehend überprüft werden, ob ein Objekt vorhanden ist, aber keine weiteren Daten liefern, da mit dem Schlüssel bereits alles erledigt ist, während Container des Typs (b) durchaus veränderbare Daten enthalten können.

Die Container können beim Füllen mit Daten

- gleiche Objekte ausschließen, was bedeutet, dass zwei Objekte als gleich anzusehen sind, wenn sie gleiche Schlüssel aufweisen. Das gilt im Fall (b) auch dann, wenn eine direkte Anwendung von `operator==` auf zwei Objekte die Antwort `false` liefert, die Objekte sich also in irgendwelchen, für die Schlüsseldefinition unerheblichen Attributen unterscheiden. Der Versuch, ein von den im Container vorhandenen unterschiedliches Objekt abzulegen, wird mit der Begründung, dieses Objekt sei bereits gespeichert, zurückgewiesen.

Andere Containerkonstruktionen können

- gleiche Objekte zulassen, d.h. zwei selbst Objekte, die sich bei Anwendung von `operator==` auf die Objekte selbst nicht unterscheiden, mit gleichen Schlüsseln trotzdem als zwei Datensätze übernommen werden.

In unserer Übungsimplementation des Rot-Schwarz-Baumes haben wir den Containertyp (a) realisiert. Zur Ablage eines Objektes mit einem getrennten Schlüssel, also zur Umsetzung eines Containers des Typs (b), werden beide Größen in der Datenstruktur

```
template <typename U, typename T> struct pair {
   U key;
   T object;
   bool operator<(pair<U,T> const& p)const{
      return key<p.key;
   }//end function
};//end struct
```

vereinigt und diese in einer sortierten Liste oder einem Baum abgelegt. Formal können wir unseren Übungsbaum damit auch zur Verwaltung solcher Daten nutzen, wobei allerdings einige Anpassungen im Iteratorbereich notwendig werden (siehe unten).

2.6.4.2 Objekte ohne Ordnungsrelationen

Auch wenn kein logisch irgendwie begründbarer Schlüssel konstruierbar ist, kann es anwendungstechnisch erwünscht sein, „sortierte“ Listen zu erstellen, die das Auffinden eines Objektes in O(log(n)) Schritten ermöglichen. Beispielsweise ist es schon etwas kompliziert, eine Rangordnung auf Bildschirmfarben zu definieren, die durch

```
unsigned char pixel[3];
```

angegeben werden, oder komplexe Zahlen zu sortieren.[21] Um das dennoch zu erreichen, wird vom Objekt bzw. dem Objektteil, der sich als Charakteristikum eignet, ein Hashwert gebildet, der als Schlüssel einer sortierten Liste dient.

Hashfunktionen sind verlustbehaftete schnelle Datenkompressionsfunktionen, die aus beliebig großen Eingabedatenströmen Ausgaben fester Länge produzieren. Bei geeignetem Design sind Hashfunktionen kollisionsarm, das heißt wird eine Nachricht auf eine Zahl $k < N = Max(Hash(\ldots))$ abgebildet, so liegt die Wahrscheinlichkeit zweier gleicher Verschlüsselungen für verschiedene Nachrichten bei

$$w\,(Hash(m_1) = Hash(m_2)) \approx 1/N$$

Aufgabe. Eine recht einfache Hashfunktion lässt sich mit einer Primzahl und Modulrechnung implementieren. Nehmen Sie beispielsweise die Zahl $n = 65.521$ und verschlüsseln Sie Textzeilen oder Binärdaten auf folgende Art: Der Startwert für den Algorithmus ist Eins. Schieben Sie je Schritt 32 Bit auf eine `int` –Variable und berechnen Sie iterativ

```
hash = ((li % n) * hash ) % n;
```

bis alle Bits des eingelesenen Strings oder Puffers bearbeitet sind. Probieren Sie anhand von Textdateien oder Programmdateien aus, wie oft Kollisionen auftreten (*die Länge der Puffer ist zu begrenzen, zum Beispiel auf 128 Bytes*).

Bei Verwendung dieser Ersatzschlüssel ist zu berücksichtigen, dass in solchen Containern nur sinnvoll geprüft werden kann, ob ein Objekt vorhanden ist oder nicht. In einem Container mit Zahlen oder Textstrings als Schlüssel kann man beispielsweise untersuchen, welche Objekte in der Nähe des gesuchten Schlüssels liegen, falls dieser nicht vorhanden ist. Bei Hashwerten macht dies keinen Sinn, da sie ja nur eingesetzt werden, wenn ohnehin keine Sortierrelation vorliegt.

2.6.4.3 Iteratoren

Die Schlüsselorientierung des Containers hat bekanntlich die Konsequenz, dass der Schlüssel nicht verändert werden darf, da ansonsten die Sortierung verloren gehen würde, das durch den Schlüssel indizierte Objekt, sofern es nicht mit dem Schlüssel identisch ist, aber sehr wohl. Die Iteratoren der Containerklassen präsentieren sich zwar nach Außen wie normale Iteratoren, agieren aber zumindest hinsichtlich der Schlüsselbegriffe wie konstante Iteratoren, das heißt die Zuweisung von Daten zum Schlüsselbegriff des Iterators wird vom Compiler nicht zugelassen. Soll also ein Objekt (*mitsamt dem Schlüssel*) durch ein anderes Objekt ersetzt werden, so

[21] Wer das schafft, ist gut, denn das haben bislang noch nicht mal die Mathematiker hinbekommen.

ist zunächst das vorhandene Objekt zu löschen, anschließend das Neue einzufügen (*Objektveränderungen ohne Wirkung auf den Schlüssel sind natürlich zulässig*).

Sie STL stellt für alle diese Containertypen Templateklassen zur Verfügung.

```
set<Key, HashFcn, EqualKey, Alloc>
map<Key, Data, HashFcn, EqualKey, Alloc>
multiset<Key, HashFcn, EqualKey, Alloc>
multimap<Key, Data, HashFcn, EqualKey,Alloc>
hash_set<Key, HashFcn, EqualKey, Alloc>
hash_map<Key, Data, HashFcn, EqualKey, Alloc>
hash_multiset<Key, HashFcn, EqualKey, Alloc>
hash_multimap<Key, Data, HashFcn, EqualKey, Alloc>
```

Die Containertypen `set` und `map` entsprechen den Schlüssel/Objektbeziehungen (a) und (b). Für die Containertypen `set` und `map` ist die Sachlage klar, für die korrespondierenden `hash`-Versionen gilt:

- Bei `set` wird entweder der Datenbereich des kompletten Objekts mit der Hashfunktion bearbeitet (*in diesem Fall erhält man beim Auslesen des Objektes aus dem Container keine neuen Informationen, da bereits alle zum Berechnen des Schlüssels verwendet wurden*) oder die Hashfunktion „weiß“, welche Teile des Objektes sie zur Ermittlung des Hashwertes heranziehen muss (*in diesem Fall können durch die Abfrage weitere Informationen gewonnen werden*). Da das Objekt direkt an den Hashwert gebunden ist, ist es nicht austauschbar.
- Bei `map` wird zusätzlich zum Objekt das für die Berechnung des Hashwertes verwendete Attribut separat angegeben. Das Objekt wird hierdurch austauschbar.
- Bei den `_multi_` - Versionen können mehrere Objekte mit gleichen Schlüsseln gespeichert werden. Spezielle Suchmethoden liefern Iteratorpaare, die den Bereich gleicher Schlüssel umfassen.

Der Containertyp `set` stellt mit den bisher diskutierten Containern teilweise kompatible Iteratoren zur Verfügung, wie folgende Zuweisungen zeigen:

```
set<string>::iterator its(st.begin());
vector<string>::iterator itv(vec.begin());
*itv=*its;              // ok
...
*its=*itv;              // ungültig, da its
                        // const_iterator
```

Der `set`-Iterator ist also immer ein `const _iterator`.

Für den Containertyp `map` gilt dies nicht mehr; er stellt einen Iterator vom Typ `pair<Key,Data>` zur Verfügung.

```
map<string>::iterator its(st.begin());
vector<string>::iterator itv(vec.begin());
```

```
vector<int>::iterator nt(vi.begin());
*itv=its->second;        // ok
its->second=*itv;        // ok
*nt=its->first;          // ok
its->first=*nt;          // verboten, da const
```

Ob der Schlüssel nach der Änderung des Objektes allerdings noch irgendeinen Sinn macht, bleibt dem Programmierer überlassen.

Speichern und Löschen von Objekten erfolgt mit Hilfe der Methoden

```
pair<iterator, bool> insert(const value_type& x);
iterator insert(iterator it, const value_type& x);
void insert(InIt first, InIt last);

iterator erase(iterator it);
iterator erase(iterator first, iterator last);
size_type erase(const Key& key);
```

Das Ergebnis eines `insert(..)`–Befehls ist vom verwendeten Containertyp abhängig. Sofern es sich nicht um einen `__multi__`–Container handelt, werden nämlich Objekte mit gleichen Schlüsseln nicht ein weiteres Mal eingefügt, was durch ungültige Iteratorwerte angezeigt wird. Bei sortierbaren Schlüsseln ist dies kein Problem, da in den meisten Fällen der Schlüsselbegriff eindeutig sein sollte; bei Hashfunktionen ist aber nicht grundsätzlich auszuschließen, dass deutlich verschiedene Objekte den gleichen Hashwert ergeben. Eine sorgfältige Auswahl der Hashfunktion ist deshalb notwendig.

Objekte lassen sich mit der Funktion `find(..)` finden, die einen Iterator als Rückgabewert liefert. Ein Objekt mit dem angegebenen Schlüssel existiert nicht, wenn `find(..)==end()` gilt, d.h. mit den Standardfunktionen kann man nur nach einem Objekt selbst suchen, nicht aber nach dem davor oder dahinter.[22]

Ein Teilbereich eines Baums kann mit den Methoden

```
iterator lower_bound(const Key& key);
iterator upper_bound(const Key& key);
```

ausgelotet werden. `lower_bound` liefert einen Iterator auf das Objekt mit nächstkleinerem Schlüssel, `upper_bound` entsprechend einen Iterator auf das Objekt mit nächstgrößerem Schlüssel. Zwischen diesen Werten kann nun beispielsweise sequentiell iteriert werden.

---

[22] Machen Sie sich aber bitte klar, dass das Objekt „vor“ oder „hinter“ dem Schlüssel bei Hash-Containern ohnehin keinen Sinn macht, da der Hashwert keine Ordnung auf den gehashten Objekten erzeugt. Bei Objekten mit <-Relation sieht das anders aus, und hier existieren auf entsprechend Methoden.

2.6.4.4 Einsatzbereiche

Wie wir noch zeigen werden, ist der suchende Zugriff in einem Baumcontainer nicht schneller als in einem sortierten Feldcontainer, und für beide Containertypen gilt, dass die Schlüssel nicht verändert werden dürfen, wenn die Sortierung erhalten bleiben soll. Für welche Einsatzbedingungen ist dann ein bestimmter Container geeigneter als ein anderer?

Operation	Feldcontainer	Baumcontainer
Suchen	O(log(n))	~O(log(n))
Einfügen/Löschen	O(n)	O(log(n))
Serieller Zugriff	O(1)	~O(1)
Indizierter Zugriff	O(1)	O(n)

Die Tabelle gibt noch einmal eine Übersicht über die Laufzeitordnungen, wobei der serielle Zugriff in einem Baum geringfügig schlechter ist als im Feldcontainer, da bei einem Vorschub immer zunächst an das `less`-Ende eines Zweiges gesprungen werden muss. Auch der Suchvorgang ist nicht ganz äquivalent, da die Balance im Rot-Schwarz-Baum nicht optimal ist. Anhand der verschiedenen Eigenschaften sollte die Auswahl des geeignetsten Typs möglich sein.

> **Aufgabe.** Zu einem späteren Zeitpunkt werden wir Werkzeuge zur Zeitmessung bereitstellen, die Sie sich aber im Vorgriff bereits besorgen können. Führen Sie Vergleichsmessungen zwischen den verschiedenen Containertypen durch und prüfen Sie die Laufzeiteigenschaften. Unterscheiden Sie dabei auch zwischen normaler und optimierter Übersetzung.
>
> Hinweis. da die Rechner heute recht schnell sind und sinnvolle Mindestlaufzeiten für Messungen bei einer Sekunde beginnen, müssen sehr große Container erzeugt werden; meist wird man sogar mehrere Durchläufe notwendig, um diese Größenordnungen zu erreichen. Achten Sie bei der Konstruktion solcher Programme aber peinlich darauf, dass auch tatsächlich die zu messenden Programmeinheiten die Laufzeit verursachen. Die Schleife
>
> ```
> for(int i=0; i<INT_MAX; ++i) j=j*i;
> ```
>
> verbraucht zwar auch Zeit, aber die hat nichts mit einem Sichvorgang in einem Container zu tun.

2.6.5 B+ – Bäume

Der Rot-Schwarz-Baum ist hinsichtlich der Balancierung nicht optimal, da der längste Weg bis zu einem Blatt maximal die doppelte Länge des kürzesten aufweisen kann. Es fehlt daher auch nicht an Algorithmen, die auf verschiedene Weise das

Optimum – maximal eine Einheit Unterschied – zu erreichen suchen. Eine Möglichkeit ist der B+ - Baum, der zwar primär kein binärer, sondern ein n-ärer Baum ist, aber formal auch zum Binärbaum abgemagert werden kann.

Bevor wir uns mit seiner Theorie auseinander setzen, sei vermerkt, dass eine optimale Balance den Suchvorgang zwar positiv beeinflussen kann, dafür aber beim Einfügen und Löschen sehr viel umfangreichere Umgruppierungen des Baumes verursachen. Die Auswahl des Rot-Schwarz-Baums als Standardalgorithmus in der STL ist deshalb nicht als fauler Kompromiss, sondern als wohlüberlegte Strategie zu sehen.

2.6.5.1 Konstruktionsprinzip

Wenn die Anzahl der Elemente sehr groß wird, wie es beispielsweise in Datenbanken der Fall ist, wird die Verwendung von binären Bäumen problematisch. Ähnlich wie bei der segmentierten Liste `deque` ist es dann sinnvoll, den Baum in Teilen zu verwalten und nur die Teile von der Festplatte zu laden, die gerade benötigt werden. Ein Weg, dies zu realisieren, ist die Zulassung von mehr als zwei Kindern pro Knoten. Wir führen hierzu einen weiteren Template-Parameter ein.[23]

```
template <typename T, int n=11> class BPPTree {
public:
private:

   struct Node{
      vector<T>     val;
      vector<Node*> gt;
      Node*         parent;
      Node*         less;

      Node(): parent(0), less(0){}
      ~Node(){
         vector<Node*>::iterator it;
         if(less) delete less;
         for(it=gt.begin();it!=gt.end();++it)
            delete *it;
      }//end destructor
   };//end struct

   Node* root;
};//end class
```

[23] Für die Speicherung der Elemente auf einem Knoten verwenden wir eine (sortierte) Liste (der Einfachheit halber als `vector` organisiert, was die Buchführung über belegte Elemente und Kinder etwas vereinfacht). Wenn die Anzahl der Elemente auf einem Knoten sehr groß wird, kann natürlich auch eine andere Containerform gewählt werden.

Jeder Knoten nimmt nun maximal *n*-1 Elemente auf, zu jedem Element wird ein Zeiger auf den Folgeknoten angelegt, in dessen Zweig (*also auch dessen Folgeknoten*) alle Elemente größer sind als das Bezugselement, aber kleiner als ein gegebenenfalls noch folgendes Element im Knoten. Zwei zusätzliche Zeiger verweisen auf den Elternknoten sowie einen Teilbaum, dessen Elemente kleiner sind als das kleinste des Knotens, aber größer als das des Elternknotens, von dem auf den Knoten verwiesen wird.

Um ein Element einzufügen, sucht man wie beim binären Baum zunächst wieder den untersten Knoten ohne weitere Nachfolger, in den das neue Element passt, und sortiert es dort in die Liste ein. Als Beispiel für einen Baum mit zwei Einträgen pro Knoten wählen wir folgende Konfiguration

```
K1(K2,5,K3,9,K4)
K2(-,1,-,3,-)     K3(-,7,-,8,-)     K4(-,11,-,13,--)
```

`K2` in `K1` verweist auf Knoten mit kleineren Einträgen als `5`, `K3` auf einen Baum mit Werten `5<x<9`, usw. Eingefügt werden soll ein Objekt mit der Schlüsselnummer **2**. Wir suchen den Endknoten, in den das Objekt passt, und fügen es dort ein, wobei wir bei Bedarf vorübergehend die Kapazität des Knotens über den vorgesehen Maximalwert erhöhen:

```
K1(K2,5,K3,9,K4)
K2(-,1,-,2,-,3,-) K3(-,7,-,8,-)    K4(-,11,-,13,--)
```

Knoten `K2` ist nun oberhalb seiner Kapazität belegt. Um die Überbelegung zu beseitigen, zerlegen wir den Knoten in zwei neue Endknoten mit gleichem Füllgrad und verschieben das mittlere Elemente in den Elternknoten. Das führt bei fortgesetztem Einfügen zur Anlage einer neuen Wurzel, und da diese gleich weit von allen Endknoten entfernt ist, ist der Baum immer balanciert. Die Balance ist somit besser als im Schwarz-Rot-Baum, denn jeder Weg von einem Knoten bis zu einem Blatt hat die gleiche Länge.[24]

Setzen wir diese Idee um: Wir bilden für das größte Element einen neuen Knoten, lassen das kleinste Element im alten Knoten und verschieben das mittlere Element und Mitziehen der Verweise in den Knoten darüber:

```
K1(K2,2,K5,5,K3,9,K4)
   K2(-,1,-,-,-)  K5(-,3,-,-,-)
   K3(-,7,-,8,-)  K4(-,11,-,13,--)
```

Ist hier wiederum die Kapazität überschritten, wiederholen wir die Aufspaltung. Dabei entsteht im Beispiel eine neue Wurzel:

```
K7(K1,5,K6,-,-)
   K1(K2,2,K5,-,-)
      K2(-,1,-,-,-)  K5(-,3,-,-,-)
```

[24] Das wird natürlich etwas durch die zusätzliche Suche in der Liste eines Knotens kompensiert.

```
K6(K3,9,K4,-,-)
   K3(-,7,-,8,-)  K4(-,11,-,13,--)
```

Wie man leicht nachvollzieht, ist die Knotenverteilung im Baum symmetrisch, allerdings muss eingeräumt werden, dass die Belegung der Knoten selbst in der Regel nicht symmetrisch ist und zwischen Eins und dem Maximalwert schwankt. Da die Geschwindigkeit der Zugriffe innerhalb der Knoten aber letztendlich doch von der Ordnung $O(1)$ ist, ist die Suche in einem solchen Baum von der Ordnung $O(\log(n))$.

2.6.5.2 Einfügeoperationen

Wir beginnen mit der Implementation des `insert`-Algorithmus.

> **Aufgabe.** Das Auffinden des Endknotens und das Einfügen des neuen Elementes in die Liste ist trivial, und ich überlasse Ihnen die Implementation bis zu dieser Stelle.

Ist ist Kapazität eines Knotens überschritten, rufen wir nun die Methode `SplitNode(Node*w, bool recurs)` auf, die rekursiv die Erzeugung neuer Knoten bis hin zu einer neuen Wurzel übernimmt. Geteilt wird der Knoten in der Mitte, d.h. bei `m=size()/2`.(*wie wir beim Löschen noch sehen werden, kann ein zu teilender Knoten auch deutlich mehr als ein überschüssiges Element enthalten*). Dazu wird ein neuer Knoten `ww` erzeugt, der die Elemente einschließlich der Knotenverweise von `(m+1)..size()-1` sowie den Knotenverweis von `m` auf `less` erhält, das Element `m` wird zwischengesichert und die Elemente im alten Knoten auf `0..m-1` beschränkt.

```
int i,mitte=w->val.size()/2;
Node* ww=new Node(w->parent);
if(w->gt.size()!=0){
   ww->less=w->gt[mitte];
   ww->less->parent=ww;
}//endif
T t=w->val[mitte];

for(i=mitte+1;i<w->val.size();i++){
   ww->val.push_back(w->val[i]);
   if(w->gt.size()!=0){
      ww->gt.push_back(w->gt[i]);
      w->gt[i]->parent=ww;
   }//endif
}//endfor
while(w->val.size()>mitte) w->val.pop_back();
while(w->gt.size()>mitte) w->gt.pop_back();
```

Wie Ihnen beim Lesen des Kodes sicher aufgefallen ist, ist die Rekursion bereits berücksichtigt: besitzt der Knoten Unterknoten, so sind die Verweise auf diese Knoten ebenfalls zu kopieren und (!) die `parent`-Verweise für die in den neuen Knoten kopierten Daten zu korrigieren.

Das frei gewordene mittlere Element wird nun zusätzlich in den Elternknoten eingefügt und erhält den Verweis auf den neuen Konten `ww`. Am Verweis auf den Knoten `w` ändern wir nichts. Die Position, an der wir Einfügen, kann durch Suche der Einfügeposition des Elementes `t` im Feld `val` oder durch Suche des Verweises auf `w` in `less` und im Feld `gt` erfolgen. Bei der Ausführung der Suche ist darauf zu achten, dass im ersten Fall eine Binärsuche durchgeführt werden kann, im zweiten Fall aber eine lineare Suche durchgeführt werden muss (!), da die Zeigerliste unsortiert ist. Wir verwenden hier die zweite Suchversion; in Anwendungen mit größeren Feldern (*Datenbanken*) wird man die Binärsuche vorziehen.[25]

```
typename vector<T>::iterator jt;
typename vector<Node*>::iterator it;
if(w->parent->less==w){
   w->parent->val.insert(w->parent->val.begin(),t);
   w->parent->gt.insert(w->parent->gt.begin(),ww);
}else{
   it=std::find(w->parent->gt.begin(),
                w->parent->gt.end(),w);
   it++;
   it=w->parent->gt.insert(it,ww);
   jt=w->parent->val.begin();
   advance(jt,distance(w->parent->gt.begin(),it));
   w->parent->val.insert(jt,t);
}//endif
SplitNode(w->parent);
```

Nach Einfügen des Elements wird auch der Elternknoten untersucht, ob eine weitere Aufspaltung notwendig ist.

Falls kein Elternknoten zum Einfügen des freien Elementes mehr vorhanden ist, der alte Knoten also die Wurzel darstellt, muss eine neue Wurzel gebildet werden. Diese erhält `w` als `less`-Verweis.

```
if(w->parent==0){
   root=w->parent=ww->parent=new Node();
```

[25] Bein Einsatz einer Binärsuchmethode ist allerdings darauf zu achten, dass diese an der richtigen Stelle anhält. Das gesuchte Element ist garantiert nicht in der Liste enthalten, was Suchalgorithmen in der Regel nicht mir der Einfügeposition, sondern mit dem Enditerator beantworten. Die Suchmethode ist daher für diese Zwecke entsprechend zu modifizieren.

```
   root->val.push_back(t);
   root->gt.push_back(ww);
   root->less=w;
}else{
   ... // vorhergehender Code
}//endif
```

2.6.5.3 Suche und Traversieren

Für Suchen im Baum und für das Traversieren werden Iteratoren benötigt. Beginn und Ende sind leicht konstruierbar:

(a) Der Startiterator verweist auf das erste Element im Endknoten der von der Wurzel ausgehenden `less`-Kette.
(b) Der Enditerator enthält den Knotenverweis Null

Dazwischen muss die Konstruktion berücksichtigen, dass nach Abarbeitung aller Objekte eines Endknotens in den Elternknoten zurückgesprungen werden muss. Dort ist das nächste Element zu indizieren und bei erneutem Vorrücken in den Unterknoten zu verzweigen (*das Ganze rekursiv, wenn der Elternknoten vollständig abgearbeitet ist und auf den Großelternknoten zurückgesprungen wird*). Neben dem aktuellen Knoten ist somit im Iterator auch zu speichern, welches das Bezugselement im Elternknoten ist. Wir realisieren dies mit einem Stack:

```
template <typename T1, typename T2, int n>
class BPPTreeIterator {
public:
   BPPTreeIterator():node(0){pos.push(0);}
   ~BPPTreeIterator(){}
   ...
   inline reference operator*()  const
      { return node->val[pos.top()]; }
   inline pointer   operator->() const
      { return &(node->val[pos.top()]); }
private:
   stack<int> pos;
   typename BPPTree<T1,n>::Node* node;
};//end class

template <typename T, int n=11> class BPPTree {
public:
  typedef BPPTreeIterator<T,T*,n>       iterator;
```

```
  typedef BPPTreeIterator<T,T const*,n>
                                  const_iterator;
  ...
```

Um komplexen Typauflösungen zunächst zu entgehen (*wir kommen in späteren Kapiteln darauf zurück*), versehen wir den Iterator, den wie als reinen Vorwärtsiterator konzipieren, mit zwei Template-Parametern zur Unterscheidung der variablen und konstanten Form.[26]

Die Verwendung lässt sich am Inkrementoperator erklären. An der Spitze des Stacks wird die aktuelle Position im aktiven Knoten gespeichert. Ist der Knoten ein Endknoten (`less==0`), wird das oberste Stackelement inkrementiert und beim nächsten Zugriff das entsprechende Knotenobjekt zugänglich gemacht. Ist die Liste vollständig abgearbeitet, wird das oberste Stackelement entfernt, zum Elternknoten gewechselt und das dazugehörende Stackelement inkrementiert, so dass das nächste Element des Elternknotens beim nächsten Zugriff verwendet wird. Beim nächsten Inkrementieren wird wieder in den Verweisbaum bis zum kleinsten Element abgestiegen, wobei für jede Ebene eine Null auf dem Stack abgelegt wird. Ist auch der Elternknoten abgearbeitet, so kann zu dessen Elternknoten zurückgesprungen und in gleicher Weise fortgefahren werden. Insgesamt führt dies zu dem Code

```
BPPTreeIterator<T1,T2,n>& operator++(){
   if(node==0)                          return *this;
   if(node->less==0){
      pos.top()++;
      if(pos.top()<node->val.size()) return *this;
   }//endif
   if(pos.top()>=node->val.size()){
     while(node!=0 & pos.top()>=node->val.size()){
        node=node->parent;
        pos.pop();
     }//endwhile
     return *this;
   }//endif
   node=node->gt[pos.top()];
   pos.top()++;
   pos.push(0);
   while(node->less!=0){
      node=node->less;
      pos.push(0);
   }//endwhile
return *this;}//end function
```

[26] Bezüglich der für Kompatibilität mit STL-Iteratoren notwendigen `typedef`-Definitionen siehe dort.

Aufgabe. Implementieren Sie die Methode

```
iterator find(T const& t);
```

zum Suchen nach einem Objekt im Baum. Der Iterator verweist wie üblich auf das gesuchte Element oder auf den Enditerator.

2.6.5.4 Löschen von Elementen

Das Löschen von Einträgen erfolgt in Umkehrung des Einfügens. Das Löschen in einem Blatt mit mehr als einem Eintrag ist zunächst problemlos: der Unterknoten, auf den der Verweis des zu löschenden Eintrags führt, wird mit dem Unterknoten des Vorgängers bzw. dem `less`-Zweig zusammengefügt und der Eintrag gelöscht. Wir nutzen für das Finden des zu löschenden Eintrags die Funktion aus den Aufgaben.

```
typename BPPTree<T,n>::iterator it=find(t);
Node* w;
if(it==end()) return;
it.node->val.erase(
              it.node->val.begin()+it.pos.top());
if(it.node->less!=0){
   w=it.node->gt[it.pos.top()];
   it.node->gt.erase(
              it.node->gt.begin()+it.pos.top());
   if(it.pos.top()==0) Join(it.node->less,w);
   else       Join(it.node->gt[it.pos.top()-1],w);
}//endif
```

Das Zusammenfügen der Unterknoten ist ein rekursiver Vorgang. Besitzen die Unterknoten weitere Kindknoten, so ist der Knoten des höchsten Elementes des linken Knotens mit dem `less`-Knoten des rechten zu verbinden. Hierbei ist ein wenig auf die Reihenfolge der Operationen zu achten. Bei der Zusammenlegung von Knoten kann es natürlich zum Überschreiten der vorgesehenen Maximalgrößen kommen, so dass anschließend die Aufspaltungsmethode aufzurufen ist. Da diese in der entgegengesetzten Richtung arbeitet – das Zusammenfügen arbeitet sich zu den Endknoten vor, während das Aufspalten von den Endknoten zur Wurzel zurückläuft –, müssen die Knoten vollständig zusammengefügt sein, bevor die Rekursion fortgesetzt wird.

```
template <typename T, int n>
void BPPTree<T,n>::Join(Node* w, Node* v){
    int i;
    Node* le=0,* gt=0;
    if(w==0)  return;
    if(w->less){
```

```
        le=v->less;
        if(w->gt.size()>0)
            gt=w->gt.back();
    }//endif

    for(i=0;i<v->val.size();i++){
        w->val.push_back(v->val[i]);
        if(w->less){
            w->gt.push_back(v->gt[i]);
            v->gt[i]->parent=w;
        }//endif
    }//endfor

    if(le){
        if(gt)   Join(gt,le);
        else     Join(w->less,le);
    }//endif

    v->gt.clear();
    v->less=0;
    delete v;
    SplitNode(w);
}//end function
```

Abschließend ist der frei werdenden Knoten zu löschen, da kein Verweis mehr auf ihn existiert, wobei allerdings zuvor die Verweiszeiger zu löschen sind, damit der Destruktor nicht noch in Benutzung befindliche Knoten löscht. Achten Sie im Code auf eine Feinheit: nach unseren bisherigen Untersuchungen sind im linken Knoten Elemente vorhanden, mit dessen höchsten Verweisknoten der `less`-Knoten des rechten Knotens verbunden wird. Wie wir gleich noch sehen werden, sind auch Knoten möglich, die keine Elementeinträge mehr besitzen. In diesem Fall sind die `less`-Einträge beider Knoten zu verknüpfen. Wir haben dies in der Implementation gleich mit berücksichtigt.

Komplizierter wird das Verfahren, wenn durch das Löschen der letzte Eintrag eines Knotens verschwindet. Man überzeuge sich zunächst davon, dass wir diese Situation erst untersuchen müssen, nachdem das oben beschriebene Löschen, Zusammenfügen und Aufspalten erfolgt ist: wird das letzte Element gelöscht, so erfolgt die Zusammenlegung dessen Unterknotenverweises mit dem `less`-Knoten, der nun der einzige Verweis in diesem Knoten ist. Die `SplitNode`-Methode interessiert sich aber nicht dafür, dass der Knoten ansonsten leer ist, sondern fügt das Element am Anfang der Liste ein, wodurch das Problem beseitigt wäre.

Ist die Liste auch nach dem Zusammenlegen leer, so muss nun in Umkehrung des Einfügens ein neues Element aus dem Elternbereich heruntergezogen werden. Das Entfernen von Elementen darf nämlich nur zum Verschwinden der Wurzel, nicht aber von anderen Knoten führen. Bei der Entfernung von Elternelementen sind aber einige Regel zu beachten:

i. Der leere Knoten ist ein Endknoten, besitzt also keine Nachfolger.

   (a) Der übergeordnete Verweis erfolgt von einem Element. Dieses kann auf den leeren Knoten kopiert und im Elternknoten gelöscht werden. Da hierdurch aber der Verweis auf den Knoten entfällt, muss dieser mit dem Knoten des Vorgängerelementes im Elternteil oder mit dem `less`-Knoten verknüpf werden.
   (b) Der übergeordnete Verweis ist der `less`-Verweis. Das erste Element des Elternknotens kann kopiert und im Elternknoten gelöscht werden, wobei der entfallende Verweis an den Knoten angefügt werden muss.

ii. Der Knoten ist ein Mittelknoten, besitzt also Nachfolger.

   (a) Im Elternknoten ist ein Elementverweis vorhanden, d.h. das Elternelement ist kleiner als alle Elemente in den Kindknoten. Es darf also nicht einfach in den freien Knoten kopiert werden, da hierdurch der `less`-Zweig die Reihenfolge verletzt. Die einzige Position, an die das Element kopiert werden kann, ist der Beginn des Endknotens des `less`-Zweiges.

   Der Eintrag im Elternknoten kann nun gelöscht werden, der höchste Verweis des Vorgängers wird mit dem `less`-Zweig des Mittelknotens verbunden. Der Mittelknoten wird gelöscht.
   (b) Weist `less` auf den freien Knoten, ist das erste Elternelement größer als alle Elemente im Unterbaum. Wir fügen es an das Ende des höchstwertigen Endblattes an und löschen es aus dem Elternknoten.

   Hierdurch wird der Verweisknoten des verschobenen Elementes frei. Die `less`-Einträge der beiden Knoten werden verbunden und an den Verweisknoten gebunden, der anschließend die neue Mitelknotenrolle übernimmt.

iii. Der Knoten ist der Wurzelknoten. Da nur noch der less-Zweig vorhanden ist, bildet less die neue Wurzel, der alte Wurzelknoten wird gelöscht.

Auch hierbei ist nach dem Verbinden von Knoten zu überprüfen, ob ein erneutes Aufspalten notwendig ist, d.h. die Arbeit an den Knoten ist komplett abzuschließen, bevor die Verbindung durchgeführt wird. Die Implementation wird leider zu einer ziemlichen Zeigerakrobatik. So lange das Wurzelelement nicht erreicht ist, können wir iterativ vorgehen (*Schritte i. und ii.*)

```
while(it.node->val.size()==0 &&
     it.node->parent!=0){
```

Die Teilschritte (i.a) und (ii.a) bzw. (i.b) und (ii.b) können miteinander verbunden werden. Wir beginnen mit dem Teil (b)

```
  if(it.node==it.node->parent->less){
      if(it.node->less!=0){
         w=it.node->less;
         while(w->less!=0) w=w->gt.back();
      }else{
```

```
         w=it.node;
      }//endif
      w->val.push_back(it.node->parent->val[0]);
      ww=it.node->parent->gt[0];

      it.node->parent->val.erase(
                  it.node->parent->val.begin());
      it.node->parent->gt.erase(
                  it.node->parent->gt.begin());
      SplitNode(w);
      Join(it.node,ww);
      it.node=it.node->parent;
   }else{
```

und fahren fort mit Teil (a)

```
      w=it.node;
      vi=std::find(w->parent->gt.begin(),
                   w->parent->gt.end(),w);
      pos=distance(w->parent->gt.begin(),vi);

      while(w->less!=0) w=w->less;
      w->val.insert(w->val.begin(),
                    it.node->parent->val[pos]);

      it.node->parent->val.erase(
               it.node->parent->val.begin()+pos);
      it.node->parent->gt.erase(vi);
      SplitNode(w);
      w=it.node->parent;
      if(pos>0)
         Join(it.node->parent->gt[pos-1],it.node);
      else
         Join(it.node->parent->less,it.node);
      it.node=w;
   }//endif
}//endwhile
```

Die Wurzel (iii.) ist am Einfachsten zu bereinigen

```
if(it.node->val.size()==0){
   if(it.node->less!=0){
      it.node->less->parent=0;
      root=it.node->less;
      it.node->less=0;
      delete it.node;
```

```
  }//endif
}//endif
```

Mittels der Überlegungen (i.a) – (iii) dürfte das Nachvollziehen des Kodes und das Ergänzen einiger Variablen kein Problem darstellen. Falls Sie weitere Experimente dazu vornehmen wollen, achten Sie darauf, dass nach einem Zusammenfügen von Knoten eine Prüfung auf Aufspaltung erfolgen muss, da sonst leicht übergroße Knoten entstehen. Eine Aufspaltung setzt aber voraus, dass der Baum konsistent ist, also auf für einen späteren Spleissvorgang entnommene Knoten nicht mehr verwiesen wird. Außerdem verlieren durch einen Spleissvorgang die Positionszeiger auf den oberen Ebenen ihre Gültigkeit, so dass sie im weiteren Verlauf der Iteration nicht mehr verwendet werden dürfen.

Aufgabe. Ergänzen Sie eine Implementation um einige Prüf- und Debugmethoden. Beispielsweise können die Knoteninhalte so formatiert ausgedruckt werden, dass die Abhängigkeit der Knoten sichtbar wird. Eine Prüffunktion kann sicherstellen, dass ausgehend von der Wurzel alle Verweise korrekt sind. Für die Reihenfolgeprüfung kann die bereits beschriebene Prüffunktion für sortierte Container verwendet werden.

2.6.5.5 Abschlussbemerkungen

Magern wir die Anzahl der Elemente in einem Knoten auf zwei ab, so haben wir eine immer maximal balancierte Form des binären Baum vor uns, d.h. zwei von einem Knoten ausgehende Weglängen unterschieden sich maximal um eine Einheit. Da aber fast jede Einfüge- oder Löschoperation zu umfangreichen Neusortierungen des Baumes führt, ist eine solche Implementation kontraproduktiv.

Strukturen und Algorithmen dieser Art sind Basis für die Konstruktion von Dateisystemen und Datenbanken. Für das Design solcher Anwendungen ist die Berücksichtigung weiterer Gesichtspunkte notwendig, wie etwa die Aufteilung des Datenbestandes auf einen auf der Platte und einem im RAM gelagerten Teil. Weichen Anforderung von Anwendung und Datenhaltungsdesign voneinander ab, kann das erhebliche Auswirkungen auf die Geschwindigkeit haben. Wir können jedoch nicht weiter in diese Gebiete eindringen und verweisen auf entsprechende Spezialliteratur.

2.7 Algorithmen und Container

Es ist klar, dass man nicht nur Objekte in Containern sammeln, sondern auch bestimmte Berechnungen mit ihnen oder auf ihnen durchführen möchte. Der eine oder andere Containertyp besitzt denn auch neben den reinen Zugriffsschnittstellen bereits Methoden, die bestimmte Algorithmen auf dem Inhalt durchführen. Die meisten Berechnungen sind recht spezieller Natur, es existieren aber auch einige, die von fast allen Anwendern und unabhängig vom Containertyp benötigt werden

oder in Zukunft benötigt werden könnten. Wollte man nun die Container in herkömmlicher Weise durch entsprechende Methoden erweitern, so stünde man vor einigen Problemen: Es würden ständig neue Versionen der Klassen erstellt, deren Herausgabe koordiniert werden müsste, und für den in einer Anwendung eingesetzten Container stünde der benötigte Algorithmus gerade nicht zur Verfügung. Aus diesem Grund wurde für die STL ein anderer Weg gewählt: Für jeden Algorithmus kann eine allgemeine Lösung implementiert werden, die das Ergebnis auf der Grundlage einer C–Zeigerarithmetik ermittelt. Sofern ein Algorithmus auf einem bestimmten Containertyp zulässig ist (*Sortieren eines* `set` *–Containers mach zum Beispiel keinen Sinn*), muss er mit Hilfe des Iteratorkonzeptes damit auch lauffähig sein – wie holprig das im Einzelfall auch immer sein mag. Algorithmen werden daher als Templatefunktionen unabhängig von den Containern definiert und implementiert (*Templateparameter der Funktionen sind Iteratoren*). Wie wir noch sehen werden, ist damit sogar die Möglichkeit gegeben, Elemente unterschiedlicher Container in einem Algorithmus zu verarbeiten. Werden nun weitere Algorithmen implementiert, so verlängert sich die Liste der Algorithmen um einige Methoden, und weder der Verwaltungsaufwand noch die mangelnde Verfügbarkeit treten als Problem in Erscheinung. Darüber hinaus ist es auch möglich, die Algorithmen für bestimmte Containertypen zu spezialisieren. Der Anwender kann die Spezialisierung nach dem nächsten Update der Bibliothek nutzen, ohne dass er sich darum kümmern muss.

Sehen wir uns zunächst für einige Algorithmen die Theorie etwas genauer an. Immer wieder benötigt werden Algorithmen für das Suchen bestimmter Elemente in einem Container oder das Sortieren der Objekte eines Containers.

2.7.1 Sortierrelationen

Um nach einem bestimmten Element in einem Container zu suchen, benötigt man eine (als Klasse implementierte) Relation `equal` zwischen Objekten. Für Sortierungen ist zusätzlich eine Relation less notwendig, da mit der Relation `equal` zwar zwischen Objekten unterschieden werden kann, dies aber noch nicht zu einer Reihenfolge der Elemente führt. Suchen und Sortieren unterscheiden sich außerdem dadurch, dass das Referenzobjekt beim Suchen von Außen kommt, während die Sortiervorgänge sich vollständig innerhalb eines Containers abspielen.

Bei einem Such- oder Sortiervorgang bedeutet $\neg(a < b)$ allerdings nicht automatisch $(b < a)$, sondern es kann ja auch der Fall $(a = b)$ vorliegen. Für Sortierungen an sich ist das relativ egal, für die Suche nach einem Element jedoch nicht unbedingt, denn wenn wir uns um diese Unterscheidung beim Sortieren nicht kümmern, haben wir ja nun unter Umständen mehrere Elemente im Container, die die Suchrelation mit dem von Außen vorgegebenen Objekt erfüllen. In sortierten Containern wird zur Vermeidung von Mehrdeutigkeiten und Missverständnisse deshalb eindeutig festgelegt, ob mehrere Elemente mit gleichen Schlüsseln zulässig sind. In unsortierten Containern, die manuell nachsortiert werden, werden Mehrfacheinträge meist beibehalten, da es nicht so ohne weiteres zulässig ist, einfach Elemente zu löschen. Zur Beseitigung der Mehrdeutigkeiten existieren zwar

auch Algorithmen, man muss sich aber darüber im Klaren sein, dass hierdurch Informationen verloren gehen können, ohne dass der Anwender große Steuerungsmöglichkeiten besitzt.

Warum Klassen `less` und `equal` und nicht Funktionen/Operatoren? Sinngemäß sind die beiden Vergleiche tatsächlich zunächst in der erwarteten Form implementiert:

```
template <class T> struct equal {
   inline bool operator()(T const& a, T const& b){
      return a==b;
   }//end function
};//end struct
template <class T> struct less {
   inline bool operator()(T const& a, T const& b){
      return a<b;
   }//end function
};//end struct
```

Der Grund für diesen Umweg liegt in der Gestaltung von Algorithmen auf Containern durch den Anwender. Wenn bei Sortier- und Suchvorgängen auf Containern spezielle Implementationen der Relationen verwendet werden sollen, muss eine Möglichkeit bestehen, diese auch in die Algorithmen zu importieren. Die Importmöglichkeit ist durch Templates gegeben, allerdings kommen Templates mit Klassen wesentlich besser zu Rande als mit Funktionen, für die gewissen Einschränkungen bestehen. Womit wir auch schon den Grund für diese Art der Relationenbeschreibung (und anderer Operationen) beschrieben haben.[27]

Aufgabe. Implementieren Sie eine Methode `equal` für den Vergleich zweier `double`-Werte. Die Werte sind als gleich anzusehen, wenn sie in den ersten 3 Dezimalstellen übereinstimmen.

Wir wollen im folgenden die wichtigsten Such- und Sortieralgorithmen anreißen. Die Implementierungen sind nur als Lehrbeispiele zu verstehen; in der Anwendungsprogrammierung sollte man auf die STL-Algorithmen zurückgreifen, da auf den internen Containerstrukturen weitere Optimierungen möglich sind. Die Grundstruktur der Implementation der vorgestellten Algorithmen ist von der Form

```
template <class Iter,
   template <> class Relation = Standard_relation>
Iter function(Iter start, Iter ende, ...
              Relation releation, ...){...}
```

und sollte daher mit allen Standard-Containern funktionieren.[28]

---

[27] Für die Standardrelationen und -Operationen sind allgemeine Templates definiert, so dass im Einzelfall nur noch Spezialisierungen notwendig sind.

[28] Den Relationsteil lassen wir zur Vereinfachung im Folgenden fort. Sie können ihn bei eigenen Versuchen ergänzen.

2.7.2 Suchen in unsortierten Containern

Besteht nur eine Relation `equal` zwischen den Objekten oder ist der Container nicht sortiert, so bleibt nur die Strategie, den Container sequentiell zu durchsuchen, bis das gesuchte Element gefunden ist. Der Suchaufwand ist proportional zur Anzahl der Elemente im Container, d.h. $O(n)$.

Ist zwischen den Objekten zusätzlich die Relation `less` gegeben, kann die Suche auch nach dieser Relation durchgeführt werden, das heißt wenn ein Objekt mit den gewünschten Eigenschaften nicht gefunden wird, kann das nächstkleinere oder nächstgrößere ausgegeben werden. Allerdings handelt es sich hierbei um Spezialvereinbarungen. Im Standard wird bei Nichtfinden eines Objektes der End-Iterator zurückgegeben.

Aufgabe. Implementieren Sie einen Suchalgorithmus, der das nächstkleinere Element zu einem gegebenen findet und einen Iterator darauf ausgibt.

2.7.3 Suchen in sortierten Containern

In sortierten Containern kann die Suche mit dem Aufwand $O(\log(n))$ durchgeführt werden, sofern der Container indexierte Zugriffe erlaubt (*oder ein Baum ist. Aber da dieser Fall bereits diskutiert wurde, beschränken wir uns auch Listencontainer*). Man beginnt mit der Suche in der Mitte des Containers. Ist das Element damit gefunden, ist die Suche beendet, ist das dort gefundene Element kleiner als das gesuchte, muss dieses sich zwischen den Momentanposition und dem Ende befinden und man kann den linken Startpunkt auf die Mitte verschieben und das verbleibende Intervall nach der gleichen Regel untersuchen. Entsprechend wird das andere Halbintervall untersucht, wenn das mittlere Element größer als das gesuchte ist. Mit einigen kleinen Anpassungen, die aus der Iteratorkonvention herrühren, erhält man so den Algorithmus

```
template <typename Iter, typename T>
Iter find_s(Iter beg, Iter end, T val){
    equal_to<T> eq;
    less<T> le;
    Iter itb(beg),it,ite(end);
    if(eq(val,*beg)) return beg;
    while(true){
        it=itb;
        advance(it,distance(itb,ite)/2+1);
        if(it==ite)          return end;
        else if(eq(val,*it)) return it;
        else if(le(*it,val)) itb=it;
        else                 ite=it;
    }//endwhile
}//end function
```

Die Prüfung wird an der Stelle `(size/2+1)` durchgeführt, so dass bei einem verbleibenden zu überprüfenden Element der Endeiterator der Teilliste erreicht und damit das Ende angezeigt wird. Da auf diese Weise das erste Listenelement nicht erreicht werden kann, muss es separat überprüft werden (*Startbereich der Funktion*). Ist das Ende einer Teilliste erreicht, ohne dass das gesuchte Element gefunden worden wäre, wird der Endeiterator der Liste zurückgegeben. Ansonsten erfolgt die Rückgabe des Iterators oder das Setzen der neuen (*virtuellen*) Intervallgrenzen.

> **Aufgabe.** Der Algorithmus ist durch die Verwendung von advance und distance so konstruiert, dass er mit jedem Listencontainer fiunktioniert. Ermitteln Sie die Laufzeitordnung O(*n*) für die Containertypen vector, deque und list aus der STL in einem praktischen Versuch. Hilfsmittel zur Ermittlung der Laufzeit finden Sie im nächsten Kapitel.

2.7.4 Bubblesort-Sortieralgorithmus

Der wohl einfachste und einleuchtenste Sortieralgorithmus ist der Bubble-Sort-Algorithmus, bei dem jedes Element mit dem nachfolgenden verglichen und das jeweils kleinere durch Austausch mit dem größeren an die vordere Position geschrieben wird. Der Algorithmus wird dann mit dem zweiten Element fortgesetzt. Am Schluss eines Durchgangs steht das kleinste Element an erster Stelle, und die Wiederholung der Sortierung mit der nächsten Position als Startpunkt führt iterativ zum gewünschten Ergebnis. Bildlich „perlen“ die großen Elemente wie „Blasen“ an das Ende der Liste, woher der Algorithmus seinen Namen hat.[29]

Algorithmisch wird diese Beschreibung durch zwei geschachtelte Schleifen dargestellt, d.h. der Rechenaufwand für eine Sortierung steigt quadratisch mit der Anzahl der Objekte im Feld. Man kann den Algorithmus jedoch vorzeitig abbrechen, wenn man feststellt, dass bereits alles sortiert ist und ein weiterer Schleifendurchlauf keine neuen Ergebnisse bringt. Eine Sortierung liegt vor, wenn in einem Durchlauf der äußeren Schleife kein Austausch zwischen Elementen vorgenommen wird. Der Gesamtalgorithmus erhält somit die Form:

```
template <typename Iter>
void bsort(Iter beg, Iter end){
    Iter it1,it2,its;
    bool not_ready;
    for(not_ready=true;beg!=end&&not_ready;--end){
        it1=it2=beg;
        for(++it2,not_ready=false;it2!=end;
                                ++it1,++it2){
            if(*it2<*it1){
                swap(*it1,*it2);
```

[29] Ich persönlich habe den angelsächsischen Humor schon immer für ebenso merkwürdig wie die Essgewohnheiten gehalten.

```
                not_ready=true;
            }//endif
        }//endfor
    }//endfor
}//end function
```

Anmerkung. Der Algorithmus ist auch in einer anderen Implementation denkbar, in der nicht nur nebeneinander liegende Objekte getauscht werden, sondern das durch die äußere Schleife indizierte Objekt in der Inneren gegen alle anderen verglichen wird. In dieser Form kann der Algorithmus jedoch nicht abgebrochen werden, da ein fehlender Austausch nicht impliziert, dass bereits alle Elemente in der richtigen Reihenfolge sind.

Der Aufwand des Bubblesortalgorithmus ist interessanterweise nicht konstant, sondern liegt somit je nach Sortierungsgrad bei

$$O(n) \leq O_{bubble} \leq O(n^2)$$

Für völlig unsortierte Container ist dieser Algorithmus der uneffektivste. Ist der Container jedoch weitgehend sortiert, kann der Bubblesort zum effektivsten Algorithmus mutieren, beispielsweise beim nachträglichen Einfügen eines Elementes in einen bereits sortierten Container.[30] Die Antwort auf die Frage „wer ist der Beste?“ ist also alles andere als trivial.

2.7.5 Quicksort-Sortieralgorithmus

Beim Quicksortalgorithmus wird nicht wie beim Bubblesortalgorithmus versucht, ein Element sofort an die korrekte Position zu schieben, sondern die Liste wird mittels eines Schlüsselwertes (*Pivotelement*) zunächst in zwei Teillisten zerlegt. Eine Teilliste nimmt alle Objekte auf, die kleiner als das Pivotelement sind, die andere die restlichen. Der Vorgang wird rekursiv mit den beiden Teillisten fortgesetzt, bis die entstehenden Listen nur noch gleich große Elemente enthalten. Im Rückwärtsschritt werden die Listen aneinander gehängt und die sortierte Gesamtliste erzeugt.

Bei optimalem Verlauf wird die Länge der Listen in jedem Schritt halbiert, d.h nach ld(n) Schritten ist der Teilungsvorgang zu Ende. Da in jeder Runde alle Objekte geprüft werden müssen, wird das Zeitverhalten durch $n * ld(n)$ beschrieben. Beim schlechtesten Verlauf wird immer nur ein Element abgespalten, so dass insgesamt

$$O(n * ld(n)) \leq O_{quick} \leq (n^2)$$

[30] Man beachte aber: er wird in diesem Fall in der Regel immer noch von Baumcontainern übertroffen!

folgt. Eine besondere Abbruchbedingung bei einer Vorsortierung gibt es nicht.

Im Vergleich mit dem Bubblesort lässt sich nur aufgrund dieser theoretischen Überlegungen feststellen:

- Bei unsortierten Containern ist der Quicksort dem Bubblesort überlegen,
- bei weitgehend sortierten Containern (Einfügeproblem) ist der Bubblesort meist deutlich überlegen.

In der Praxis kann man sich daher durchaus Gedanken darüber machen, welchen der beiden Algorithmen man in welchen Status der Rechnung auf das Sortierproblem anwendet.

Das Problem ist offenbar die Auswahl eines geeigneten Pivotelements. Leider muss jede Optimierung in dieser Richtung mit einem so hohen Aufwand bezahlt werden, dass die mehr oder weniger zufällige Auswahl eines beliebigen Elementes genügen muss. Es genügt allerdings auch nicht, nur irgendein Element herauszugreifen, wie die folgende Überlegung zeigt: sortiert man nach der Eigenschaft `less` und erwischt zufällig das kleinste Element der Liste, bleibt eine der entstehenden Listen leer und der Stack läuft bei Fortsetzung der Rekursion über (*das Gleiche passiert mit jeder anderen Form der Entscheidung, wie man leicht überlegt*). Vermieden wird dies, wenn man sicherstellt, dass mindestens ein vom ersten Element verschiedenes weiteres in der Liste vorhanden ist und man das größere der beiden als Pivotelement verwendet. Wir verwenden hier als Kontrollelement das erste und als Kandidat das mittlere Element, weil es beim Nachsortieren bereits einmal sortierter Container die besten Ergebnisse liefert. In diesem Fall zerfällt die Liste nämlich jeweils in fast gleich große Teillisten. Der Leser mache sich im Übrigen klar, dass gut gemeinte Empfehlungen wie „*benutze drei Elemente und berechne den Mittelwert*“ wenig hilfreich sind. Sofern man Zahlen sortiert – was sicher bei einer Versuchsimplementation der erste Ansatz sein dürfte – ist das eine brauchbare Methode, aber wie ist der Mittelwert bei einer lexikalischen Sortierung von Worten zu bilden? Mit anderen Worten, es existieren viele Mengen, zwischen deren Elementen eine natürlich Ordnung „<“ definiert ist, eine Operation „+“ vielleicht noch einen Sinn macht (*Aneinanderketten von Strings*) , die für die Mittelwertbildung notwendige Operation „/(int)“ aber abseits jeder Realität liegt.[31]

In vielen Implementierungen des Algorithmus wird nun tatsächlich mit verschiedenen Listen gearbeitet, auf die die Objekte aufgeteilt werden. Das eignet sich aber nur für den Container „list“, sofern man direkten Zugriff auf die Knoten hat. Wir

[31] Natürlich könnte man einigermaßen verkrampft versuchen, auch für Strings einen Mittelwertsbegriff zu definieren. Allerdings würde das dazu führen, dass man solche Überlegungen für jede zu sortierende Sondermenge erneut durchführen muss, d.h. die universelle Einsetzbarkeit des Algorithmus wäre nicht mehr gegeben.

implementieren den Algorithmus hier ohne Aufteilung und in einer Form, die wieder auf jedem Listencontainer funktioniert:

```
template<typename Iter>
void qsort(Iter beg, Iter end){
    typename Iter::value_type pivot;
    less<typename Iter::value_type> le;
    equal_to<typename Iter::value_type> eq;
    less_equal<typename Iter::value_type> le_q;
    Iter it,jt;

    it=beg;
    advance(it,distance(beg,end)/2);
    if(*it==*beg){
        for(it=beg;it!=end&&*it==*beg;++it);
        if(*it==*beg) return;
    }//endif
    pivot=max(*beg,*it);

    it=beg;
    jt=end;
    --jt;
    while(true){
        while(le(*it,pivot)) ++it;
        while(le_q(pivot,*jt)) --jt;
        if(distance(beg,it)<distance(beg,jt))
             swap(*jt,*it);
        else break;
    }//endwhile
    if(distance(beg,it)>1) qsort(beg,it);
    if(distance(it,end)>1) qsort(it,end);
}//end function
```

Die Sortierfunktion erhält Start- und Endeiterator der Liste und legt zunächst das Pivotelement fest, das in der Regel das erste oder zweite Element der Liste ist. Zwei Hilfsiteratoren `it` und `jt` laufen nun von beiden Enden der Liste aufeinander zu, wobei zwischen den Speicherstellen `beg` und `it` alle Elemente stehen, die kleiner sind als das Pivotelement, zwischen `jt` und `(end-1)` alle Elemente, die größer oder gleich sind. Sofern der Wert, auf den `it` verweist, kleiner ist als das Pivotelement, kann `it` inkrementiert werden, entsprechendes gilt für das Dekrementieren von `jt`. Sind beim Stoppen des In- und Dekrementierens `it` und `jt` noch nicht aneinander vorbei gelaufen, so werden die Elemente vertauscht und damit in die richtigen Teillisten verschoben und anschließend weiter in- bzw. dekrementiert. Andernfalls ist die Aufteilung beendet und es liegen zwischen `beg` und `it` sowie `it` und `end` zwei neue Teillisten vor, die weiter sortiert werden können.

2.7.6 Heapsort-Sortieralgorithmus

Die Vorstufe zu einem weiteren Sortieralgorithmus besteht aus der Erzeugung einer teilsortierten Liste, also einem Heap. In einem Heap besteht die Eltern-Kind-Beziehung (die Indizierung beginnt bei Eins (!!))

```
Elternindex k -> Kinder  2*k  und  2*k+1
```

sowie die Teilordnung

$$L_k \geq L_{2\,k}, L_{2\,k+1}$$

d.h. das Elternfeld ist immer größer als seine beiden Kinder. Zwischen den Kindern gibt es allerdings keine Relation, weshalb es sich hierbei auch nur um eine Teilordnung handelt. Das Feld

Wert	50	25	40	12	13	6	30
Beziehung	1	1.1	1.2	1.1.1	1.1.2	1.2.1	1.2.2

erfüllt beispielsweise die Heap-Definition, besitzt jedoch keine totale Ordnung, da das kleinste Element an der Position 1.2.1 vom größeren Kind des Stamm-Elternteils indiziert wird.

Aus einem Heap kann man nun recht leicht eine sortierte Liste erzeugen, indem man iterativ das erste Element entnimmt und an das Ende einer Liste schreibt, bis der Heap leer ist. Da es sich aufgrund der Sortiervorgänge auf dem Heap immer um das größte oder kleinste noch verbliebene Element handelt, ist die Ergebnisliste wie gewünscht sortiert.

Wie Elemente in einen Heap eingefügt oder aus ihm entfernt werden, haben wir bereits diskutiert. Es bleibt also nur noch, einen Heap aus einer unsortierten Liste zu erzeugen. Eine Heap-Ordnung kann folgendermaßen hergestellt werden:

(a) Beginnend in der Mitte des Feldes wird das größte Element der Speicherstellen k, 2^*k und 2^*k+1 durch Vertauschen an die Stelle k geschoben.
(b) Sofern ein Tausch stattgefunden hat und das Kind, mit dem der Platz getauscht worden ist, weitere Kinder hat, ist die Beziehung zwischen diesem Kind und den dazu gehörenden Enkeln nun möglicherweise nicht mehr korrekt und muss überprüft werden. Die Prüfung ist daher iterativ mit dem Index 2^*k oder 2^*k+1 zu wiederholen, bis keine Kindeskinder mehr vorhanden sind oder kein Tausch stattgefunden hat.
(c) Die Tauschaktion (a) wird bis zum Feldindex 1 nach unten fortgesetzt.

Den Algorithmus implementieren wir in Form einer Klasse, die von einem Container erbt. Es ist zwar auch möglich, ihn wie die anderen Algorithmen als Funktion zu implementieren, jedoch muss man sich dann beim Anfügen und entnehmen von

Elementen selbst um eine Verwaltung der Liste kümmern. Als erste Methode implementieren wir (b), da diese Funktion in verschiedenen Varianten in allen weiteren Teilalgorithmen benötigt wird.

```
template <typename T,
          template <class> class Ctnr=vector >
class Heap: public Ctnr<T> {
private:
   bool hdown(int i,int length){
      typename Ctnr<T>::iterator it_f,it_s1,it_s2;
      bool swapped=true;
      it_f=Ctnr<T>::begin();
      advance(it_f,i-1);
      it_s1=it_f;
      while (swapped && i <= length/2) {
         advance(it_s1,i);
         i<<=1;
         if(i+1 <= length){
            it_s2=it_s1;
            ++it_s2;
            if(*it_s1 < *it_s2){
                ++i;
                it_s1=it_s2;
            }//endif
         }//endif
         if(*it_s1 > *it_f){
            swap<T>(*it_f,*it_s1);
         }else{
            swapped=false;
         }//endif
         it_f=it_s1;
      }//endwhile
      return swapped;
   }//end function
```

Zu berücksichtigen ist, dass der Algorithmus die Listenpositionen von 1 bis *N* numeriert, während die Felder in C/C++ von 0 bis *N*-1 indiziert werden. Im ersten `advance`-Befehl wird auf den Index umgerechnet, während der zweite `advance`-Befehl ohne Korrektur durchgeführt wird, da hier der Abstand zum ersten Kind benötigt wird. Sind zwei Kinder vorhanden, wird zunächst das größere ausgewählt, da ja das größte Element nach vorne zu ziehen ist. Bei einem Tausch dient dieses Kind anschließend als neuer Elternindex. Ähnlich wie beim Bubblesort-Algorithmus kann die Berechnung abgebrochen werden, sofern kein Austausch

der Elemente stattfindet. Wir haben dies mit der Variablen `swapped` berücksichtigt, obwohl ein Abbruch aufgrund der logarithmischen Tiefe sehr viel weniger Auswirkungen hat als beim Bubblesort-Algorithmus.

Die Methoden für die Herstellung der Ordnung sowie das Anfügen und Entnehmen von Elementen unterscheiden sich im Wesentlichen nur in der Art der Aufrufe der `hdown`-Funktion (*siehe oben*):

```
public:
   void make_heap(){
      for(int i=Ctnr<T>::size()/2;i>=1;i--){
         hdown(i,Ctnr<T>::size());
      }//endfor
   }//end function

   void push_heap(T const& val){
      push_back(val);
      for(int i=Ctnr<T>::size()/2;i>=1;i>>=1){
         if(!hdown(i,Ctnr<T>::size())) break;
      }//endfor
   }//end function

   T pop_heap(){
     swap<T>(*Ctnr<T>::begin(),*Ctnr<T>::rbegin());
      T value=Container<T>::back();
      Cntnr<T>::pop_back();
      hdown(1,Ctnr<T>::size());
      return value;
   }//end function
```

Falls Sie sich gewundert haben, dass `hdown()` die Länge der Liste als Parameter erhält und nicht intern die Funktion `size()` verwendet, betrachten Sie die Sortierfunktion für das Erzeugen einer sortierten Liste:

```
void hsort(){
   int i; typename Ctnr<T>::iterator it;
   make_heap();
   for(i=Ctnr<T>::size()-1;i>0;i--){
      it=Ctnr<T>::begin();
      advance(it,i);
      swap<T>(*Ctnr<T>::begin(),*it);
      hdown(1,i);
   }//end for
}//end function
```

Diese Konstruktion ermöglicht es uns, das letzte Element aus dem Sortiervorgang auszublenden, ohne es aus dem Feld löschen zu müssen.

Der Heapsort-Algorithmus besitzt wie der Quicksort-Algorithmus die Komplexität $O(n^* \log(n))$, die für alle Arten der Vorsortierung des Containers gilt (*Quicksort entartet ja im ungünstigsten Fall zu* $O(n^2)$). Die erste Bildung der Heapstruktur ist allerdings etwas aufwändig, so dass der Heapsort-Algorithmus in der Regel langsamer ist als der Quicksort-Algorithmus.

Ein Heap ist aufgrund seiner Struktur mit einem binären Baum vergleichbar, wobei die Besonderheit eines Heaps darin besteht, dass er immer balanciert ist und auch bei Anfügen beliebig gearteter Elemente nicht schief werden kann. Bezahlen muss man die Balance mit der Teilsortierung.

Von allen drei diskutierten Sortieralgorithmen existieren unterschiedliche Varianten (*z.B. Heapstrukturen mit mehr als 2 Kindern*), die in Abhängigkeit von den Elementeigenschaften die Anzahl der Vergleichs- oder Kopieroperationen minimieren.

2.8 Suchen in Strings

2.8.1 Einführende Bemerkungen

Suchen in Strings sind in mehrfacher Hinsicht Sonderfälle. Strings können als Container von Zeichen betrachtet werden. Das Zeichenalphabet umfasst bei sprachdarstellenden Strings europäischer Sprachen etwa 100 Zeichen, kann aber in anderen Schriftfamilien (*chinesisch, alt-ägyptisch*) mehrere hundert bis mehrere tausend Zeichen umfassen oder gar nichts mit einer menschlichen Sprache zu tun haben (*Gen-Sequenzen*). Strings sind unsortiert in unserem Sortiersinn, aber aufgrund einer Vielzahl abstrakter Regeln auch weit von statistischer Zusammensetzung der Zeichen entfernt. Verschiedene Strings, also verschiedene Container, lassen sich aber mittels einer „lexigrafischen Ordnung“ in einem Übercontainer sortiert anordnen (*obwohl dabei der Sinngehalt verloren geht, wenn es sich um Teilstrings eines Satzes handelt, die dort „sortiert“ werden*).

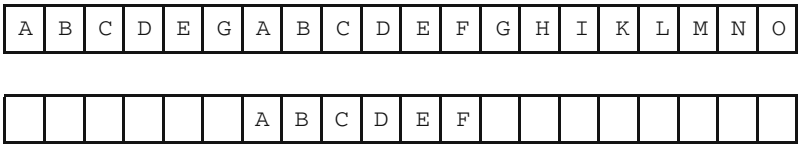
Die Suche in einem String besteht in den meisten Fällen nicht darin, ein bestimmtes Element zu finden, sondern die Position eines Teilstrings – also eines Containers – in einem anderen Container. Um die Verwirrung vollständig zu machen, ist vielfach auch keine exakte Übereinstimmung notwendig, sondern die gefundene Position muss nur eine Zeichenfolge aufweisen, die dem vorgegebenen Suchmuster in irgendeiner Weise ähnlich ist. Was man als „ähnlich“ betrachtet, ist natürlich schon genauer zu definieren.

Wenn man das Thema weiter verfolgt, gelangt man relativ schnell zu den sogenannten „reguläre Ausdrücken “. Hierbei handelt es sich um Zeichenketten, die nur teilweise den zu suchenden String enthalten, meist aber zu einem großen Teil aus Regeln bestehen, welche Zeichen aufeinander folgen oder dies eben nicht tun dürfen, und die ihren Ursprung in der Theorie der formalen Sprachen in der

Informatik haben. Die Regeln, nach denen diese Ausdrücke aufgebaut werden, sind wieder ein Kapitel für sich und nicht in allen Systemen einheitlich. Das Problem besteht darin, dass in einem Ausdruck Zeichen als zu suchende Zeichen und auch als Steuerzeichen für die auszuwertenden Regeln auftreten können.

2.8.2 Naive Suche

Die „naive“ Lösung besteht darin, das erste Zeichen des Suchmusters im zu prüfenden String zu suchen und bei einer Übereinstimmung schrittweise auch die folgenden Zeichen zu vergleichen.



In diesem Beispiel hätte zunächst ein Vergleich der kompletten Musterkette ab dem ersten Zeichen des zu prüfenden Strings stattgefunden, der am Buchstaben „G“ gescheitert wäre. Durch Vorschieben wäre nach einigen Fehlversuchen das zweite Auftreten des Prüfmusters erreicht und anschließend ein kompletter Vergleich positiv durchgeführt worden. Im ungünstigsten Fall liegt der theoretische Aufwand einer solchen Suche bei

$$O(len_{Prüfstring} * len_{Musterstring})$$

weshalb dieses Suchverfahren meist schlechte Bewertungen bekommt.

Ein modifiziertes Problem der Stringsuche besteht darin, String zu finden, der dem Suchmuster „ähnlich“ ist, wobei man natürlich zunächst definieren muss, was man unter „ähnlich“ versteht. Darf beispielsweise eine beliebiges Zeichen abweichen, so kann eine Suche mit dem Vergleich der ersten beiden Zeichen beginnen, die schrittweise über den Prüfstring geschoben werden. Bei Übereinstimmung beider Zeichen ist bis zum Ende oder dem zweiten Fehler zu prüfen, bei einem korrekten Zeichen darf keine weitere Abweichung mehr auftreten, bei einem Doppelfehler kann man ohne weitere Prüfung das Paar um eine Position nach Rechts verschieben.

Weitere Ähnlichkeitsdefinition umfassen Abweichungen nur in bestimmtern Zeichen, Abweichungen in Schreibweisen (*Groß-/Kleinschreibung, d.h. die Zeichen sind vor dem Vergleich einer Normierung zu unterwerfen*), Verdrehern (*„ Felher“ statt „ Fehler“*), Fehlstellen (*„ Fhler“*) oder Verdopplungen (*„ Feehler“*) bzw. zusätzliche Zeichen (*„ Ferhler“*). Das Problem bei Ähnlichkeitssuchen oder auch Austauschaktionen zur Korrektur gefundener Unstimmigkeiten besteht weniger im Aufstellen geeigneter Regeln als vielmehr darin, eine Parametrierung hierfür zu finden, die eine Neuprogrammierung bei Aufstellen einer neuen Regel unnötig macht. Wir kommen an einer anderen Stelle im Zusammenhang mit regulären Ausdrücken darauf zurück.

Man kann die „naive“ Vorgehensweise auch noch durch Rückgriff auf weitere Regeln beschleunigen. In theoretischen Betrachtungen werden meist Strings wie „abaabbabbbaabaaab“ eingesetzt, die aber weder vom Umfang der Zeichensätze noch von der Grammatik her in der Praxis auftreten. Untersucht man Zeichenketten realer Sprachen (*wozu auch DNA–Sequenzen in der Biologie gezählt werden können*), so kommt man bei der Suche nach dem Wort „Auftritt“ in einem Text schnell dahinter, dass die Suche nach „A“ gegenüber einer Suche nach „t“ wesentlich mehr falsche Primärtreffer liefert und bei Auffinden eines „t“ die Prüfung auf ein zweites „t“ im Abstand 3 sinnvoller ist als die Prüfung des nächsten Buchstabens „r“.

Trägt man diesen Eigenschaften durch eine Klassendefinition

```
template <class T, class C=language_traits<T>>
class language_string: public basic_string<T>{ ...
```

mit Spracheigenschaftsklassen `language_traits` ähnlich den Zeicheneigenschaftsklassen `char _traits` Rechnung, so dürfte der Aufwand für das Finden des Musters eher in der Größenordnung

$$O(len_{Prüfstring})$$

liegen. Iteratoren auf dieser Klasse sind sicher keine ganz einfachen Gebilde mehr, da ein Inkrementieren sowohl Vor- als auch Rückwärtsbewegung bedeuten kann und der Iterator auf dem anderen String entsprechend zu synchronisieren ist. Solche Klassen existieren aber zumindest in den Standardbibliotheken nicht.

Aufgabe. Entwerfen Sie einen Algorithmus, bei dem die Reihenfolge der Zeichen des Musterstrings bei der Prüfung festgelegt werden kann.

2.8.3 Boyer-Moore-Algorithmus

Kann man auch unabhängig von einer speziellen Grammatik eine Verbesserung der Suchgeschwindigkeit erreichen? Es zeigt sich, dass ohne weitere Nebenbedingungen größere Teile des Strings bei der Suche übersprungen werden können. Betrachten wir dazu nochmals das Suchbeispiel. Im Gegensatz zur naiven Suche beginnen wir mit dem Vergleich am hinteren Ende des Musterstrings. Wie im naiven Fall bricht der Vergleich ab, sobald die erste Differenz erkannt wird (*oder ist zu Ende, wenn die Strings übereinstimmen*). Dies sei an der Stelle k der Fall, d.h. m-k Zeichen stimmen überein.

Die Anzahl der Positionen, um die der Musterstring weiterbewegt werden kann, kann auf zwei Arten ermittelt werden:

(a) Das Muster der Zeichen (k+1, ... m) stimmt in beiden Strings überein. Eine vollständige Übereinstimmung von Muster- und Teststring kann also frühestens dann wieder auftreten, wenn durch den Vorschub des Musterstrings die Zeichenfolge ein weiteres Mal mit der gefundenen Position im Teststring koinzidiert. Der Musterstring kann also bis zum erneuten Auftreten des gefundenen

Teilmusters vorgeschoben werden, d.h. falls das Muster nicht erneut auftritt, bis hinter die geprüfte Stelle.

(b) Das Zeichen im Teststring an der (*relativen*) Stelle k muss für eine vollständige Übereinstimmung auch im Musterstring vorhanden sein. Der Musterstring kann also so weit verschoben werden, bis er an der durch k indizierten Teststringstelle mit diesem übereinstimmt; falls das Zeichen im Musterstring nicht auftritt, kann dieser bis hinter die Stelle k verschoben werden.

Je nach Prüfergebnis kann der Musterstring im Extremfall also bis zur eigenen Länge vor der nächsten Prüfung verschoben werden, d.h. die Laufzeitordnung des Verfahrens bewegt sich im Bereich $O(n/m)..O(n)$. Eine optimale Strategie würde das Maximum der jeweiligen Vorschübe aus (a) und (b) verwenden. Es stellt sich allerdings die Frage nach den Kosten für die beiden Prüfungen.

Zu (a). Durch eine Vorverarbeitung des Musterstrings lässt sich die Vorschubberechnung sehr effektiv gestalten. Dazu notiert man in einer Tabelle, um wieviele Zeichen der Musterstring zu veschieben ist, wenn die erste Abweichung zum Teststring – von hinten gezählt – an der Stelle k auftritt. Diese Analyse ist recht einfach zu implementieren. Betrachten wir dazu den Musterstring

```
muster="abcdbbcbb cebcccbgahabc";
```

- Wenn bereits das erste Zeichen abweicht, ist der Musterstring um eine Einheit zu verschieben. Der erste Tabelleneintrag ist eine Eins.
- Es wird nun von hinten nach vorne überprüft, an welchen Stellen das letzte Zeichen im Musterstring erneut auftritt. Das erste Auftreten wird ebenfalls in der Tabelle notiert.
- Bei jeder gefundenen Übereinstimmung wird nun iterativ überprüft, ob dies auch für das Zeichen davor gilt. Ist dies nicht der Fall, wird die Iteration abgebrochen, wird die Iterationstiefe erstmalig erreicht, wird auch diese Position in der Tabelle notiert; bei Übereinstimmung wird die Iteration auf der nächsten Stufe fortgesetzt, wobei darauf zu auchten ist, dass die Iterationstiefe = Länge des Untermusters nicht größer wird als die Entfernung der gefundenen Position vom Ende und der Beginn des Musterstrings nicht überschritten wird.
- Abschließend werden die noch nicht besetzten Tabellenpositionen so belegt, dass jeweils der Rest des Musterstrings weitergeschoben wird.

Diese Überlegungen führen zu der folgenden Implementation:

```
int i,j; char const* ch;
vector<int> move;
move.push_back(1);
ch=&muster.at(muster.size()-1);
for(i=2;i<muster.size();i++){
   if(muster.at(muster.size()-i)==*ch){
      if(move.size()<2) move.push_back(i-2);
```

```
      for(j=1;j<i && j<=muster.size()-i;j++){
         if(muster.at(muster.size()-i-j)!=*(ch-j))
            break;
         if(move.size()<j+2) move.push_back(i-j-2);
      }//endfor
   }//endif
}//endfor
for(j=move.size();j<muster.size();j++)
   move.push_back(muster.size()-j);
```

Aufgabe. Diese Implementation ist eigentlich zu aufwändig. Die äußere Schleife kann auch vorzeitig beendet werden, wenn klar ist, dass keine weiteren Eintragungen in die Tabelle zu erwarten sind. Ergänzen Sie den Code entsprechend.

Für das Beispiel liefert diese Auswertung die Vorschübe

```
(1,6,17,19,18,...2,1)
```

Die Mustersuche beginnt nun durch Vergleich der Zeichen von Test- und Musterstring von Hinten. Wird eine Abweichung festgestellt, so gibt die Tabelle Auskunft, wie der Offset des Musterstrings über dem Teststring zu erhöhen ist.

```
ofs=0;
while(muster.size()+ofs<=s.size()){
   for(i=1;i<=muster.size();i++){
      if(s.at(ofs+muster.size()-i)!=
         muster.at(muster.size()-i))
         goto difference;
   }//endfor
   return ofs;
   difference:
      ofs+=move[i-1];
}//endwhile
return -1;
```

Aufgabe. Tabelle und Offset sind für fortgesetzte Suchen verwendbar. Integrieren Sie alles in eine Klasse.

Zu (b). Die zweite Strategie lässt sich ebenfalls durch Tabellen realisieren, allerdings nicht so glatt wie in Fall (a). Betrachten wir dazu die Strings

```
muster =   „acbcabc"
test   =  „...acabc..acaac.."
```

Die Differenz tritt bei der ersten Teilkoinzidenz (*hinterer Teil des Teststrings*) bereits beim zweiten Zeichen auf, bei der zweiten aber erst beim fünften. Bei beiden

ist das falsche Zeichen im Teststring ein „a“. Der Vorschub des Musterstrings hat auf das nächste „a“ zu erfolgen – in den beiden Fällen auf verschiedene Positionen. Die Tabelle muss also alle Positionen enthalten, die ein bestimmtes Zeichen im Musterstring einnimmt. Nun macht es auch wenig Sinn, eine mehrdimensionale Tabelle zu erstellen, denn wenn es sich um andere Zeichen als ASCII-Zeichen handelt, wird die Tabelle unhandlich groß, ist vermutlich nur schwach besetzt, da ein String in der Regel deutlich weniger Zeichen enthält als der Zeichenvorrat, und die Dimensionen sind nicht alle in der gleichen Tiefe belegt (*„ e“ ist häufiger als „z“ und wird mit entsprechend mehr Positionseinträgen in der Tabelle vorhanden sein*). Eine Tabelle mit den passenden Eigenschaften ist die `multimap`:

```
multimap<char,int> mp;
..
mp.clear();
for(i=0;i<muster.size();i++){
    pm.insert(pair<char,int>(
              muster.at(muster.size()-1-i),i);
}//endfor
ofs=0;
```

Bei einer Nichtübereinstimmung muss in dieser Tabelle nur nach einem Eintrag gesucht werden, dessen Abstand vom Ende größer als die aktuelle Position ist.

```
multimap<char,int>::iterator it;
while(muster.size()+ofs<=s.size()){
   for(i=1;i<=muster.size();i++){
      if((ch=s.at(ofs+muster.size()-i))!=
           muster.at(muster.size()-i))
         goto difference;
   }//endfor
   return ofs;
   difference:
      it=mp.find(s.at(ofs+muster.size()-i));
      while(it!=mp.end() && it->second<=i &&
            it->first==ch) it++;
      if(it==mp.end())
         ofs+=muster.size()-i;
      else
         ofs+=it->second-i;
}//endwhile
ofs=-1;
```

Bewertung. Welcher Algorithmus ist vorzuziehen? Die Antwort hängt wie bereits beim naiven Algorithmus vom Einsatzgebiet ab. Der Vergleich von Zeichenfolgen kann mit Hilfe von Assemblerbefehlen recht schnell erfolgen. Bei relativ kurzen und ständig wechselnden Strings frisst der Aufwand für die Initialisierung die schnellere Positionierung des Teststrings schnell auf. Erst ab einer gewissen Länge oder wiederholten Suchen mit gleichen Strings kommen die Vorteile der schnelleren Suche zum Tragen.

Der Aufwand bei Algorithmus (b) ist aufgrund der verwendeten Tabelle grundsätzlich höher als bei Algorithmus (a). Trotzdem kann er vorteilhaft sein, etwa wenn der Teststring viele Zeichen enthält, die im Muster selten oder gar nicht auftreten, oder der Musterstring aus vielen gleichartigen Wiederholungen besteht.

2.8.4 Suffix-Bäume

2.8.4.1 Der Aufbau eines Suffixbaumes

Sofern (längere) Strings mehrfach nach verschiedenen Kriterien untersucht werden sollen, stellt sich die Frage, ob man durch einen erhöhten Aufwand bei der Initialisierung nicht die anschließenden Auswertungen beschleunigen kann. Eine derartige Möglichkeit bietet die Zerlegung eines Strings in einen Suffixbaum. Dieser entsteht durch rekursives Einfügen des Strings in den Baum und anschließendes Streichen des ersten Zeichens, bis der ursprüngliche String komplett geleert ist.

Erklären wir dies zunächst an einem einfachen Beispiel. Gegeben sei der String `sasr`. Der Baum besteht zunächst nur aus der Wurzel, die auf diesen String zeigt:

```
.-sasr
```

Nach Streichen des ersten Buchstabens wird nun der verbleibende String eingefügt, wobei geprüft wird, wie weit er mit dem ersten String den gleichen Beginn hat. Da die ersten Zeichen unterschiedlich sind, zeigt die Wurzel nun auf zwei Strings:

```
.-+-asr
    |
    +-sasr
```

Im nächsten Schritt stellt man nun fest, dass gleiche Anfangsbuchstaben vorliegen. Der String `sasr` wird nun an der ersten Postion, an der die Strings nicht mehr übereinstimmen, selbst zum Knoten, und der Baum nimmt folgende Gestalt an:

```
.-+-asr
  |
  +-s-+-asr
      |
      +-r
```

Im letzten Schritt wird nun der verbleibende Buchstabe eingefügt. Zusätzlich wird am Ende eines jeden Zweiges die Position angegeben, an der dieser Teilstring beginnt, so dass der Gesamtbaum nun folgendes Aussehen annimmt.

```
.-+-asr         (1)
  |
  +-r           (3)
  |
  +-s-+-asr     (0)
      |
      +-r     (2)
```

Die Vorbereitung des Strings verursacht natürlich einigen Zeitaufwand und einen Mehrbedarf an Speicherplatz für die Baumstruktur, eine Analyse wird jedoch nun recht einfach: ausgehend von der Wurzel prüft man, mit welchen Zweigen der Teststring übereinstimmt, und erhält sämtliche Positionen, an der er zu finden ist, wenn man die Längen aller Zweige im verbleibenden Restbaum einsammelt. Auch Ähnlichkeitsanalysen werden so recht einfach durchführbar.

2.8.4.2 Die Erzeugung eines Suffixbaumes

Für die Realisierung eines Suffixbaumes speichern wir den String einmalig und führen in jedem Knoten Zeiger auf den Teilbereich des Strings, der in diesem Knoten abgebildet wird. Die Liste der Unterknoten organisieren wir des schnelleren Zugriffs beim Suchen halber ebenfalls als Baum.

```
class SuffixTree {
....
protected:
   struct Node {
      char const* beg, *end;
      set<Node> st;
   };
   set<Node> root;
   string s;
};//end class
```

Aufgabe. Konstruieren Sie auf dem Papier die Knoten für das oben Beispiel `sasr`. Stellen Sie sicher, dass Sie das Konzept verstanden haben.

Da `Node` Parameter eines Binärbaumes ist, müssen Gleichheits- und Kleineroperator implementiert werden. Die Operatoren beziehen sich ausschließlich auf die Inhalte der durch die Zeiger indizierten Teilstring; die Zeigerwerte selbst oder die

Inhalte der Knotenlisten dürfen nicht berücksichtigt werden. Zwei Knoten sind logisch gleich, wenn sie identische Teilstrings indizieren. Die Kleinerrelation muss außer dem Inhalt auch noch die Länge der Strings berücksichtigen. Wir erhalten so unter Verwendung von Funktionen der C-Bibliothek

```
struct Node {
...
   bool operator==(Node const& n) const {
      return distance(beg,end)==distance(n.beg,n.end) &&
             memcmp(beg,n.beg,distance(beg,end))==0;
   }
   bool operator<(Node const& n) const {
      int c=memcmp(beg,n.beg,
           min(distance(beg,end),distance(n.beg,n.end)));
      return c<0 ||
       (c==0 && distance(beg,end)<distance(n.beg,n.end));
   }
```

Wie wir in der Einführung gesehen haben, kann es im Laufe der Generierung des Baumes notwendig sein, einen Teilstring weiter aufzusplitten. Hierzu definieren wir zusätzlich noch eine Methode, die die Position des ersten Unterschiedes zwischen den indizierten Strings ermittelt

```
int first_difference(Node const& n) const{
    int len=min(distance(beg,end),distance(n.beg,n.end));
    for(int i=0;i<len;i++)
       if(*(beg+i)!=*(n.beg+i))
          return i;
    return len;
  }//end function
```

Die Erzeugung des Baumes führen wir iterativ und rekursiv durch. Die Iteration übernimmt die schrittweise Abspaltung der Zeichen am Stringanfang:

```
void SuffixTree::set_string(string t){
    root.clear();
    s=t;
    for(int i=0;i<s.length();i++){
        Node n(s.c_str()+i,s.c_str()+s.length());
        loader(root,n);
    }
}//end  function
```

Die Methode `loader` führt nun das Einfügen des Teilstrings in den Baum aus, wobei bestehende Knoten ggf. weiter zerlegt werden. Um in den Knotenlisten nach vorhandenen Einträgen zu suchen, verwenden wir allerdings nicht die Methode `find`, sondern `lower_bound` (siehe weiter unten), die einen Iterator auf einen identischen oder den nächstkleineren Knoten liefert.

```
void SuffixTree::loader(set<Node>& sn, Node& n){
    set<Node>::const_iterator it;
    it=sn.lower_bound(n);
```

Liefert diese Methode den Enditerator zurück, so ist der einzufügende Teilstring zwar lexikalisch größer als alle bislang vorhandenen, kann aber auch zu einer Aufspaltung eines bereits vorhandenen Teilstrings Anlass geben. Sofern möglich, müssen wir in diesem Fall den Iterator repositionieren (im anderen Fall wird der neue String in die Unterknotenliste eingefügt):

```
if(it==sn.end()){
        if(sn.end()==sn.begin()){
            sn.insert(n);
            return;
        }else{
            it--;
        }//endif
}//endif
```

Wir können nun im nächsten Schritt untersuchen, wie weit der vorhanden Teilstring mit dem einzufügenden übereinstimmt. Besteht vollständige Übereinstimmung, aber der einzufügende String ist länger, so kann er um den aktuellen Knotenteilstring verkleinert und (rekursiv) in die Liste des aktuellen Knotens eingefügt werden.

```
Node* nd =const_cast<Node*>(&(*it));
    int dpos=nd->first_difference(n);
    if(dpos==distance(nd->beg,nd->end)){
        n.beg+=distance(nd->beg,nd->end);
        loader(nd->st,n);
        return;
}
```

Dies ist nicht ganz so einfach, wie es aussieht, da der als Listenstruktur gewählte Baumcontainer nur konstante Iteratoren besitzt. Durch einen `const _cast` umgehen wir das Problem. Da in dieser Operation nur die Knotenliste verändert wird, die bei den Ordnungsrelationen ohnehin keine Rolle spielt, ist dies nicht weiter problematisch.

Ist gar keine Übereinstimmung vorhanden, kann wie bei der Auslieferung des Endeiterators aber noch eine lexikalische Übereinstimmung mit dem davor liegenden Knoten bestehen. Wir Repositionieren den Iterator daher nochmals, falls möglich, oder fügen ein:

```
if(dpos==0){
        if(it==sn.begin()){
            sn.insert(n);
            return;
        }else{
            it--;
            nd =const_cast<Node*>(&(*it));
            dpos=nd->first_difference(n);
            if(dpos==0){
                sn.insert(n);
                return;
            }//endif
        }//endif
}//endif
```

Im verbleibenden Fall haben wir eine Teilübereinstimmung. Der Knoten im Baum muss daher in zwei Knoten zerlegt werden, wobei der zweite den hinteren Stringteil sowie die alte Knotenliste übernimmt und seinerseits Unterknoten des ersten, den gemeinsamen Teil enthaltenden alten Knotens übernimmt. Der einzufügende Knoten wird ebenfalls in abgespeckter Form in die Unterknotenliste eingefügt.

```
   Node hnode=*nd;
   hnode.beg=hnode.beg+dpos;
   n.beg+=dpos;
   nd->end=nd->beg+dpos;
   nd->st.clear();
   nd->st.insert(hnode);
   nd->st.insert(n);
}//end function
```

Aufgabe. Diese Operation ist nicht ganz so unkritisch wie die erste, da hier auch auf den Schlüsselteil eines Knotens in einem binären Baum zugegriffen wird. Nicht umsonst handelt es sich hierbei um eine normalerweise nicht zulässige Operation. Vergewissern Sie sich daher, unter welchen Bedingungen in einer lexikalisch geordneten Liste von Strings die Ordnung erhalten bleibt, wenn ein Teilstring verkürzt wird, und ob diese Bedingungen hier tatsächlich eingehalten werden !

2.8.4.3 Positionen beliebiger Teststrings finden

Ein beliebiger Teststring kann potentiell an beliebig vielen Positionen zu finden sein, so dass wir als Ergebnis der Suche eine Liste generieren müssen. Auch dies machen wir rekursiv.

```
set<int> SuffixTree::find(string t) const{
    Node n(t.c_str(),t.c_str()+t.length());
    set<int> pos_set;
    do_find(pos_set,root,n);
    return pos_set;
}
```

Die Rekursionsfunktion ist ähnlich der Generierungsmethode aufgebaut. Auch hier verwenden wir eine Ähnlichkeitssuche mit Repositionierung:

```
void SuffixTree::do_find(set<int>& pset,
                         set<Node> const& nset,
                         Node& node) const{
    set<Node>::const_iterator it=nset.lower_bound(node);
    if(it==nset.end()){
        if(nset.end()==nset.begin()) return;
        it--;
    }//endif
    int dpos=it->first_difference(node);
    if(dpos==0){
        if(it==nset.begin()) return;
        it--;
        dpos=it->first_difference(node);
    }//endif
```

Sind die Inhalte gleich, müssen nur noch die Positionen eingesammelt werden, desgleichen, wenn der Teststring nur ein Teilstring des aktuellen Knotens ist. Ist der aktuelle Knoten vollständig im längeren Teststring enthalten, wird in der Unterknotenliste fortgesetzt. Besteht keine vollständige Übereinstimmung, wird abgebrochen.

```
if(*it==node){
        list_positions(pset,it->st,
                       distance(it->beg,it->end));
        return;
    }//endif
    if(dpos==distance(it->beg,it->end)){
        node.beg+=distance(it->beg,it->end);
```

```
        do_find(pset,it->st,node);
        return;
    }//endif

    if(dpos==distance(node.beg,node.end)){
        list_positions(pset,it->st,
                       distance(it->beg,it->end));
    }
}
```

Die Positionsliste erhält man, wenn man zu allen erreichbaren Blättern fortschreitet und dabei die Teilstringlängen akkumuliert. Nach Abzug von der Gesamtstringlänge bleibt die Position übrig.

```
void SuffixTree::list_positions(set<int>& pset,
                                set<Node> const& nset,
                                int len) const{
   if(nset.empty()){
      pset.insert(s.length()-len);
   }else{
      for(set<Node>::const_iterator it=nset.begin();
         it!=nset.end();it++)
         list_positions(pset,it->st,
                        len+distance(it->beg,it->end));
   }//endif
}
```

Aufgabe. Führen Sie Laufzeitmessungen an den verschiedenen Algorithmen mit sehr langen Strings durch (z.B. durch Übernahme von Buchinhalten o.Ä. in einen String; mit Laufzeitermittlungen befasst sich das nächste Hauptkapitel). Ermitteln Sie Randbedingungen für den Einsatz der Algorithmen.

2.9 Algorithmen der STL

Wie wir bereits mehrfach gezeigt haben, ist es mit Hilfe von Iteratoren relativ einfach möglich, bestimmte Operationen (*Algorithmen*) auf den Elementen eines Containers oder den Elementen verschiedener Container ausführen zu lassen, ohne dass man sich über die Art der Container Gedanken machen müsste (*von Containern mit Iteratoren des Typs* `pair<T1,T2>` *einmal abgesehen*). Beispielsweise lässt sich das Skalarproduktes mit einer spezialisierten Funktion folgendermaßen berechnen:

```
template <class it1, class it2, class T>
T skalarprodukt(it1 beg, it1 end, it2 sec){
    T sum=0;
    while(beg!=end){
```

```
        sum+=(*beg * *sec);
        ++beg; ++sec;
    }//endwhile
    return sum;
}//end function
```

Klar ist, dass die durch die Iteratoren `(beg,end)` und `(sec)` repräsentierten Container gleich groß sein (*zumindest darf der zweite Container nicht kleiner sein*) und kompatible Datentypen aufweisen müssen. Die Iteratoren können aber unterschiedlichen Containertypen entstammen.

2.9.1 Grunddesign der Algorithmen

Die STL stellt eine Vielzahl von fertigen Algorithmen zur Verfügung. Dabei kommt das folgende Konstruktionsprinzip zur Anwendung:

```
// Berechnung von Einzeldaten aus den
// Containerwerten
template <class InpIt, class T>
T  _algorithm(InpIt beg, InpIt end) { .. }
template <class InpIt_1, class InpIt_2, class T>
T  _algorithm(InpIt_1 beg1, InpIt_1 end,
              InpIt_2 beg2) { .. }
...
// Berechnung neuer Daten im Container
template <class InpIt, class OutIt>
void _algorithm(InpIt begi, InpIt end, OutIt bego) { .. }
template <class InpIt_1, class InpIt_2,
          class OutIt>
void _algorithm(InpIt_1 beg1, InpIt_1 end,
                InpIt_2 beg2, OutIt bego) { .. }
...
```

Der Iterator des ersten Containers wird jeweils mit Start und Ende übergeben, alle anderen Container übergeben nur ihren Startiterator. Bei Operationen auf Containern wird für die Ausgabe ein separater Startiterator angegeben. Dieser darf mit einem der Eingabeiteratoren übereinstimmen, wenn für das Ergebnis kein neuer Container verwendet werden soll.

Intern kann die Übergabe eines Eingabeiterators als gleichzeitiger Ausgabeiterator erhebliche Auswirkungen auf den Rechengang haben, wenn vorzeitiges Überschreiben von später im Algorithmus noch benötigten Werten vermieden werden muss (*siehe Kap. 3*). In der STL ist dies berücksichtigt; bei der Konstruktion eigener Algorithmen muss darauf geachtet werden, dies nicht zu übersehen. Wenn

im ersten Anlauf nicht alles implementiert wird (*ich habe ja oben selbst darauf hingewiesen, den Arbeitsaufwand zunächst auf das Notwendige zu deckeln*), sollte zumindest ein Kommentar höflich darauf hinweisen, dass der Algorithmus mit einer bestimmten Verarbeitungsart noch nicht klar kommt, anstatt das kommentarlos Datenunfug ausgegeben wird.

Wir sehen uns nun einige der von der STL zur Verfügung gestellten Algorithmen an. Der Algorithmus

```
template<class InIt, class Fun>
   Fun for_each(InIt first, InIt last, Fun f);
```

führt auf allen Elementen des Containers die Funktion/das Objekt `void f(*InIt)` aus und gibt anschließend `f` als Rückgabewert aus. Gemäß STL–Konvention sollten dabei die Werte der Containerelemente unverändert bleiben.[32]

Falls das jetzt etwas noch verwirrend auf Sie wirkt (*es handelt sich hier um eine Verallgemeinerung des Relationenbegriffs aus dem letzten Kapitel*), schauen Sie sich die folgenden Beispiele an:

```
void Print(T& t){
     cout << t << endl;
}//end function
...
for_each(v.begin(),v.end(),Print);
```

druckt mittels der Funktion `Print` alle Elemente des Vektors `v` aus. `f` ist in diesem Fall eine Funktion des angegebenen Typs, und die Rückgabe ihrer Adresse von `for _each` ist in der Regel für das weitere Geschehen uninteressant.

Ist der hinter den Iteratoren stehenden Conatiner nun beispielsweise ein echter Vektor, dessen Betrag ermittelt werden soll, kann das folgendermaßen implementiert werden:

```
struct Betrag {
    double ssum;
    Betrag(){ssum=0;};
    inline void operator()(double& d){
        ssum+=(d*d);
    };//end operator
};//end struct
...
r=sqrt(for_each(v.begin(),v.end(),Betrag()).ssum);
```

Übergeben wird an `for _each` nun keine Funktion, sondern ein Objekt, das vermöge von `void operator()(..)` die gleiche Aufruffunktionalität herstellt

[32] Wer dennoch etwas ändert, kann sich zwar freuen, wenn das herauskommt, was er sich erhofft hat, darf sich aber nicht beschweren, wenn der STL-Programmierer die Sache etwas anders angefasst hat und es nicht klappt.

wie die `Print`-Funktion und das nach Rückgabe durch `for _each` zu weiteren Aktionen herangezogen werden kann.

Für Änderungen von Containerelementen sind die Methoden

```
template<class InIt, class OutIt, class Unop>
OutIt transform(InIt first, InIt last, OutIt x,
                Unop uop);
template<class InIt1, class InIt2, class OutIt,
         class Binop>
OutIt transform(InIt1 first1, InIt1 last1,
                InIt2 first2, OutIt x, Binop bop);
```

mit

```
*x = uop(*first)
```

beziehungsweise

```
*x = bop(*first1, *first2)
```

zuständig. Die unären beziehungsweise binären Methoden können wie oben als normale Funktionen oder Klassen programmiert und als Vorlagenparameter angegeben werden. Da in diesem Konzept einige Möglichkeiten stecken, die nicht in ein paar Zeilen zu beschreiben sind, widmen wir ihm ein weiteres Kapitel im Anschluss an die Vorstellung der STL-Algorithmen und verzichten hier auf weitere Beispiele.

2.9.2 Suchalgorithmen für einzelne Elemente[33]

Suchen nach einzelnen Elementen in einem Container. Gesucht wird das erste (*oder letzte*) Element, das einen bestimmten Wert aufweist, eine bestimmte (*unäre*) Relation erfüllt, in einem vorgegebenen zweiten Container enthalten oder nicht enthalten ist beziehungsweise mit einem/keinem Element dieses Containers eine bestimmte (*binäre*) Relation erfüllt oder den gleichen Wert wie sein Nachbarelement aufweist beziehungsweise mit diesem eine Relation erfüllt.[34] Der Rückgabewert ist ein Iterator auf das betreffender Element des ersten Containers.

Die ersten beiden Algorithmen suchen nach dem ersten Element in einem Container, das einen bestimmten Wert aufweist oder eine bestimmte Relation erfüllt. Eine Relation ist wieder eine Funktion oder ein Objekt im bei der Funktion

---

[33] Die Zeile template<class ...> sparen wir im folgenden aus Platzgründen ein.

[34] Das war ein langer komplizierter Satz, und ich werde mich, hoffentlich erfolgreich, bemühen, das nicht häufiger zu machen. Ich möchte jedoch die Liste der Bibliotheksalgorithmen möglichst kompakt abhandeln, da Sie sich der Algorithmen zwar im Bedarfsfall bedienen, diese aber nicht selbst implementieren sollen.

for_each besprochenen Sinn; die von der STL bereitgestellten Standardrelationen werden weiter hinten ausführlich diskutiert.

```
InIt   find (InIt first, InIt last,  const T& val);
InIt  find_if  (InIt first, InIt last, Pred pr);
```

Die folgenden Algorithmen suchen nach Teilübereinstimmungen, das heißt zumindest einige Elemente der zweiten Folge finden sich in der ersten wieder. Der Rückgabeiterator verweist auf den Beginn des Auftretens der Prüffolge. Findet sich beispielsweise eine Übereinstimmung zwischen Position 20 der zu testenden Folge und der Position 5 der Prüffolge, so weist der Rückgabeiterator auf die Position 15 in der zu testenden Folge, obwohl die Übereinstimmung weiter hinten liegt.

```
FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
                     FwdIt2 first2, FwdIt2 last2);
FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
                     FwdIt2 first2, FwdIt2 last2,
                     Pred pr);
FwdIt1 find_first_not_of(FwdIt1 first1,
                     FwdIt1 last1,
                     FwdIt2 first2, FwdIt2 last2);
FwdIt1 find_first_not_of(FwdIt1 first1,
                     FwdIt1 last1,
                     FwdIt2 first2, FwdIt2 last2,
                     Pred pr);
```

Diese Algorithmen erlauben Ähnlichkeitssuchen, beispielsweise zum Aufspüren von Schreibfehlern bei einzelnen Zeichen.

2.9.3 Suchen nach mehrfach auftretenden Elementen

Zum Auffinden benachbarter gleicher Elemente, also beispielsweise Schreibfehler durch Zeichenverdopplungen, dienen die Algorithmen

```
FwdIt  adjacent_find(FwdIt first, FwdIt last);
FwdIt  adjacent_find(FwdIt first, FwdIt last,
                                      Pred pr);
```

Die Anzahl der benachbarten gleichen Elemente kann in den folgenden Algorithmen exakt vorgegeben werden.

```
FwdIt  search_n(FwdIt first, FwdIt last,
                Dist n, const T& val);
FwdIt  search_n(FwdIt first, FwdIt last,
                Dist n, const T& val, Pred pr);
```

2.9.4 Vollständige Übereinstimmung

Die Algorithmen haben ähnliche Aufgaben wie die Suchfunktionen nach einzelnen Elementen, jedoch müssen nun Zeichenketten (*im allgemeinen Sinn, also Ketten beliebiger Objekte*) im ersten und zweiten Container vollständig übereinstimmten beziehungsweise elementweise die Relation erfüllen.

```
FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
                      FwdIt2 first2, FwdIt2 last2);
FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
                      FwdIt2 first2, FwdIt2 last2,
                      Pred pr);
```

Während `seach` das erste Auftreten eines Teilstrings ermittelt, suchen die folgenden Funktionen das letzte Auftreten.

```
FwdIt1 find_end (FwdIt1 first1, FwdIt1 last1,
                      FwdIt2 first2, FwdIt2 last2);
FwdIt1 find_end (FwdIt1 first1, FwdIt1 last1,
                      FwdIt2 first2, FwdIt2 last2,
                      Pred pr);
```

2.9.5 Binärsuche

Auf sortierten Containern sind schnellere Suchverfahren möglich. Da einige Containertypen sowohl sortierte als auch unsortierte Instanzen zulassen, der Zustand aber nicht unbedingt leicht sichtbar ist, sollte der Nutzer Kontrollmechanismen vorsehen.

```
bool binary_search(FwdIt first, FwdIt last,
                   const T& val);
bool binary_search(FwdIt first, FwdIt last,
                   const T& val, Pred pr);
```

2.9.6 Anzahlen bestimmter Elemente

Feststellen der Anzahlen von Elementen mit bestimmten Werten oder erfüllten unären Relationen

```
size_t count(InIt first, InIt last, const T& val,
             Dist& n);
size_t count_if(InIt first, InIt last, Pred pr);
```

2.9.7 Unterschiede und Ähnlichkeiten

Der erste Algorithmus findet das erste unterschiedliche Zeichen in beiden Containern. Der Rückgabewert enthält Iteratoren auf die entsprechenden Positionen in beiden Containern und ist deshalb vom Typ `pair<..>`. Alternativ können auch zwei Container auf den gleichen Inhalt überprüft werden. Der Rückgabewert ist in diesem Fall `true` oder `false`

```
pair<InIt1, InIt2> mismatch(InIt1 first,
              InIt1 last, InIt2 x);
pair<InIt1, InIt2> mismatch(InIt1 first,
              InIt1 last, InIt2 x, Pred pr);
bool equal(InIt1 first, InIt1 last, InIt2 x);
bool equal(InIt1 first, InIt1 last,
              InIt2 x, Pred pr);
```

Die Identitäsrelationen können wieder durch andere Relationen abgelöst werden. Die Container müssen von gleicher Größe sein.

2.9.8 Enthaltensein von Elementen

Die folgenden Algorithmen prüfen, ob alle Elemente der zweiten Iteratorfolge auch in der ersten Iteratorfolge auftreten. Die ersten beiden Algorithmen setzen voraus, dass die Elemente der Größe nach sortiert sind, die beiden folgenden Algorithmen sind nicht von einer Sortierung abhängig.

```
bool includes(InIt1 first1, InIt1 last1,
            InIt2 first2, InIt2 last2);
bool includes(InIt1 first1, InIt1 last1,
            InIt2 first2, InIt2 last2, Pred pr);
bool lexicographical_compare(InIt1 first1,
              InIt1 last1,
              InIt2 first2, InIt2 last2);
bool lexicographical_compare(InIt1 first1,
              InIt1 last1,
              InIt2 first2, InIt2 last2, Pred pr);
```

Beispiel. Die Strings `"abcdtgh"` und `"tbad"` liefern mit dem ersten Algorithmus den Rückgabewert `false`, mit dem dritten `true`. Werden die Strings in die

sortierten Reihenfolgen `"abcdght"` und `"abdt"` gebracht, erkennt auch der erste Algorithmus die Relation.

2.9.9 Kopieren von Containern

Kopieroperationen und elementweise Austauschoperationen zwischen Containern. Die Methoden sind nicht überlappungssicher . Zum Kopieren in überlappenden Sequenzen muss der Anwender selbst die kompatible Kopieroperation aussuchen. Der erste Algorithmus kopiert den Inhalt des ersten Containers in den zweiten, der eine entsprechende Größe besitzen muss, und liefert einen Iterator auf das erste Zeichen hinter der kopierten Sequenz zurück.

```
OutIt   copy(InIt first, InIt last, OutIt x);
```

Der zweite Algorithmus kopiert den Inhalt des ersten Containers vom Ende her in den zweiten, das heißt der Iterator `x` muss mindestens einen Abstand zum Beginn des Containers aufweisen, der der Größe des ersten Containers entspricht. Der dritte Algorithmus tauscht die Elemente zwischen den Containern aus.

```
BidIt2 copy_backward(BidIt1 first, BidIt1 last,
                     BidIt2 x);

Beispiel:  C1:  "x1xx2x"
           C2:  "yyyyyyyyy", x=C2.last
           Re:  "yyyx1xx2x"

FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last,
                             FwdIt2 x);
```

Bei Verwendung von Rückwärtsiteratoren lässt sich die Reihenfolge der Elemente im Zielcontainer invertieren.

2.9.10 Austauschen von Elementen

Austauschoperationen von einzelnen Elementen in Containern, wobei zwischen einem direkten Austausch im vorgegebenen Container und einer Kopie unter Austausch unterschieden wird. Anstelle eines Austauschs, der eine Untersuchung der vorhandenen Elemente und eine Ausführung der Austauschanweisung nur bei Erfüllen einer Relation voraussetzt, ist auch ein Füllen eines bestimmten Bereiches mit einer Konstanten oder einem generierten Wert (*beispielsweise einer Zufallzahl, Funktionstyp* `*OutIt g()`) möglich. Die Funktionen sollten auch ohne weitere Erläuterung verständlich sein.

```
void replace(FwdIt first, FwdIt last,
             const T& vold, const T& vnew);
```

```
void replace_if(FwdIt first, FwdIt last,
            Pred pr, const T& val);

OutIt replace_copy(InIt first, InIt last, OutIt x,
            const T& vold, const T& vnew);
OutIt replace_copy_if(InIt first, InIt last,
             OutIt x, Pred pr, const T& val);

void fill(FwdIt first, FwdIt last, const T& x);
void fill_n(OutIt first, Size n, const T& x);

void generate(FwdIt first, FwdIt last, Gen g);
void generate_n(OutIt first, Dist n, Gen g);
```

2.9.11 Löschen von Elementen

Löschen von Elementen mit bestimmten Werten oder von Mehrfacheinträgen in einem Container. Auch sind weitere Kommentare wohl nicht notwendig.

```
FwdIt remove(FwdIt first, FwdIt last,
             const T& val);
FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
OutIt remove_copy(InIt first, InIt last, OutIt x,
             const T&val);
OutIt remove_copy_if(InIt first, InIt last,
             OutIt x, Pred pr);

FwdIt unique(FwdIt first, FwdIt last);
FwdIt unique(FwdIt first, FwdIt last, Pred pr);
OutIt unique_copy(InIt first, InIt last, OutIt x);
OutIt unique_copy(InIt first, InIt last, OutIt x,
                  Pred pr);
```

2.9.12 Reihenfolgeänderungen

Sortierfunktionen beziehungsweise Funktionen zur Veränderung der Reihenfolge. Die erste Funktionengruppe kehrt die Reihenfolge der Elemente eines Containers um beziehungsweise führt ein zyklisches Linksschieben durch, bei der das durch den Iterator `middle` gekennzeichnete Element an die erste Position geschoben wird.

```
void reverse(BidIt first, BidIt last);
OutIt reverse_copy(BidIt first, BidIt last,
          OutIt x);
void rotate(FwdIt first, FwdIt middle, FwdIt last);
```

```
OutIt rotate_copy(FwdIt first,FwdIt middle,
           FwdIt last,OutIt x)
```

Die zweite Funktionengruppe ordnet die Elemente nach dem Zufallsprinzip beziehungsweise der durch die Funktion `f()` gegebenen Reihenfolge an.

```
void random_shuffle(RanIt first, RanIt last);
void random_shuffle(RanIt first, RanIt last,
            Fun& f);
```

Die Veränderung der Reihenfolge kann auch systematisch erfolgen, in dem ausgehend von einer sortierten Folge nacheinander alle Permutationen erzeugt werden. Jeder Aufruf erzeugt eine andere Reihenfolge, der Rückgabewert ist `true`, wenn die sortierte Reihenfolge wieder erreicht wird. Ein Beispiel zu diesem Algorithmus findet sich im Unterkapitel „Auswahl des richtigen Algorithmus“

```
bool next_permutation(BidIt first, BidIt last);
bool next_permutation(BidIt first, BidIt last,
                      Pred pr);
bool prev_permutation(BidIt first, BidIt last);
bool prev_permutation(BidIt first, BidIt last,
                      Pred pr);
```

Die Elemente werden durch die Algorithmen der dritten Gruppe in zwei Klassen sortiert, wobei die ersten Elemente bis zum Rückgabeiterator die angegebene Relation erfüllen, die weiteren Elemente nicht. Der zweite Algorithmus ändert dabei die relative Position der Elemente in einer Partition nicht, das heißt kommt x vor y im Quellcontainer und befinden sich beide anschließend in der gleichen Partition, so kommt immer noch x vor y .

```
BidIt partition(BidIt first, BidIt last, Pred pr);
FwdIt stable_partition(FwdIt first, FwdIt last,
                       Pred pr);
```

In der vierten Gruppe werden die Elemente mit Hilfe der Relation < (*oder einer speziellen Relation*) vollständig sortiert, wobei die zweite Teilgruppe Elemente mit gleicher Bewertung in der ursprünglichen relativen Reihenfolge belässt. Die dritte Teilgruppe sortiert nur die Elemente, die kleiner als das angegebene Element `middle` sind, die vierte Teilgruppe generiert eine Partition um den angegebenen Iterator `nth` oder die angegeben Relation.

```
void sort(RanIt first, RanIt last);
void sort(RanIt first, RanIt last, Pred pr);
void stable_sort(BidIt first, BidIt last);
void stable_sort(BidIt first, BidIt last, Pred pr);
void partial_sort(RanIt first, RanIt middle,
                  RanIt last);
void partial_sort(RanIt first,RanIt middle,
                  RanIt last,Pred pr)
```

```
RanIt partial_sort_copy(InIt first1, InIt last1,
                RanIt first2, RanIt last2);
RanIt partial_sort_copy(InIt first1, InIt last1,
               RanIt first2, RanIt last2,
                 Pred pr);
void nth_element(RanIt first, RanIt nth,
                 RanIt last);
void nth_element(RanIt first, RanIt nth,
                 RanIt last, Pred pr);
```

Eine spezielle Gruppe von Sortieralgorithmen bilden die heap –Algorithmen, die binäre Bäume erzeugen (*siehe Sortieralgorithmen, Kap. 4.6.1*). Der Algorithmus `make_heap` erzeugt das in Kap. 4.6.1 dargestellte Speicherschema, der Algorithmus `sort_heap` erzeugt daraus eine Sequenz mit normaler aufsteigender Sortieren. `push_heap` und `pop_heap` setzen das Vorliegen einer Heap–Sortierung voraus. `push _heap` sortiert das letzte Element in den Heap ein, `pop_heap` kopiert das erste Element an die letzte Position und sortiert den Heap neu. Die Einzelheiten kann der Leser anhand des Schemas in Kap. 4.6 nachvollziehen.

```
void push_heap(RanIt first, RanIt last);
void push_heap(RanIt first, RanIt last, Pred pr);
void pop_heap(RanIt first, RanIt last);
void pop_heap(RanIt first, RanIt last, Pred pr);
void make_heap(RanIt first, RanIt last);
void make_heap(RanIt first, RanIt last, Pred pr);
void sort_heap(RanIt first, RanIt last);
void sort_heap(RanIt first, RanIt last, Pred pr);
```

2.9.13 Extremalwerte

Die folgenden Funktionen ermitteln das kleinste beziehungsweise größte Element in einem Container. Die Methoden `min(..)/ max(..)` sind für die Untersuchung einzelner Elemente implementiert.

```
FwdIt max_element(FwdIt first, FwdIt last);
FwdIt max_element(FwdIt first, FwdIt last,
                  Pred pr);
FwdIt min_element(FwdIt first, FwdIt last);
FwdIt min_element(FwdIt first, FwdIt last,
                  Pred pr);
```

In den folgenden vier Methoden wird die Position ermittelt, hinter der oder vor der ein neues Element eingefügt werden kann. `equal _range` kombiniert beide Operationen und gibt beide Iteratoren in einer `pair`–Variablen zurück

```
FwdIt lower_bound(FwdIt first, FwdIt last,
                  const T& val);
FwdIt lower_bound(FwdIt first, FwdIt last,
                  const T& val, Pred pr);
FwdIt upper_bound(FwdIt first, FwdIt last,
                  const T& val);
FwdIt upper_bound(FwdIt first, FwdIt last,
                  const T& val, Pred pr);
pair<FwdIt, FwdIt> equal_range(FwdIt first,
                  FwdIt last, const T& val);
pair<FwdIt, FwdIt> equal_range(FwdIt first,
                  FwdIt last,const T& val,Pred pr);
```

2.9.14 Mischen von Containern

Das Zusammenlegen von Containern kann durch sukzessives Kopieren erfolgen. Sind die Container sortiert, so ist das Ergebnis der folgenden Zusammenführungen ein ebenfalls sortierter Container. Während ein `merge`–Algorithmus alle Elemente beider Container zusammenführt, behandeln die `set`–Algorithmen die Container wie Mengen, das heißt gleiche Elemente tauchen in einer Vereinigung nur einmal auf usw.

```
OutIt merge(InIt1 first1, InIt1 last1,
               InIt2 first2, InIt2 last2, OutIt x);
OutIt merge(InIt1 first1, InIt1 last1,
               InIt2 first2, InIt2 last2, OutIt x,
               Pred pr);
void inplace_merge(BidIt first, BidIt middle,
               BidIt last);
void inplace_merge(BidIt first, BidIt middle,
               BidIt last, Pred pr);
OutIt set_union(InIt1 first1, InIt1 last1,
               InIt2 first2, InIt2 last2, OutIt x);
OutIt set_union(InIt1 first1, InIt1 last1,
               InIt2 first2, InIt2 last2, OutIt x,
               Pred pr);
OutIt set_intersection(InIt1 first1, InIt1 last1,
               InIt2 first2, InIt2 last2, OutIt x);
```

```
OutIt set_intersection(InIt1 first1, InIt1 last1,
               InIt2 first2, InIt2 last2, OutIt x,
               Pred pr);
OutIt set_difference(InIt1 first1, InIt1 last1,
               InIt2 first2, InIt2 last2, OutIt x);
OutIt set_difference(InIt1 first1, InIt1 last1,
               InIt2 first2, InIt2 last2, OutIt x,
               Pred pr);
OutIt set_symmetric_difference(InIt1 first1,
               InIt1 last1,
               InIt2 first2, InIt2 last2, OutIt x);
OutIt set_symmetric_difference(InIt1 first1,
               InIt1 last1, InIt2 first2,
               InIt2 last2, OutIt x, Pred pr);
```

Man kann sich die Frage stellen, ob diese kurzen Erläuterungen überhaupt hinreichend für die Benutzung der STL–Algorithmen sind oder nicht jeweils ausführliche Beispiele sinnvoll gewesen wären. Mit großer Wahrscheinlichkeit werden Sie auch mit diesen kurzen Angaben in der Lage sein, für eine Aufgabe einen oder mehrere geeignete Kandidaten zu ermitteln. Wenn ein Algorithmus erstmalig eingesetzt wird, ist erfahrungsgemäß eine kleine Testreihe, was der Algorithmus macht, was er speziell mit Ihren Daten macht und ob das herauskommt, was Sie erwarten, besser als viele Worte. Ich unterstelle daher, dass unsere kurze Diskussion für eine grundlegende Orientierung ausreicht und Sie Einzelheiten bei Bedarf durch kleine Versuchsimplementationen feststellen, wie im folgenden Beispiel für Permutationsalgorithmus :

```
vector<string>v(0,3) ;
vector<string>::iterator it;
v[0] = "A" ; v[1] = "B" ; v[2] = "C" ;
for(it = v.begin(); it != v.end(); ++it)
    cout << *it << " " ;
cout << endl ;
while ( next_permutation(v.begin(), v.end()) ) {
        for(it = v.begin(); it != v.end(); ++it)
 cout << *it << " " ;
 cout << endl;
}//endwhile
//Programm-Ausgabe:
A B C
A C B
B A C
B C A
C A B
C B A
```

Aufgabe. Entwerfen Sie Testfälle für die verscheidenen Algorithmen und machen Sie sich mit den genauen Funktionalitäten vertraut.[35]

2.10 Relationen und eigene Algorithmen

2.10.1 Binäre und unäre Operatoren

Nicht in der STL vorhandene Algorithmen können alternativ zu einer klassischen Implementierung auch mit Hilfe der bereits erwähnten transform –Algorithmen konstruiert werden. Hier kommen die bei der Diskussion der Sortier- und Suchalgorithmen vorgestellten Klassenimplementationen von Relationen verstärkt zum Einsatz.

```
template<class InIt, class OutIt, class Unop>
    OutIt transform(InIt first, InIt last, OutIt x,
                    Unop uop);
template<class InIt1, class InIt2, class OutIt,
         class Binop>
    OutIt transform(InIt1 first1, InIt1 last1,
        InIt2 first2, OutIt x, Binop bop);
```

Die Funktionen akzeptieren ein oder zwei Eingabeiteratoren und einen Ausgabeiterator sowie eine Operatorfunktion, die die Werte miteinander verknüpft. Da nur von einem der Eingabeiteratoren ein Enditerator zu den Parameter gehört, muss der zweite Eingabecontainer und der Ausgabecontainer, der je nach Operatoraufbau auch einer der Eingabecontainer sein kann, mindestens die gleiche Größe besitzen. Um die Möglichkeiten, die hinter diesem Konzept stecken, erkennen zu können, ist eine genauere Untersuchung der letzten Vorlagenparameter notwendig, die unäre oder binäre Operatoren bezeichnen. Hierfür stellt die STL wiederum eine Reihe von fertigen Operatoren zur Verfügung:

```
// Binäre Funktionen:
divides<T>, minus<T>, modulus<T>, multiplies<T>,
negate<T>, plus<T>,

equal_to<T>, greater<T>, greater_equal<T>,  identity<T>,
less<T>, less_equal<T>, not_equal_to<T>,
```

[35] Die STL enthält möglicherweise weitere Algorithmen, die nach Abschluss der Redaktion dieses Kapitels aufgenommen wurden. Beziehen Sie auch solche Algorithmen in Ihre Untersuchungen ein.

```
logical_and<T>, logical_not<T>, logical_or<T>,
hash<T>,
// Unäre Funktion:
unary_negate<AdaptablePredicate>
```

Die Operatoren sind ihrerseits Spezialisierungen zweier Basisklassen, die vom Anwendungsprogrammierer für die Implementation eigener spezieller Operatoren genutzt werden können, falls das gewünschte nicht in der Liste vorhanden ist:

```
template<class Arg, class Result>
    struct unary_function {
        typedef Arg argument_type;
        typedef Result result_type;
    };
template<class Arg1, class Arg2, class Result>
    struct binary_function {
        typedef Arg1 first_argument_type;
        typedef Arg2 second_argument_type;
         typedef Result result_type;    };
```

Diese Basisklassen können naturgemäß noch keine eigenen Funktionen ausführen und stellen lediglich Typisierungen der `template` –Parameter bereit, die für Typkontrollen in komplexeren Methoden verwendet werden. Die Funktionalität wird in den erbenden Klassen mit Hilfe des Klammeroperators `operator()(..)` implementiert. `negate` als unäre und `plus` als binäre Funktion sind damit folgendermaßen spezialisiert:

```
template<class T>
struct negate : public unary_function<T, T> {
   inline T operator()(const T& x) const
        { return -x; };
};
template<class T>
struct plus : public binary_function<T, T, T> {
    inline T operator()(const T& x, const T& y) const
        {return x+y;};
};
```

Die Ersatzimplementierung für `valarray<T=::operator+(..)` mit dem Typ `vector` ist der Algorithmus:

```
vector<long>a,b;
...
transform(a.begin(),a.end(),b.begin(),a.begin(),
       plus<long>());
```

mit dem formalen Innenleben

```
for(;it1!=end1;++it1,++it2,++ot)
    *ot=plus(*it1,*it2);
```

Aufgabe. Geben Sie die formale Implementation für einige der anderen Standardoperatoren an.

Das Konzept ermöglicht auch das Durchbrechen einer weiteren Schranke: War es nach der bisherigen Diskussion nur möglich, Iteratoren unterschiedlicher Container, aber gleicher Datentypen zu mischen, zu können nun natürlich auch Operatoren definiert werden, die beliebige unterschiedliche Typen verarbeiten.

Aufgabe. Implementieren Sie einen unären Operator, der ein Feld von Vektoren auswertet und ein Feld der Beträge der Vektoren erzeugt.

2.10.2 Adapterklassen für komplexe Operationen

Mit diesem Satz an Funktionen sind allerdings eine Reihe häufig auftretender Fälle nur unzureichend abgedeckt. Beispielsweise sind die Operationen

```
(*it < 7)        // Größenvergleich mit einer Konstanten
...
*ito = *it1 * *it1 + *it2  // mehr als 2 Operationen
```

so nicht umzusetzen und müssen durch eigene Operatoren zu implementiert werden. Dies würde aber bedeuten, dass jede etwas komplexere Anwendung größere Mengen eigener Operationen implementieren müsste, die nur einmal benötigt werden – ein nicht praktikabler Aufwand gegenüber der herkömmlichen Arbeitsweise. Spezielle Adapterklassen beheben dieses Problem (*zumindest für einfache Beziehungen*). Die beiden Beispiele lassen sich nämlich durchaus mit den vorhandenen Operatoren bearbeiten, wenn im ersten Beispiel eine Konstante anstelle eines Iterators in den Vergleichsoperator eingesetzt wird, im zweiten Beispiel mehrere Operatoren miteinander gekoppelt werden. Für diese unterschiedlichen Aufgaben existieren mehrere Adapterklassen.

Die Adapter `binder1st` und `binder2nd` erlauben es, binäre Funktionen in Algorithmen einzusetzen, die nur einen Eingabeiterator besitzen. Der zweite für binäre Funktionen notwendige Eingabewert ist eine Konstante, die vom Adapter bereit gestellt wird, der wiederum konsequenterweise eine unäre Funktion ist. Die beiden Adapter unterscheiden sich lediglich darin, ob das erste oder das zweite Argument das konstante ist. Wir werden hier einen Fall untersuchen; den anderen können Sie entsprechend konstruieren.

Soll beispielsweise das erste Element größer Null einer Sequenz gesucht werden, so wird dem zweiten Argument des Operators `greater` mit einem Binder der Wert Null zugewiesen:

```
list<int>::iterator first_positive =
    find_if(L.begin(), L.end(),
    bind2nd(greater<int>(), 0));
```

Aufgabe. Bevor Sie jetzt weiterlesen, legen Sie bitte eine kurze Pause ein und skizzieren Sie eine Implementation der Klasse `binder2nd` . Stimmen Ihre Vorstellungen in etwa mit der Implementation überein, die in der STL so aussieht?

```
template<class Pred>
class binder2nd : public
   unary_function<Pred::first_argument_type,
                  Pred::result_type> {
public:

    binder2nd(const Pred& pr,
              const Pred::second_argument_type& y):
        op(pr),value(y){}

    result_type operator()(const argument_type& x)
                                              const{
        return pr(x,value);
    }//end function
protected:
    Pred op;
    Pred::second_argument_type value;
};//end class
```

Ist Ihnen aufgefallen, dass die Klasse `binder2nd` heißt, der Aufruf in der `transform(..)`–Methode aber `bind2nd` geschrieben wird und keine Template-Konstruktionen einer Klasse enthält? Zur Vereinfachung der Erzeugung einer Instanz der Binderklasse und zur Typisierung und Typprüfung ist eine Template-Methode implementiert, die einiges an Schreibarbeit einspart:

```
template<class Pred, class T>
inline binder2nd<Pred>
  bind2nd(const Pred& _X, const T& _Y){
    return (binder2nd<Pred>(_X,
            Pred::second_argument_type(_Y))); }
```

Setzen Sie zur Übung nun einmal die Implementation von `greater` in den Code ein und lösen Sie die Relationen auf (*das ergibt einen guten Einblick in das, womit sich der Compiler zu beschäftigen hat*):

```
template<class _Ty>
  struct greater :
           binary_function<_Ty, _Ty, bool> {
  bool operator()(const _Ty& _X, const _Ty& _Y)
           const
         {return (_X > _Y); }
};
```

An der Implementation wird übrigens auch der Sinn der Typisierung in den Basisklassen deutlich: Ohne die Typisierung `second_argument_type` usw. ist es nicht möglich, die Verwendung eines binären Operators als `template`–Parameter durch den Compiler sicherstellen zu lassen.

Auf ähnliche Art werden auch komplere Operationen realisiert, die mehr als zwei Größen miteinander verknüpfen. Soll beispielsweise das erste Element im Bereich $1 \leq x \leq 10$ in einem Container gefunden werden, so erfordert dies die Auswertung von

```
(1 <= x ) && (x <= 10)
```

Nach unseren bisherigen Überlegungen liesse sich die durch einen Adapter realisieren, der das Ergebnis zweier unärer Operationen des `bind2nd`-Typs als Eingaben für eine binäre Operation behandelt. Der Adapter besitzt also den Typ `binary_function`.

Die Berechnung von $\sin(x^2)$ erfordert einen Adapter, der das Ergebnis einer unären Operation in eine weitere unäre Operation einspeist. Da er nur einen Eingabeparameter besitzt, ist er vom Typ `unary_function`. Wir benötigen auf jeden Fall zwei Adaptermethoden `compose1` und `compose2`. Bevor wir überlegen, ob weitere Adapterklassen notwendig sind, entwerfen wir zunächst Konstruktionen für diese beiden. Der unär-unär-Typ besitzt das Aussehen

```
template <class UnOp1, class UnOp2>
class composer_1: public
  unary_function<typename UnOp2::argument_type,
                 typename UnOp1::result_type>
{
protected:
    UnOP1 op1;
    UnOp2 op2;
public:
  composer_1(const UnOp1& __x,
           const UnOp1& __z )
         :op1(__x), op2(__y) {}
  typename result_type operator()
    (const argument_type& x) const
```

```
    {return op1(op2(x)));
  }
};//end class
```

Beachten Sie, welche Typen für die Typisierung verwendet werden. Der in der Operation verwendete Iterator besitzt den Argumenttyp des inneren Unäroperators, der Ausgabeiterator den Rückgabetyp des äußeren.

> **Aufgabe.** Wir bei den `binder`-Adaptern kann die Implementation durch eine Methode `compose1` erleichtert werden, die von der Aufzählung der Templateparameter entbindet und ein Objekt des Typs `composer _1` als Rückgabewert besitzt. Implementieren Sie eine solche Methode.

Für den zweiten Typ erhalten wir mit entsprechenden Überlegungen die Definition

```
template <class BinOp, class UnOp1, class UnOp2>
class composer_2: public binary_function<
             typename UnOp1::argument_type,
             typename UnOp2::argument_type,
             typename BinOp::result_type>
{
protected:
    BinOP _M_fn1;
    UnOP1 _M_fn2;
    UnOp2 _M_fn3;
public:
  composer_2(const BinOp& __x,
             const UnOp1& __y,
             const UnOp1& __z )
        :_M_fn1(__x), _M_fn2(__y), _M_fn2(__z) {}
  typename result_type operator()
    (const first_argument_type& __x,
     const second_argument_type& __y) const
     {return _M_fn1(_M_fn2(__x),_M_fn3(__y));
  }
};//end class
```

Die erste Aufgabe, die Bestimmung von $1 \leq x \leq 10$, wird unter Verwendung eines der Standardalgorithmen der STL durch die Relation

```
list<int>::iterator in_range =
  find_if(L.begin(), L.end(),
     compose2(logical_and<bool>(),
```

```
        bind2nd(greater_equal<int>(), 1),
        bind2nd(less_equal<int>(), 10)));
```

realisiert, der zweite durch

```
transform(c.begin(),c.end(),
       compose1(func(sin()),func(square())));
```

Hierbei sind die Sinus- und die Quadratfunktion durch spezielle unäre Operationen zu realisieren (*man kann entsprechende unäre Operatoren definieren oder Funktionsadapter verwenden*, *siehe unten*).

Sind damit alle notwendigen Adapterklassen definiert? Die Beispiele $z = x*y+y$ und $z = u*v+w$ scheinen dem zu widersprechen und Adapterklassen für mehrere binäre Operationen zu erfordern. Aber Vorsicht! Bevor Sie nun zur Konstruktion von `composer _3` schreiten, sollten sie folgendes bedenken:

- Das erste Beispiel weist nur einen Pseudobedarf an zwei binären Operationen auf. Der Primäradapter übernimmt nachwie vor nur zwei Argumente, die er in bestimmter Weise auf zwei binäre Operationen verteilen muss. Solche Spezialfälle sind also eher durch eine entsprechende Spezialisierung von `binary _function` zu realisieren als durch eine weitere abstrakte Klasse.
- Das zweite Beispiel ist echt, aber hierfür existiert noch gar keine `transform`-Funktion mit drei Eingabeiteratoren. Erst wenn Sie diese implementieren, können Sie mit Hilfe von `composer _3` dafür sorgen, dass die vorhandenen unären und binären Operatoren weiter genutzt werden können.

Für die Verwendung vorhandener Funktionen sind Funktionsadapter definiert, so dass Funktionen nicht extra in ein Operatorenformat umgeschrieben werden müssen. Die Definition ist nun leicht verständlich

```
template <class _Arg, class _Result>
class pointer_to_unary_function :
    public unary_function<_Arg, _Result> {
protected:
  _Result (*_M_ptr)(_Arg);
public:
  pointer_to_unary_function();
  pointer_to_unary_function(_Result (*__x)(_Arg)) :
         _M_ptr(__x) {}
  _Result operator()(_Arg __x) const {
        return _M_ptr(__x); }
};
```

Aufgabe. Implementieren Sie einen Adapter für binäre Funktionen sowie Aufrufmethoden `func(..)` für die Implementation.

Einige Container arbeiten mit dem Datentyp pair<A,B=, der für die bisherigen Operatoren unzugänglich ist. Zwei Adapter beheben auch dieses Problem:

```
template <class _Pair>
struct _Select1st : public
  unary_function<_Pair,
                 typename _Pair::first_type> {

   const typename _Pair::first_type& operator()
    (const _Pair& __x) const {
   return __x.first;
  }
};
```

2.10.3 Aufwandsabschätzung

Grundsätzlich lassen sich auf diesem Weg fast alle Algorithmen formulieren, die elementweise Umrechnungen von Containerelementen durchführen, wenn das Bild vielleicht auch recht ungewohnt ist. Der Abstand zur mathematischen Formulierung scheint größer zu werden, wenn Methoden und Adapter verwendet werden, gegebenenfalls in iterierter Form wie in einigen der angegebenen Beispiele. Wir wollen daher an dieser Stelle eine Frage stellen: Lohnt sich die Verwendung dieser Techniken überhaupt? Sehen wir uns daher die Gründe, die für eine Verwendung dieser Technik sprechen, genauer an.

A. Die meisten der von der STL angebotenen Funktionen scheinen relativ einfach, so dass es dem Programmentwickler anfangs vermutlich leichter fällt, in drei bis sechs Programmzeilen die Funktion direkt zu implementieren als in der STL die passenden Algorithmusfunktionen zu identifizieren und sich mit der ungewohnten Notation herumzuschlagen.[36] Da der Programmierentwickler das Innenleben des Containers nicht so detailliert kennt, wie der STL–Entwickler, muss dieser Code allerdings nicht unbedingt die effizienteste Lösung sein, und auch nicht jeder denkt automatisch an bestimmte Optimierungsschritte. Zwischen „spontanen“ und „state of the art “–Implementierungen können aber durchaus auch Größenordnungen in der Ausführungszeit liegen. Als einfaches Beispiel begutachten Sie bitte die folgenden Implementierungen:

```
container<T> a;
container<T>::iterator it;
```

[36] Ungewohnt heißt nicht ungewöhnlich oder andersartig. Wenn man beispielsweise in etwas aufwendigeren arithmetischen Formeln die implizite Reihenfolge von Rechenoperationen, also $*$ vor + usw., durch Klammerung vollständig darstellt, also a∗b∗c+d = (((a∗b)∗c)+d, so braucht man Klammern und Operatoren im Grunde nur durch die Bezeichnungen der STL zu ersetzen. Die Schreibweise ist somit zwar ungewohnt, weicht aber nicht vom normalen mathematischen Standard ab.

```
// Alternative 1
for(it=a.begin();it!=a.end();++it){...}
// Alternative 2
container<T>::iterator et(a.end());
for(it=a.begin();it!=et;++it){...}
```

Sofern sich hinter `a.end()` etwas anderes verbirgt als ein `inline`–Durchgriff auf ein Attribut, ist die zweite Version vorzuziehen. Zum einen weiß das aber nur der Entwickler der Bibliothek, zum anderen denkt nicht jeder Entwickler immer an die zweite Alternative. Das ist aber nur ein Beispiel. Insgesamt gilt

- Einfache Optimierungen werden nicht übersehen, der Code ist effizient.
- Der Code ist von vornherein stabil.[37] Fehler durch mehrfache Verwendung des gleichen Parameters können nicht auftreten. Auch die Fallen ungültiger Iteratoren nach Einfüge- oder Löschoperationen sowie die Probleme beim Übergang vom inversen zum Vorwärtsiterator werden nicht übersehen.
- Der Algorithmus ist optimal gewählt. Soll beispielsweise eine Sortierung vorgenommen werden, so kann zwischen einer Vielzahl unterschiedlicher Algorithmen gewählt werden. Wenn sich der optimale Algorithmus auch erst bei genauer Analyse des vorliegenden Problems ergibt, so existieren doch bestimmte Präferenzen für die verschiedenen Containertypen, die auf jeden Fall berücksichtigt sind. Und da man wohl unterstellen kann, dass der Container nach einer Problemanalyse bewusst ausgewählt wurde, ist der automatisch implementierte Algorithmus mit guter Wahrscheinlichkeit mit dem optimalen identisch.
- Versteckte Optimierungen sind berücksichtigt. Der Bibliotheksprogrammierer kennt alle Eigenschaften des Containers und der Iteratoren und kann daher an Stellen optimieren, die außerhalb der Möglichkeiten eines normalen Programmierers liegen.

B. Die Notation unterscheidet sich zwar von mathematischen Formeln, die man in eigenen Algorithmen 1:1 umsetzen könnte, und die Ausdrucke werden teilweise recht lang. Verwendet man bei längeren Schachtelungen in einem Algorithmus jedoch eine strukturierte Schreibweise, wie sie für die Anweisungen im Code ebenfalls Verwendung finden, so lässt sich feststellen

- Der Code wird nicht unübersichtlich, sondern ist unmittelbar mit der Theorie vergleichbar.
- Der Code wird programmtechnisch gesehen sogar kürzer (*eine Anweisungszeile gegenüber einer* `for` *–Schleife mit mehreren Anweisungen*) und aussagekräftiger, da der Name des Algorithmus bereits im Klartext aussagt, was passieren wird. All das erleichtert die Revision.

[37] Diese Aussage bezieht sich natürlich nicht auf rechnerische Instabilitäten.

- Durch die durchgehende Verwendung von `inline`–Anweisungen kann der Übersetzer hocheffizienten Code erzeugen, der in der Effizienz direkt notierten Formeln nicht nachsteht.

Wenn Sie in der STL nicht vorhandene Algorithmen implementieren und sich an die vorgestellten Konstruktionsprinzipien halten, entsteht bei der Implementation von Operatoren hocheffizienter Code.

2.10.4 Ein Beispiel

Geben wir noch ein Beispiel zur praktischen Demonstration dieser Aussagen. Es sei eine Sequenz von Werten gegeben (*beispielsweise durch Einlesen aus einer Datei*), zu denen eine Konstante addiert werden soll und die anschließend ohne Änderung der Reihenfolge an den Beginn einer bereits existierenden zweiten Sequenz des Typs `deque` geladen werden sollen (`push _back(..)` *kommt also nicht in Frage*). Ohne dass das eine ehrenrührige Bemerkung sein soll, können wir wohl unterstellen, dass unser Musterentwickler die Containerwerkzeuge (*nur*) so weit beherrscht, wie das in den letzten Wochen seiner Arbeit notwendig gewesen ist, und sich ihrer zunächst intuitiv zu bedienen versucht. Ein erster Versuch könnte dann durchaus so aussehen:

```
deque<T> d;
...
for(i=0;i<n;++i){
    read(data);
    d.insert(d.begin(),data+C);
}//endfor
```

Das funktioniert zwar zunächst, jedoch stellt er bei einer anschließenden Prüfung fest, dass die Elemente in der umkehrten Reihenfolge im Container gelandet sind. Also startet er einen zweiten Versuch:

```
deque<T>::iterator it;
...
for(i=0,it=d.begin();i<n;++i,++it){
    read(data);
    d.insert(it,data+C);
}//endfor
```

Sieht gut aus, funktioniert aber gar nicht! Haben Sie den Fehler erkannt? Genau, nach der Einfügeoperation ist der Iterator nicht mehr gültig, und sowohl Inkrementieren als auch die Nutzung im nächsten Aufruf gehen daneben. Der nächste Versuch funktioniert nun allerdings, nachdem sich unser Musterentwickler die Methode `insert` noch einmal genau angesehen hat:

```
for(i=0,it=d.begin();i<n;++i){
    read(data);
    it=d.insert(it,data+C);
}//endfor
```

Als Algorithmus geht das allerdings noch eleganter. Dabei setzen wir zunächst voraus, dass die Werte (*mittels einer* `for` *–Schleife*) in ein Feld `data[n]` einlesen worden sind – eine Operation, bei der auch unser Musterentwickler wahrscheinlich keine Fehler machen kann. Das Feld selbst kann als einfacher Container mit Zeigern auf die Elemente als Iteratoren aufgefasst werden, die Addition der Konstanten können wir durch Binder und binäre Methoden erledigen. Für die gesuchte Einfügeoperation finden wir in der STL die Adapterklasse

```
template<class Cont>
class insert_iterator
        : public iterator<output_iterator_tag, void,
         void> {
public:
    typedef Cont container_type;
    typedef Cont::value_type value_type;
    explicit insert_iterator(Cont& x,
          Cont::iterator it);
    insert_iterator& operator=(
          const Cont::value_type&  val);
    insert_iterator& operator*();
    insert_iterator& operator++();
    insert_iterator& operator++(int);
protected:
    Cont& container;
    Cont::iterator iter;
};//end class
```

die über die Hilfsfunktion `inserter` initialisiert wird und genau das macht, was wir hier benötigen: Das Einfügen von Elementen in eine Ausgabe–Sequenz unter Beibehaltung der Reihenfolge. Unser Problem wird nun durch den Algorithmus

```
T dat[n];
...
transform(dat,dat+n,
            inserter(d,d.begin()),
                   bind2nd(plus<T>(),C));
```

gelöst. Um zu dieser Implementation zu gelangen, ist sicher auch etwas Aufwand notwendig (*schließlich haben wir erst eine weitere Adapterklasse ausfindig machen müssen*), aber wenn wirklich so viel schief laufen sollte, wie im Beispiel angegeben, hätte sich der Aufwand der Suche in der Bibliothek sicher gelohnt.