3 Generische Programmierung

3.1 Generische Funktionen:

```
template<class C>
C Max(C a, C b) { return a > b ? a : b; }
```

Aufruf funktioniert nur wenn `operator>` definiert ist.

`Max(2,3)` gleichbedeutend mit `Max<int>(2,3)` (*Parameterdeduktion*).

`Max<double>(2.0,3)` hier automatische Deduktion nicht möglich.

3.2 Generische Klassen

Ein Bespiel: Längenbegrenzte Stacks

Verwalte Keller mit gegebener Maximallänge.

Headerfile:

```
#ifndef LEDA_BSTACK_H
#define LEDA_BSTACK_H
...
#endif
```



```
template<class E>
class b_stack {
  E* first;  // start of array
  E* stop;   // yust beyond allocated space
  E* free;   // pointer to first free element

public:
  b_stack(int n); // ...
  ~b_stack(); // ...
  E&   top() const; // ...
  E&   pop(); // ...
  void push(const E& x); // ...
  int  size()  const; // ...
  bool empty() const; // ...
}
```


Konstruktor und Destruktor

```
b_stack(int n) {
  assert(n > 0); // no empty stacks
  free = first = new E[n];
  assert(first != 0);
  stop = first + n;
}

~b_stack() { delete [] first; }
```

Nur-Lese-Funktionen

```
E& top() const {
  assert(!empty());
  return *(free-1);
}
int     size() const { return free - first; }
bool   empty() const { return free != first; }
```


Push und Pop

Hier mal ausserhalb der Klassendefinition.

```
template<class E>
void stack<E>::push(const E& x)
{
  assert(free != stop);
  *free = x;
  free++;
}

template<class E>
E& stack<E>::pop()
{
  assert(!empty());
  free--;
  return *free;
}
```


3.3 Werte als Template-Parameter

Effizienterer längenbegrenzter Keller:

```
template<class E, int n>
class b_stack {
  E first[n];
  E *free;
public:
  E& push(const E& x) {
    assert(free != first + n);
    *free = x;
    free++;
  }
  ...
}
```

- Kein `stop` Member
- Eine Indirektion weniger
- Seltener dynamische Speicherverwaltung

3.4 „Geschichtlicher Rückblick“

```
#define b_stack(E,n)\
  class b_stack_template ## E ## n {\
    E first[n];\
    E *free;\
  public: \
    E& push(const E& x) {\
      assert(free != first + n);\
      *free = x;\
      free++;\
    }\
    ...
  }
```

4 Nützliche Datenstrukturen

4.1 Dynamisch wachsende Felder

```
template<class C>
class DynamicArray {
int currentSize;
int allocatedSize;
C *data;
public:
  DynamicArray(int initialSize = 1) { ... };
  void setSize(int newSize) {
    if (newSize > allocatedSize) {
      allocatedSize *= 2;
      // neu allozieren,
      ...
      // kopieren
      ...
      // altes Freigeben
    }
    // Faelle fuer schrumpfen
```



```
    ...
```

- Reservierter Speicherbereich wächst bei Bedarf geometrisch
- Schrumpfen auch möglich
- **WEHE**, bei Zeigern auf Elemente!
- → STL-Klasse `vector`
- In LEDA intern verwendet

Anwendung für Stapel

- Komfortabler als bounded Array
- Flexibler
- Keine allzu große Speicherverschwendung
- k Operationen gehen in Zeit $\mathcal{O}(k)$;
  → *amortisierte Analyse*
- Sehr schnell (Cache) vor allem bei kleinen Elementen

4.2 Hashtabellen

engl. *to hash*: in kleine Stücke hacken. Substantiv: *hash* = Durcheinander, Kuddelmuddel

Gleiche Wurzel wie *hacken*.

Besser: „Streuspeicherung“?

Gegeben: Menge von Paaren $M = (x_i, y_i)$ ($x_i \in S$ ist *Index*, $y_i \in W$ ist *Element*) Zu gegebenem Schlüssel x, finde y. (+ Einfügen, Löschen etc.)

Idee: Nimm Array $A[n]$ ($n \geq |M|$) und bestimme eine einfach auswertbare Funktion $f : S \to \{1, \ldots, n\}$, so daß $A[f(x_i)] = (x_i, y_i)$.

Problem: $|S| \gg |M|$. $\Rightarrow \neg\exists$ injektive Funktion $\forall M$

Lösung: Quasizufällig Funktion, für die im Mittel $f^{-1}(k)$ klein ist.

Noch zu tun: Kollisionsauflösung. Zum Beispiel Nachbarfeld oder Liste von Treffern.

LEDA: `h_array, map`

4.3 Ein Zoo arraybasierter Datenstrukturen

- FIFO-Schlangen (first in first out). Trick: „Enden zusammenbiegen“.
- Prioritätsschlangen (kleinster Schlüssel zuerst.)
- Teilmengen von $n \ldots n+k$. Feld von Bits. Oft reicht ein Maschinenwort.
- Assoziative Arrays. D.h. eine partielle Abbildung Schlüssel$\rightarrow$ Wert. Wobei Schlüssel z.B. Zeichenketten sein können. $\rightarrow$*Hashtabellen*

4.4 Einfach verkettete Listen

```
template<class E>
struct ListStackItem
{ ListStackItem *next;
  E element;
}

template<class E>
class ListStack {
  typedef ListStackItem<E> Item;
  Item *start;
public:
  ListStack() { size = 0; start = 0; }
```



```
  void push(const E& x)
  { Item *item = new Item; // ineffizient?
    assert item != 0;
    item->next = start;
    item->element = x;
    start = item;
  }

  E& pop() {
    assert(start != 0);
    Item *head = start;
    E result   = head->element;
    start      = head->next;
    delete head; // ineffizient?
  }
}
```


Wann einfach verkettete Listen

- LEDA `slist` (Item enthält Zeiger auf Element) Benutzt für `stack` und `queue`
- Schnelles sequentielles Durchlaufen
- Gut bei großen Elementen
- Einfügen in konstanter Zeit
  - Am Anfang
  - Am Ende
  - Hinter gegebenem Item
- Löschen am Anfang, Am Ende, Nachfolger eines gegebenen Elements
- FIFO
- Konkatenieren
- Zyklische Listen

4.5 Doppelt verkettete Listen

Item speichert Zeiger **Vorgänger-** und **Nachfolger**

- Rückwärts durchlaufen
- Enfügen vor gegebenen Item
- Löschen eines gegebenem Item

4.6 Listen versus Arrays

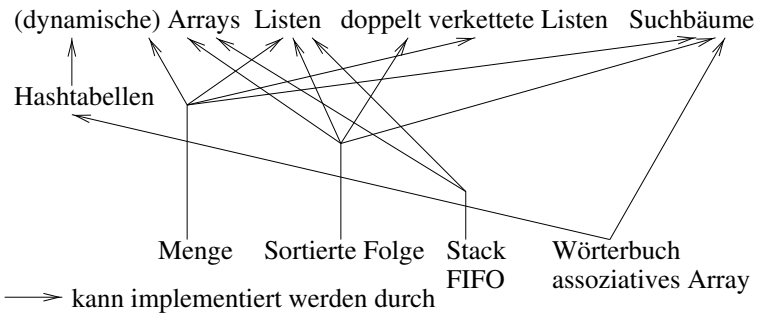
- Arrays sind oft schneller (Cache)
- Listen brauchen mehr Platz (vor allem doppelt verkettete) als Arrays fester Größe
- Arrays erlauben beliebigen Zugriff
- Arrays vereinfachen manchmal Datei-Ein/Ausgabe und Kommunikation
- Listen erlauben Einfügen und Löschen
- Pointer auf Listenelemente funktionieren immer

4.7 Suchbäume

Binäre Suchbäume:

- Knoten speichern Schlüssel
- In-Order-Durchlauf produziert Schlüssel in sortierter Reihenfolge. (linker Teilbaum, Knoten, rechter Teilbaum)

Suchen, Einfügen, Löschen geht in $\mathcal{O}$(Tiefe) Zeit.

Die „ganze Kunst“: logarithmische Tiefe erzwingen.

4.8 Physikalische versus logische Datenstrukturen