Anwendung auf Algorithmen
Lineare Ordnung durch ein Interface (Java comparable)

C++

```
class comparable{
   virtual int compare (comparable& x) = 0;
```

Die Funktion vergleicht das Objekt, welches die Funktion aufruft, mit x und liefert

$$\begin{cases} +1, & \text{falls dieses } > x \\ 0, & \text{falls dieses } = x \\ -1, & \text{falls dieses } < x \end{cases}$$

Anwendung in generischen Sortieralgorithmen
Quicksort(comparable * A[])
zum Vergleich if(x.compare(y) ≤ 0) ...
Sortiere ein Feld von point:

```
class point : public comparable{
   point ...
   int compare(comparable & p){}
      // p ist tatsächlich ein Point!
   double px = ((point&) p).x;
   double py = ((point&) p).y;
   (x,y) < (px,py)
   // lexikographische Ordnung
   return ... ;
};
```

Aufruf von Quicksort

```
point* A [] = new point*[100];
(point**)
for( i=o; i<100; i++){
   A[i] = new point(i, i*i);
}
Quicksort(A, 100);
```

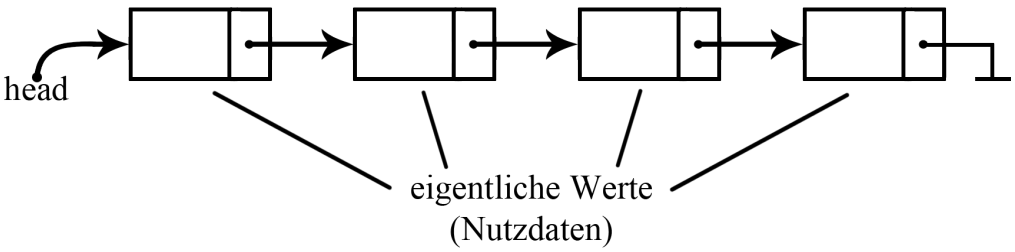
Datenstrukturen: → binäre Suchbäume, Schlüssel vom Typ „comparable“

3.3 Weitere Anwendungen von Vererbung

3.3.1 Generische Datenstrukturen

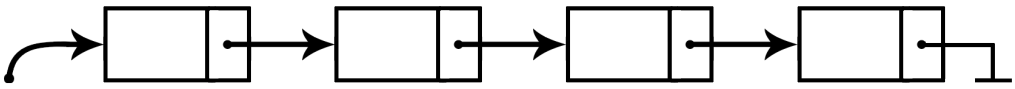
Eine generische Datenstruktur ist zum Beispiel eine Liste von beliebigen Objekten. Diese Liste soll leicht wiederverwendbar sein z.B. eine einfach verkettete Liste. Dadurch muss sie nicht immer neu implementiert werden.

Einfach verkettete Listen



Beobachtung:
Implementierung der Operationen (push, pop, ...) ist unabhängig vom Typ der Werte (int, string, point, ...).

Abstrahieren: Liste ohne Werte



1.) Basisklasse für allgemeine Listenelemente

```
class slist_elem{
   slist_elem* next;
   slist_elem(slist_elem* p){
      next = p;
   }
}
```

2.) Basisklasse für allgemeine Liste

```
class slist{
   slist_elem* first;
   slist(){
      first = NULL;
   }

   // hier eigentlich Destruktor

   void push(slist_elem * p){
      p->next = first
      first = p;
   }

   slist_elem* pop(){
      if(first == NULL) return first;
      slist_elem* p = first;
      first = first.next;
      return p;
   }
}
```

`slist` funktionier auch für alle von slist_elem abgeleiteten Klassen.

Beispiel 3.4 Liste von points

```
class point: public slist_elem{
   // wie vorher ...
}
point p
```

```
class point_list: public slist{
   // neues Interface, das nur Points erlaubt
   // -> Filter
   void push(point* p) {
      slist :: push(p);
   }
   point* pop(){
```

```
        return (point*) slist :: pop();
        // Casting ist sicher, da nur points in
        // der Liste sind
    }
    ...
}
```

1.) Leite point von slist_elem ab
2.) slist kann nun für point verwendet werden

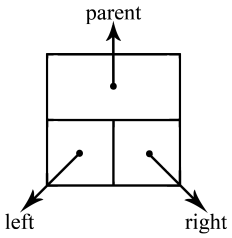
```
slist L;
point* p new point(x,y);
L.push(p);
point* q = (point*)L.pop();
```

Man hat nun Polymorphie.
`L.push(new rechteck(...));` ist auch möglich.

→ doppelt-verkettete Listen

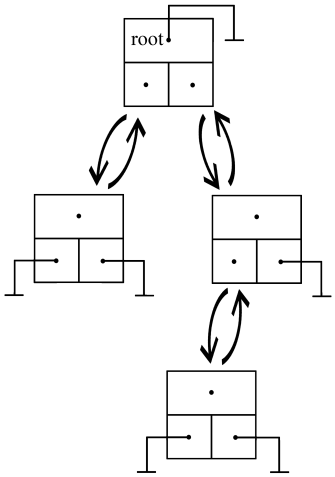
Aufwendigste Datenstruktur: z.B. balancierter Suchbaum z.B. AVL-Baum

Die 1. Klasse wird für die Knoten genutzt.




```
class bin_tree_node{
    bin_tree_node* left;
    bin_tree_node* right;
    bin_tree_node* parent;
};
```

Die 2. Klasse wird für den Baum genutzt.




```
class bin_tree{
   virtual int comp(bin_tree_node* p,
            bin_tree_node* q) = 0
   // Die Funktion comp(...) wird in einer
   // abgeleiteten Klasse definiert.
   void insert(bin_tree_node* p){
      // fügt p in den Baum ein und
      // verwendet comp zum Vergleich
   }
   bin_tree_node* lookup(bin_tree_node* p){
      if(comp(q,p) > 0)
         q = q->left;
      else
         q = q->right;
   }
};
```

Anwendung auf points

```
class point: public bin_tree_node{
   double x,y;
}
```

```
class point_bin_tree: public bin_tree{
   // Definiere comp-Funktion
   int comp(bin_tree_node* p, bin_tree_node* q){
      point* a = (point*) p;
      point* b = (point*) q;
      // lexikografische Ordnung:
      if(a->x < b->x) return -1;
      if(a->x > b->x) return +1;
      if(a->y < b->y) return -1;
      if(a->y > b->y) return +1;
      return 0;
   }
   void insert(point* p){
      bin_tree :: insert(p);
   }
   point* min(){
      return (point*) bin_tree :: min();
   }
```

bin_tree_node → point
bin_tree → point_bin_tree
Wiederverwendung von Cache!

Templates oder Schablonen

Eine weitere Anwendung von generische Datenstrukturen ist ein Template oder eine Schablone.

1) Funktionstemplates
Ein Funktionstemplate ist eine „Schablone“ die angibt, wie eine Funktion erzeugt werden soll. Aus einem solchen Template können Funktionen mit gleichem Ablauf aber verschiedenen Parametertypen erzeugt werden. Die Funktion hat also immer die gleiche Parameteranzahl aber die Parameter können unterschiedliche Datentypen haben.

```
template<class T> // "class" kann durch
   // "typename" ersetzt werden
   void swap(T& x, T& y){
      T tmp = x;
      x = y;
      y = tmp;
   }
```

Wichtig: Die Implementierung ist nicht abhängig vom Typ T.
Anwendung:

```
double a = 3.14;
double b = 15;
swap(a,b);
```

Der Compiler instanziiert das Template für T = double.
Ähnliche Funktionen sind Minimum (min(x,y)) und Maximum (max(x,y))

2) Klassentemplates
Klassentemplates lassen sich im Gegensatz zu Funktionstemplates nicht überladen.

Beispiel für einen Stack:

```
template<class T>
   class stack{
      T* A; // Feld von T's
      int sz;
      int t;
      public:
         void push(T x){...}
         T pop(){...}
   }
```

Verwendung:

```
stack<int> S;
stack<point> S;
```

Beispiel 3.5 Beispiel für mehrere Typen

```
template<class K, class I>
   class dictionary{
      // Wörterbuch mit Schlüssel vom Typ K
      // und Informationen vom Typ I
      // mögliche Implementierung: AVL-Baum
      void insert(K, I){
         // trägt neues Paar ein
      }
      bool lookup(K key){
         // testet, ob Eintrag mit Schlüssel key
            vorhanden ist.
      }
      I translate(K key){...}
      void remove(K key){...}
   }
```

Anwendung: Word-Count
Zähle wie oft einzelne Wörter in einem Text vorkommen.
`dictionary<string, int> D;` speichert Paare (S, i). Der String S ist bis jetzt i mal vorgekommen.

Templates in Kombination mit Vererbung am Beispiel einer einfach verketteten Liste:

```
template<class T>
   class list :: public slist{
      void push(T* x){slist :: push(x);}
      T pop(){return (T*) slist :: pop();}
      ...
   }
```

analog: bin_tree<T>

4 Fortgeschrittene Datenstrukturen und Algorithmen

LEDA: Library of Efficient Data Types and Algorithms

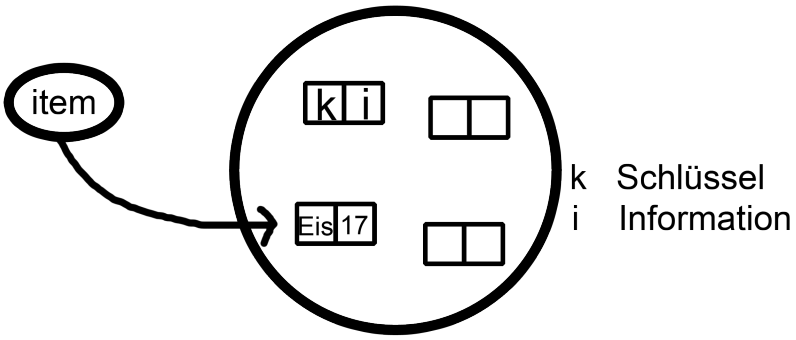
4.1 Spezifikation von Datentypen in LEDA

4.1.1 Item-Konzept

Viele Datenstrukturen werden dabei definiert als eine Menge von sogenannten Items.

Beispiel 4.1 Dictionary
`dictionary<string, int> D` speichert Paare aus Schlüseln (strings) und Informationen (int).

D ist eine Menge von Items (dic_item).



Operationen

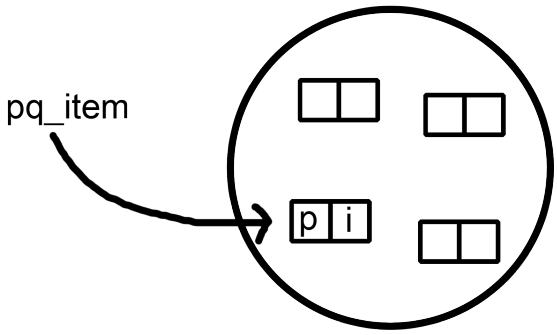
```
dic_item D.insert(string s, int i)\\
   if( D enthält kein Item mit Schlüssel s){
      Füge Item (s,i) ein;
      return (s,i);
   }
   else{
      Ersetze die Information durch i;
      return (s,i);
   }
}
dic_item D.lookup(string s){
   Liefert das Item mit Schlüssel s
   if(kein solches existiert)
      return NULL;
}
void D.change_inf(dic_item it, int x){
   ersetze die Information von it durch x
}
void delete_item(dic_item it){
   Item: Zugriff über Position (Abstraktion von den
      Begriffen Pointer, Referenz, Index)
}
```

Vergleich verschiedener Spezifikationen
Dictionary als Funktion $D : K \to I$ $\leftrightarrow$ Item-Konzept

Beispiel 4.2 Priority Queues
Da bei Priority Queues immer ein Extremwert gesucht wird, eigenen sich für die Implementation Fibonacci-Heaps besonders gut, da eine Abfrage des Extremwertes in $\mathcal{O}(1)$ möglich ist.

`priority_queue<P,I> PQ;`
P: Priorität z.B. Zahlen (int)
I: Information z.B. Knoten eines Graphen (node)
`priority_queue<int, node> PQ;`

PQ ist eine Menge von Items (pq_item)



Operationen

```
pq_item PQ.insert(P prio, I inf)\\
      Fügt Item (prio, inf) ein; // Im Gegensatz zum
         Dictionary dürfen bei der Priority-Queue
         Wiederholungen auftreten.
      return (prio, inf);
}
PQ.findmin(){
   Liefert ein Item mit minimaler Priorität.
   Falls die Queue leer ist, wird NULL zurück gegeben.
}
P PQ.prio(pq_item it)
I PQ.inf(pq_item it)
void PQ.delmin(){
   Löscht ein minimales Item.
}
void PQ.decrease_p(pq_item it, P q){
   Vermindert die Priorität von it auf q.
   (q ist kleiner gleich der alten Priorität von it)
}
```