

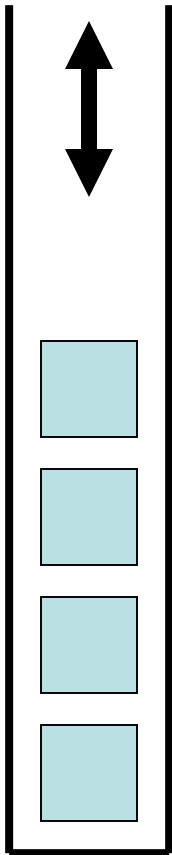
Proseminar
Effiziente Algorithmen
Kapitel 2: Datenstrukturen

Prof. Dr. Christian Scheideler
WS 2017

Übersicht

- Stacks
- Queues
- Dictionaries
- Priority Queues
- Mengen
- Probleme im Bereich Datenstrukturen

Stack

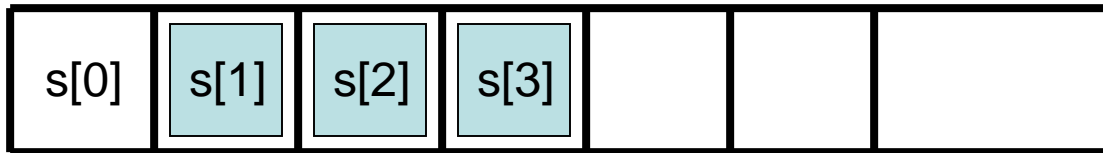


Operationen:

- `Init(s)`: erzeuge leeren Stack `s`
- `Push(x,s)`: packe `x` auf Stack `s`
- `Pop(s)`: entferne oberstes Element von `s`
- `Empty(s)`: testet, ob Stack `s` leer ist

Stack ganzer Zahlen

- Implementierung als Array:



↑
Größe (anfangs 0)

Push(x):

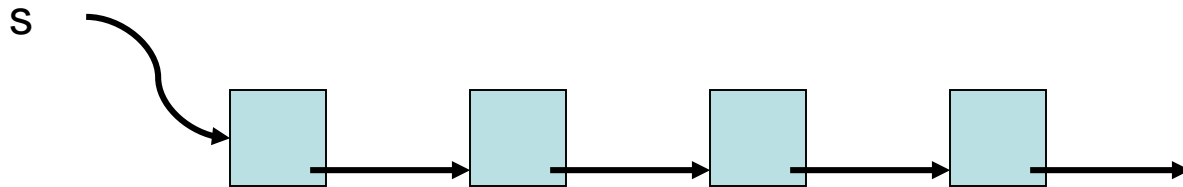
```
s[0] = s[0]+1;  
s[s[0]] = x;
```

Pop():

```
if (s[0]==0) return -1;  
s[0] = s[0]-1;  
return s[s[0]+1];
```

Stack ganzer Zahlen

- Implementierung als Pointer-Struktur:

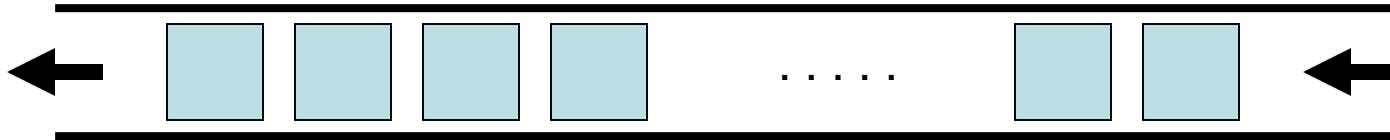


```
class stackel {  
    int zahl;  
    stackel *nachf; };  
stackel *s = NULL;
```

```
Push(x):  
    q = new stackel;  
    q->zahl = x;  
    q->nachf = s;  
    s = q;
```

```
Pop():  
    if (s==NULL) return s;  
    q = s; s = s->nachf;  
    return(q);
```

Queue

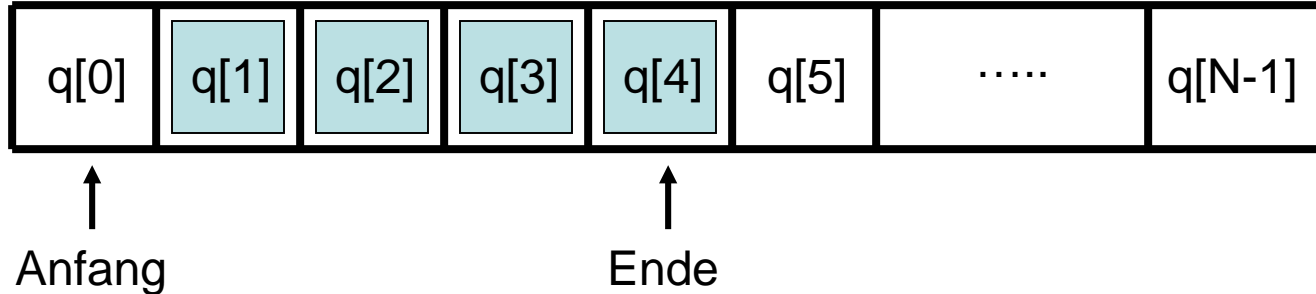


Operationen:

- $\text{Init}(q)$: erzeugt eine leere Queue q
- $\text{Enqueue}(x,q)$: hängt x hinten an Queue q an
- $\text{Dequeue}(q)$: gibt Element vorne in der Queue zurück
- $\text{Empty}(q)$: testet, ob Queue q leer ist

Queue ganzer Zahlen

- Implementierung als Array:



```
class queue {  
    int q[N];  
    int Anfang;  
    int Ende; };
```

```
q = new queue;  
q->Anfang=q->Ende=0;
```

Enqueue(x):

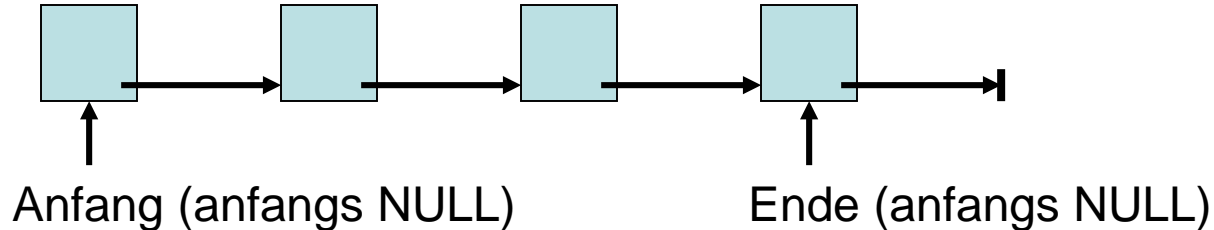
```
Ende=(Ende+1)%N;  
q[Ende]=x;
```

Dequeue():

```
if (Anfang == Ende)  
    return -1;  
Anfang=(Anfang+1)%N;  
return(q[Anfang]);
```

Queue ganzer Zahlen

- Implementierung als Pointer-Struktur:



```
class queueel {  
    int zahl;  
    queueel *nachf; };
```

```
class queue {  
    queueel *Anfang;  
    queueel *Ende; };
```

```
q = new queue;
```

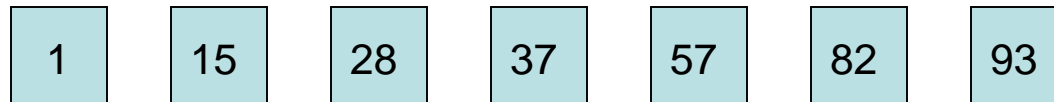
Enqueue(x):

```
p = new queueel;  
p->zahl=x;  
p->nachf=NULL;  
if (Anfang==NULL)  
    Anfang=Ende=p;  
else {Ende->nachf=p;  
    Ende = p; }
```

Dequeue():

```
if (Anfang == NULL)  
    return NULL;  
p = Anfang;  
Anfang=Anfang->nachf;  
if (Anfang == NULL)  
    Ende=NULL;  
return p;
```


Dictionary

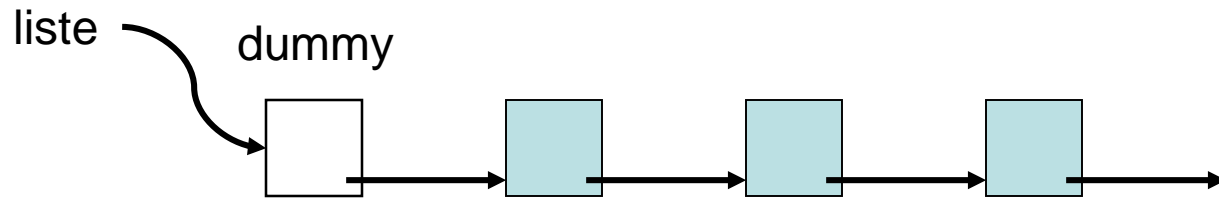


Operationen:

- $\text{Insert}(x,d)$: füge x in das Dictionary d ein
- $\text{Delete}(x,d)$: lösche x (Eintrag oder Schlüssel) vom Dictionary d
- $\text{Search}(k,d)$: suche nach Eintrag mit Schlüssel k in d

Dictionary

- Implementierung als ungeordnete Liste:



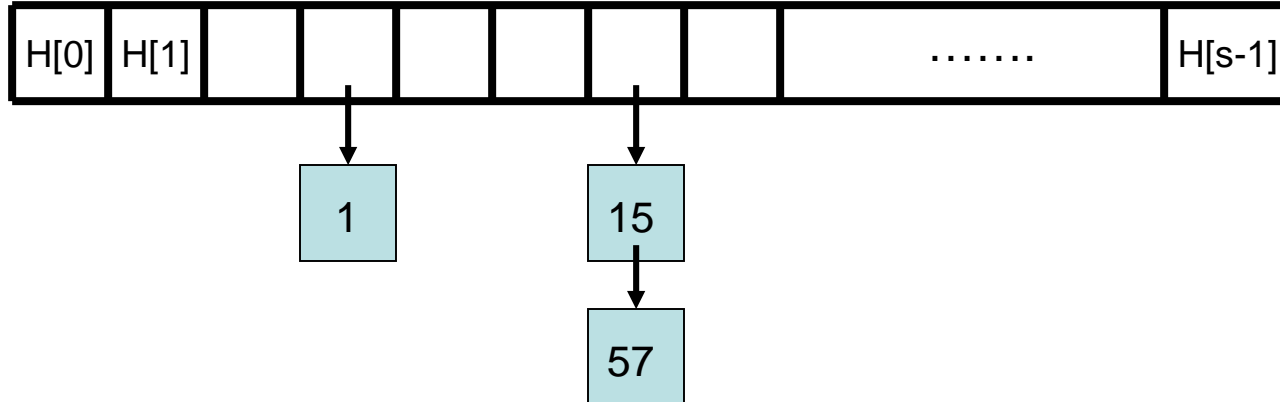
```
class listel {  
    int zahl;  
    listel *nachf; };  
  
liste = new listel;  
// dummy Element  
liste->nachf=NULL;
```

```
Insert(x):  
q = new listel;  
q->zahl=x;  
q->nachf =  
    liste->nachf;  
liste->nachf = q;
```

```
Delete(k):  
if (liste->nachf==NULL)  
    return NULL;  
q=liste;  
while (q->nachf!=NULL) {  
    if (q->nachf->zahl==k)  
        q->nachf=q->nachf->nachf;  
    else q=q->nachf; }  
}
```

Dictionary

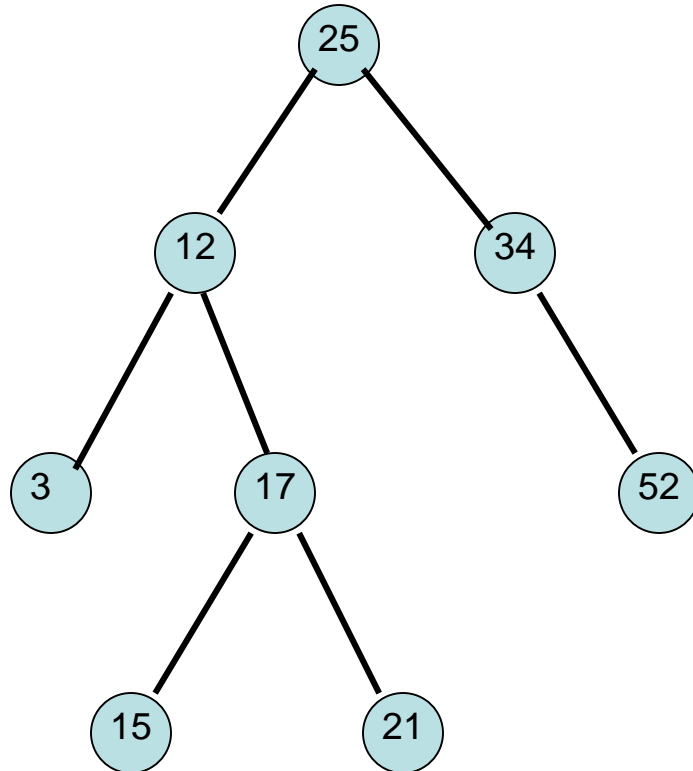
- Implementierung als Hashtabelle:



- p : genügend große Primzahl ($>$ Anzahl Elemente)
- H : Tabelle mit Pointern auf ungeordnete Listen
- h : Hashfunktion, $h(x) = ((a \cdot x + b) \bmod p) \bmod s$, p prim
- Regel: Element x in Liste $H[h(x)]$

Dictionary

- Implementierung als Splay-Baum:



Splay-Baum: binärer Suchbaum

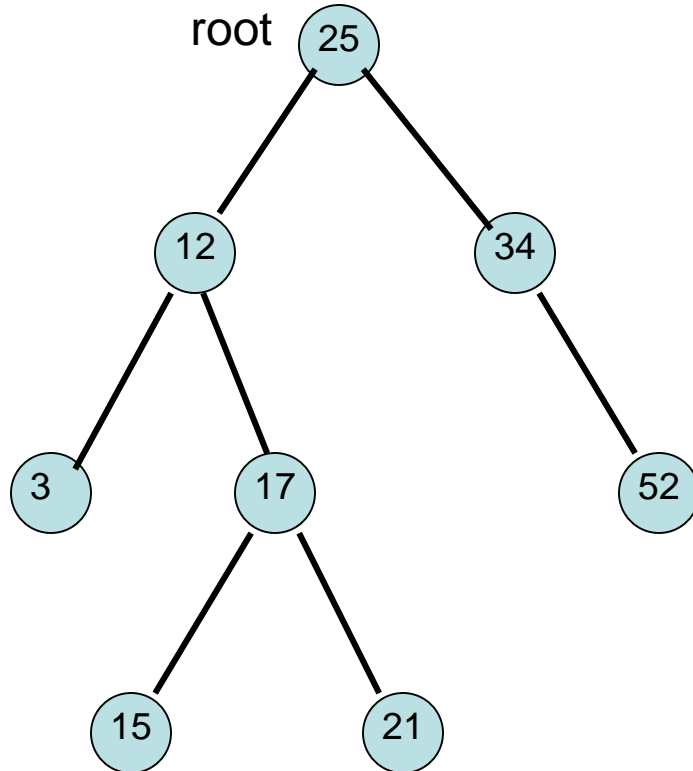
Binärer Suchbaum:

Für jeden Knoten v gilt:

- Anzahl Kinder ≤ 2
- Schlüssel im linken Teilbaum \leq Schlüssel(v) \leq Schlüssel im rechten Teilbaum

Dictionary

- Implementierung als Splay-Baum:



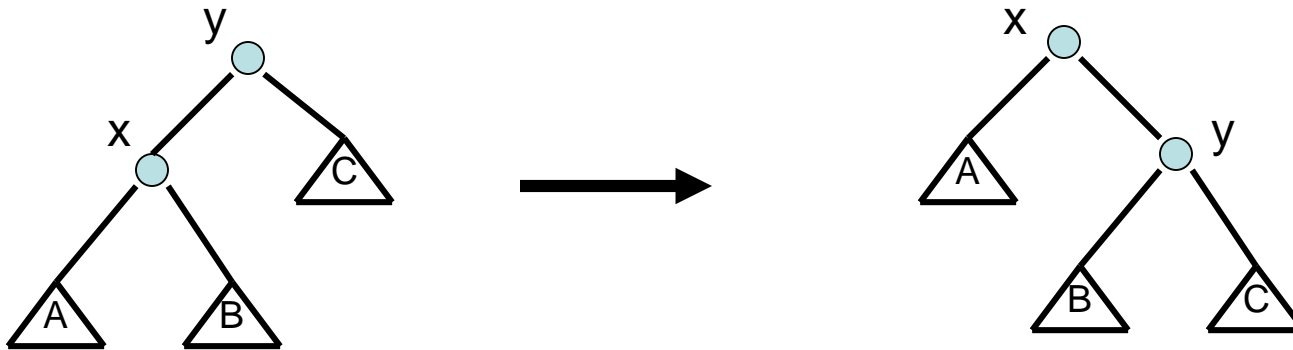
```
class node {  
    int zahl;  
    node *father;  
    node *lson, *rson;  
};
```

Search(k):

```
v := root;  
while (v!=NULL && v->zahl!=k)  
{ if (v->zahl>k) v=v->lson;  
    else v=v->rson;  
}  
return v;
```

Splay-Baum

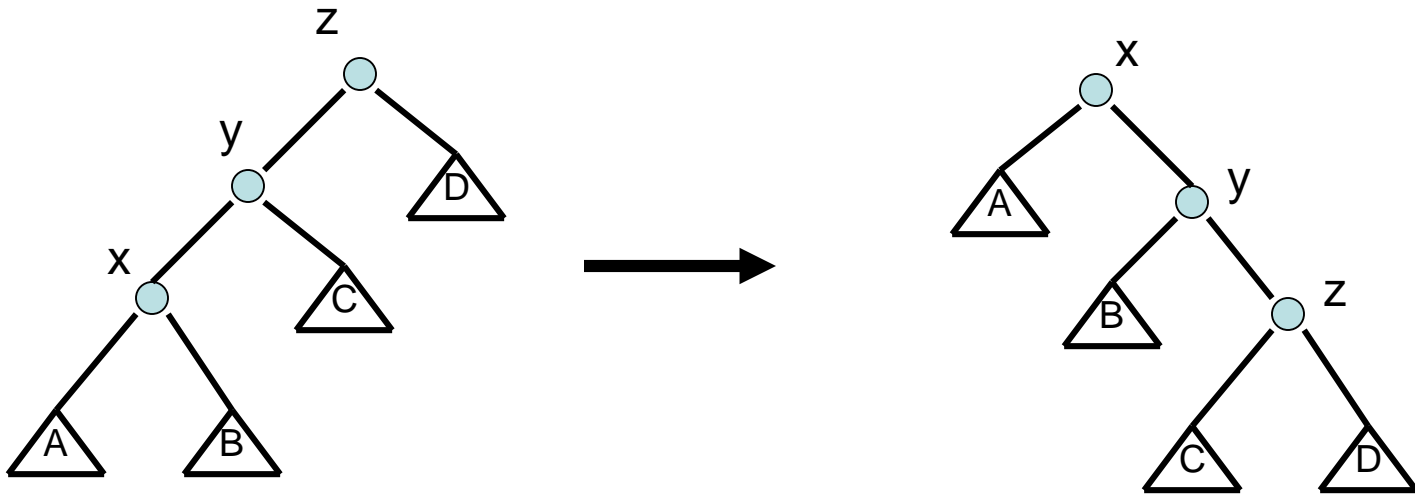
- Insert und Delete wie Search
- Balancierung des Suchbaums pro Search-Operation:
momentaner Knoten ist x
(1) x ist Kind der Wurzel



Fall x rechts von y analog

Splay-Baum

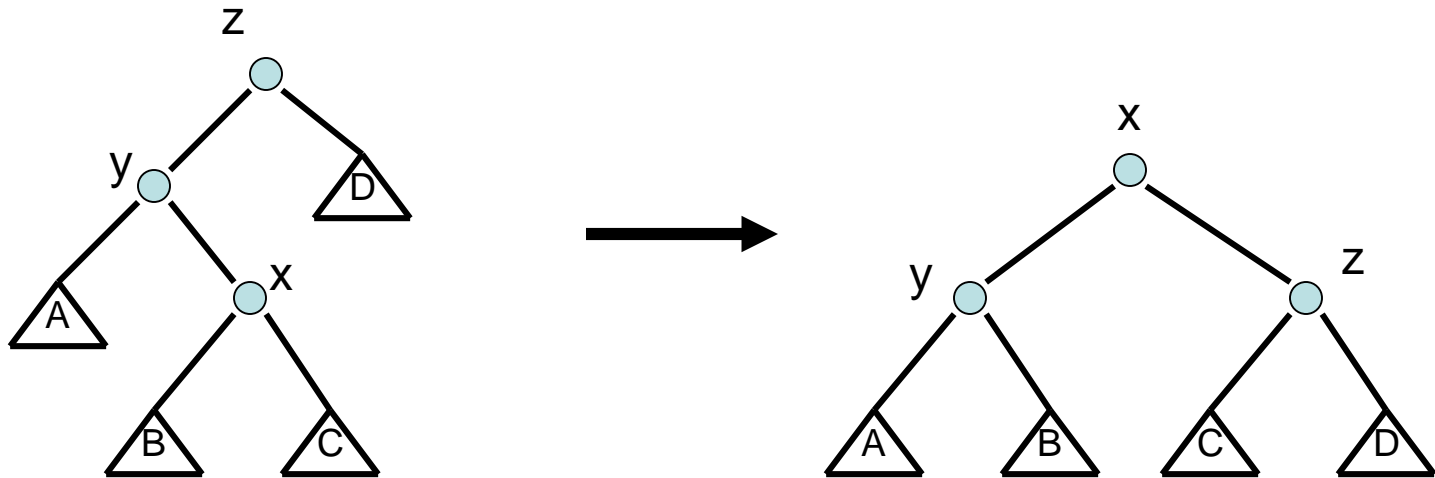
- (2) x hat Vater und Großvater rechts (bzw. links)



Fall x rechts von y und y rechts von z analog

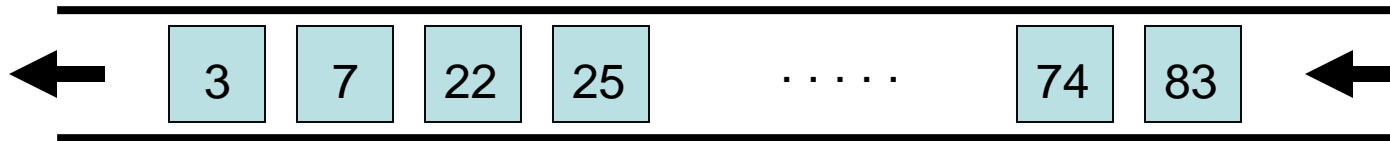
Splay-Baum

- (3) x hat Vater links und Großvater rechts



Fall x hat Vater rechts und Großvater links analog

Priority Queue

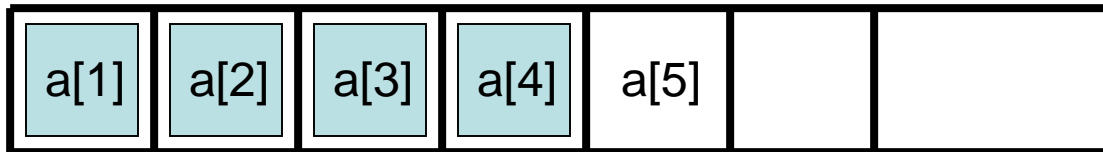


Operationen:

- $\text{Init}(q)$: erzeugt eine leere Priority Queue q
- $\text{Insert}(x, q)$: fügt x in die Priority Queue q ein
- $\text{Minimum}(q)$: gibt kleinsten Schlüssel zurück
- $\text{ExtractMin}(q)$: gibt kleinsten Schlüssel zurück und löscht ihn
- $\text{Empty}(q)$: testet, ob Priority Queue q leer ist
- $\text{Merge}(q, p)$: verschmelze q und p zu einer Priority Queue

Priority Queue

- Implementierung als Array:



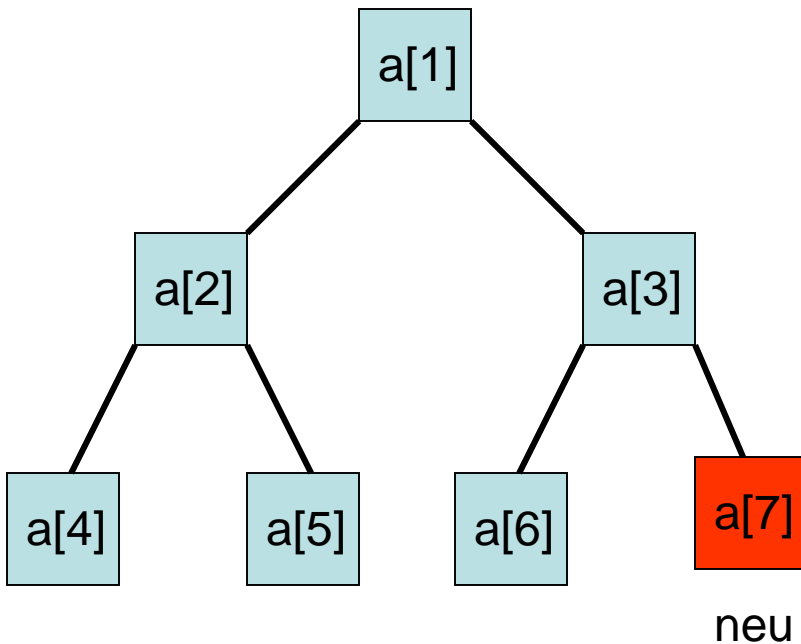
Regeln:

- bei n Einträgen sind Positionen $a[1]$ bis $a[n]$ besetzt ($a[0]: n$)
- für alle i gilt $a[i] \leq \min\{a[2i], a[2i+1]\}$ (Heap-Eigenschaft)

Dadurch ist minimaler Schlüssel immer an der Wurzel.

Minimum(q) ist also einfach.

Priority Queue



- size: Anzahl Einträge
- min(x,y): gibt $\min\{x,y\}$ zurück
- argmin(i,j): gibt $k \in \{i,j\}$ mit $a[k]=\min\{a[i],a[j]\}$ zurück

Insert(x):

```
size=size+1;
a[size]=x;
i = size;
while (i!=0 && a[i/2]>a[i]) {
    t=a[i]; a[i]=a[i/2]; a[i/2]=t;
    i=i/2;
}
```

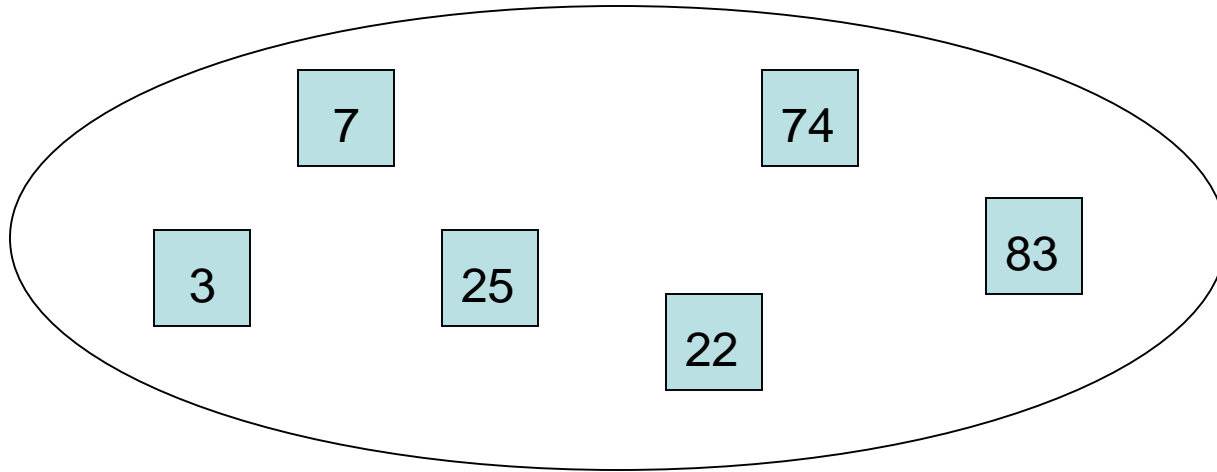
ExtractMin():

```
if (size==0) return -1;
x=a[1]; a[1]=a[size]; size--;
while (i<size) {
    if (size==2i && a[2i]<a[i])
        {a[2i]<->a[i]; i=2i;}
    if (size>2i && min(a[2i],a[2i+1])<a[i])
        { j = argmin(2i,2i+1);
          a[j]<->a[i]; i=j; } }
return x;
```

Priority Queue

- Merge-Operation teuer bei binärem Heap
- Bessere DS: Binomial-Heap
(wird in Vorlesung „Grundlegende Algorithmen“
vorgestellt)

Mengen

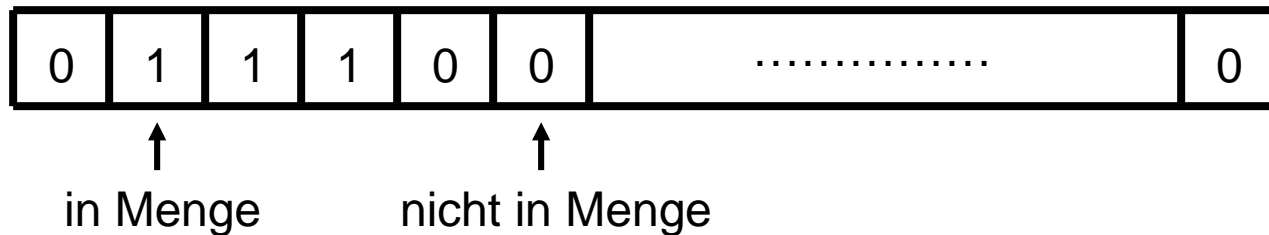


Operationen:

- $\text{Member}(x,S)$: testet, ob x ein Mitglied von S ist
- $\text{Union}(A,B)$: gibt die Vereinigung von A und B zurück
- $\text{Intersection}(A,B)$: gibt den Schnitt von A und B zurück
- $\text{Insert/Delete}(x,S)$: füge x in S ein / lösche x aus S

Mengen

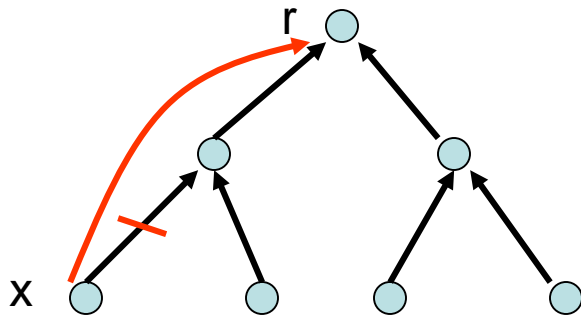
- Implementierung als Array: (vorzuziehen, falls Wertebereich klein)



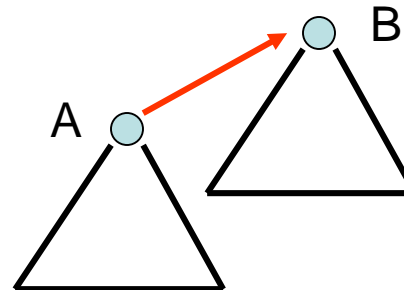
- Implementierung als Hashtabelle:
ähnlich zu Dictionary
- Implementierung als Suchbaum:
ähnlich zu Dictionary

Mengen

- Operationen:
 - $\text{Union}(A,B)$: gibt $A \cup B$ zurück
 - $\text{Find}(x)$: liefert Menge A von Element x
- Implementierung als Baum:
Menge identifiziert durch Wurzel des Baums



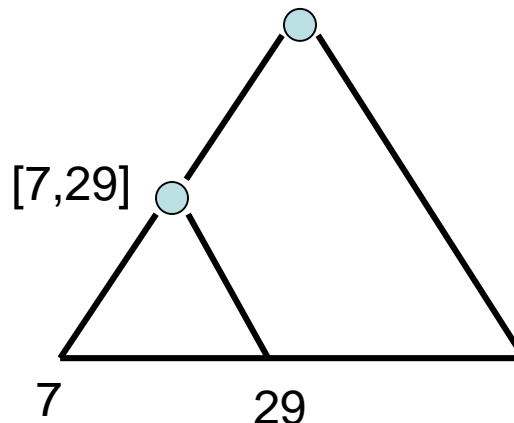
Find(x): biege Zeiger von x und Vorfahren auf r , gib r aus



Union(A,B): verbinde flacheren Baum (A) mit tieferem (B)

Mengen

- Operationen:
 - Intersection(A,B): gibt $A \cap B$ zurück
- Implementierung als Suchbaum:
Füge Intervall $[\min(T(v)), \max(T(v))]$ zu jedem Knoten v hinzu, wobei $T(v)$ den Teilbaum mit Wurzel v angibt.
Dann können über diese Intervalle effizient gemeinsame Elemente ermittelt werden.



Probleme

- 10038: Jolly Jumpers
- 484: The Department of Redundancy Department
- 443: Humble Numbers
- 496: Simply Subsets
- 10107: What is the Median?
- 1136: Help R2-D2!
- 10125: Sumsets

Hausaufgabe:

- 261: The Window Property