Vorrangswarteschlangen

- Elementare Implementierungen
- Binäre Heaps
- Index-Heaps
- Binomial Queues

Vorrangswarteschlangen

- Elementare Implementierungen
- Binäre Heaps
- Index-Heaps
- Binomial Queues

Manipulation von Mengen

Grundidee:

Abstrakte Spezifikation von Operationen auf einer Kollektion von Mengen (d.h. Spezifikation eines abstrakten Datentyps, ADT)

z.B. Wörterbuchproblem: Gegeben ist eine Menge von Elementen (Nutzdaten, value) und ihre Namen (Suchschlüssel, key), zusammen mit folgenden Operationen:

- search(key) - Suchen, Zugriff
- insert(key, value) - Einfügen
- delete(key) - Entfernen

Das Wörterbuchproblem ist ein Spezialfall des allgemeineren Problems der Manipulation von Mengen. Weitere, häufig in der Literatur beschriebene Spezialfälle:

- Vorrangswarteschlangen
- Union-Find-Probleme

ADT der Vorrangswarteschlange

Spezifikation der zugrundeliegenden Menge:

Die Vorrangswarteschlange (engl. Priority Queue) operiert auf einer Menge von Elementen, für die eine (Prioritäts-) Ordnung definiert ist, d.h. die Elemente müssen eine Vergleichsoperation unterstützen. Oft bestehen die Elemente wie bei Wörterbuchanwendungen aus einem numerischen Schlüssel und Nutzdaten.

Eine Vorrangswarteschlange unterstützt mindestens zwei grundlegende Operationen:

- Insert: Einfügen
- DeleteMin: Minimum entfernen

Damit ist die Vorrangswarteschlange eine Erweiterung der bekannten Strukturen *Stack* und *Queue*, die als Spezialfall aus diesem ADT hergeleitet werden können.

```
template <class Comparable>
class PriorityQueue
{
public:
  virtual void insert(const Comparable& x) = 0;
  // Fügt neues Element x ein. Elemente dürfen
  // i.A. doppelt vorkommen.

  virtual void deleteMin(Comparable& min) = 0;
  // Findet Minimum, schreibt es in min und
  // löscht es.
};
```

Komplexere Vorrangswarteschlangen

Viele Probleme erfordern zusätzliche Operationen auf Vorrangswarteschlangen.

Die komplette Schnittstelle umfaßt daher folgende Operationen:

- Insert: Einfügen eines Elementes
- DeleteMin: Minimum entfernen
- Build: Aufbau eine Vorrangswarteschlange aus *N* Elementen
- Change: Veränderung der Priorität eines beliebigen Elementes
- Remove: Entfernen eines beliebigen Elementes
- Merge: Verschmelzung zweier Vorrangswarteschlangen

Je nach Anwendung wird nur eine Teilmenge dieser Operationen gebraucht, für die jeweils eine unterschiedliche Implementierung sinnvoll ist. Bestimmte Anwendungen erfordern zusätzlich die Operation FindMin, ohne das Minimum zu löschen, andere ReplaceMin, die dann gesondert implementiert werden sollten.

Anwendungen von Vorrangswarteschlangen

- Scheduler in Betriebssystemen: Die Nutzdaten sind hier Jobs, das Prioritätskriterium ist z.B. die Laufzeit. Für schnelle Antwortzeiten erhalten kurze Jobs hohe Priorität, lange Jobs eine niedrige. Die Vorrangswarteschlange erlaubt effizientes Einfügen, Suchen und Entfernen des nächsten, auszuführenden Jobs mit der kürzesten Laufzeit. Zusätzlich kann sich die Priorität ändern und Jobs gelöscht werden.
- Grundlage für Heapsort (s. Programmiertechnik 2): Gebraucht wird ein schneller Aufbau der Vorrangswarteschlange und schnelles Finden und Entfernen des Minimums zur Sortierung.
- Implementierung der Kandidatenliste im Algorithmus von Dijkstra: Prioritätskriterium ist die Distanz zum Startknoten, die Nutzdaten sind die noch zu besuchenden Knoten des Graphen. Gebraucht wird Insert, DeleteMin und zusätzlich Change, wenn ein neuer, kürzerer Weg zu einem Knoten gefunden wird.
- Sog. Greedy-Algorithmen, bei deren Durchlauf wiederholt ein Minimum gefunden werden muß.
- In Kompressionsalgorithmen für Dateien

Elementare Implementierungen (1)

PriorityQueueArray.h

```
template <class Comparable>
class PriorityQueueArray : public PriorityQueue<Comparable>
{
public:
      PriorityQueueArray(int n = 100); // n: Größe des Feldes

      virtual void insert(const Comparable& x) {
            array[currentSize++] = x;
      }

      virtual void deleteMin(Comparable& min) {
            Comparable tmp;
            int minpos = 0;
            for(i=0;i<currentSize;i++)
                  if(array[i] < array[minpos]) minpos = i;
            tmp = array[minpos];
            array[minpos] = array[currentSize--];
            return tmp;
      }
private:
      int currentSize;
      vector<Comparable> array;
};
```

Beispiel: Implementierung als ungeordnetes Feld

Alternativen:

- geordnetes Feld
- ungeordnete (doppelt verkettete) Liste
- geordnete Liste

Elementare Implementierungen (2)

In Stacks und Queues lassen sich alle Operation in O(1) durchführen, in Vorrangswarteschlangen finden sich leicht Implementierungen, bei denen eine der beiden Basisoperationen O(1) ist, die andere O(N).

- Geordnete Felder/Listen erlauben DeleteMin in O(1), aber Insert ist O(N) (eager approach: Arbeit wird im voraus erledigt).
- Ungeordnete Felder/Listen erledigen Insert in O(1), aber DeleteMin in O(N) (lazy approach: Arbeit wird erst dann erledigt, wenn sie anfällt).
- Doppelt verkettete Listen erlauben Remove in O(1), brauchen aber mehr Speicherplatz und etwas aufwendigere Operationen

	insert	delMin	remove	change	merge
geordnetes Feld	N	1	N	N	N
geordnete Liste	N	1	1*	N	N
ungeordnetes Feld	1	N	1*	1*	N
ungeordnete Liste	1	N	1*	1*	1

* falls Position bekannt, sonst N wegen linearer Suche

Vorrangswarteschlangen

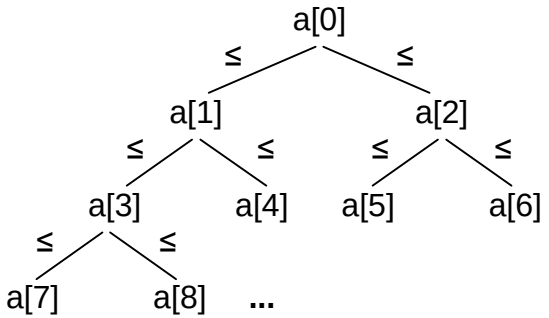
- Elementare Implementierungen
- Binäre Heaps
- Index-Heaps
- Binomial Queues

Binäre Heaps (Wiederholung aus Programmiertechnik 2)

Heapgeordnete Bäume:

Ein Baum ist heapgeordnet, wenn jeder Knoten kleiner oder gleich seinen Kindern ist.

Graphische Veranschaulichung:



Kein Knoten in einem heapgeordneten Baum ist kleiner als die Wurzel.

Vollständiger Binärbaum:

Jede Ebene des Binärbaums ist vollständig gefüllt, bis auf die letzte. Diese wird von links nach rechts gefüllt. Vollständige Binärbäume können sequentiell als Feld dargestellt werden, bei dem die Wurzel an Position 1, ihre Kinder an Pos. 2 und 3, die Enkel an Pos. 4, 5, 6, 7 usw. sind.

Heap:

Eine Menge von Knoten, die als vollständiger, heapgeordneter Binärbaum angeordnet und in einem Feld repräsentiert ist.

Operation Upheap („swim“)

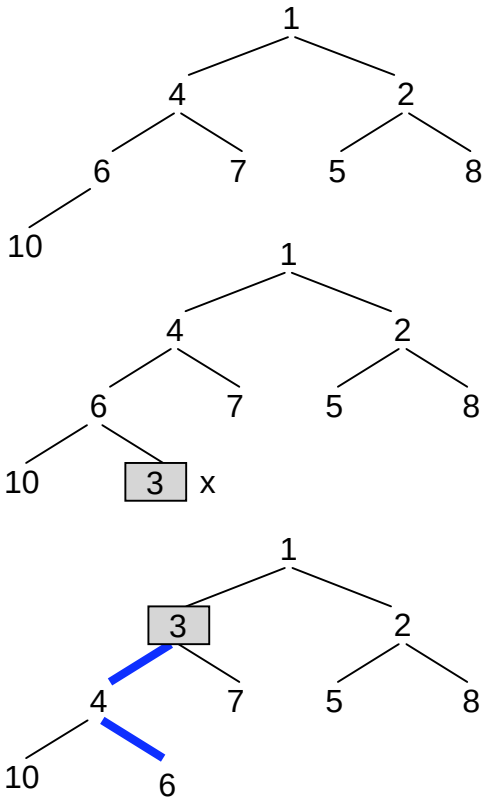
Insert: Einfügen eines neuen Elements

Ein neues Element wird an das Ende des Heaps abgespeichert.

Im allgemeinen ist dann die Heap-Bedingung (Eltern $\leq$ Kinder) verletzt.

Um die Bedingung zu reparieren wird das neue Element nach oben verschoben (upheap-Operation).

Beispiel:



Heap zu Beginn.

Neues Element x = 3 wird am Ende des Heaps eingefügt.

Heap-Bedingung ist verletzt.

Upheap: x = 3 wird nach oben verschoben.

Heap-Bedingung ist nun wieder erfüllt.

O(log N)

Operation Downheap („sink“)

DeleteMin: Löschen des kleinsten Elements

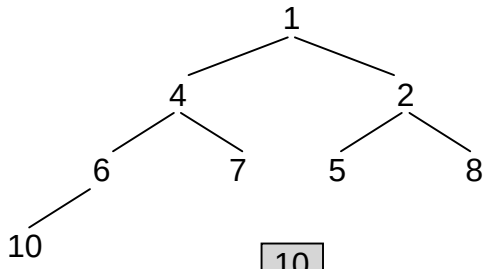
Das kleinste Element a[0] kann einfach gelöscht werden durch

```
a[0] = a[n-1];          // Überschreibe kleinstes Element mit letztem Element
n = n-1;
```

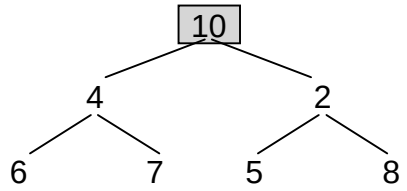
Im allgemeinen ist dann die Heap-Bedingung verletzt.

Um die Bedingung zu reparieren, wird a[0] nach unten verschoben (downheap-Operation).

Beispiel:



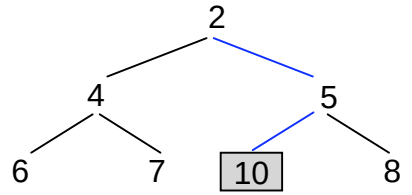
Heap zu Beginn.



1 wird gelöscht und mit letztem Element 10 überschrieben.

Heap-Bedingung ist verletzt.

O(log N)



Downheap: 10 wird nach unten in Richtung kleinere Kinder verschoben.

Heap-Bedingung ist nun wieder erfüllt.

Aufbau eines Heaps

Implementierung der Operation Build

Gegeben ist ein beliebiges Feld a (Elemente beliebig angeordnet).

Wie läßt sich a am effizientesten in einen Heap umbauen?

Algorithmus:

Wende auf letzten Nicht-Blatt-Knoten (hat den Index n/2 – 1; ganzzahlige Division) beginnend auf jeden Knoten eine downheap-Operation an, bis die Wurzel erreicht ist.

```
void build(int a[ ], int n)
// baut a in ein Heap um
{
      for (int i = n/2 - 1; i >= 0; i--)
            downheap(a, n, i);
}
```

O(N) (s. Programmiertechnik 2)

Begründung, warum der letzte Nicht-Blatt-Knoten den Index n/2 – 1 hat:

Letzter Nicht-Blatt-Knoten muss Elternknoten des letzten Blattes sein.

Letztes Blatt hat Index n-1.

Elternknoten hat daher den Index (n-2)/2 = n/2 -1 (ganzzahlige Division)

Vorrangswarteschlangen

- Elementare Implementierungen
- Binäre Heaps
- Index-Heaps
- Binomial Queues

Vollständige Vorrangswarteschlangen mit Heaps

Heaps erlauben eine effiziente Implementierung der Operationen Insert (O(log N)), DeleteMin (O(log N)) und Build (O(N)).

Die Implementierung vollständiger Vorrangswarteschlangen (d.h. zusätzliche Operationen Change, Remove, Merge) mit Heaps sind dagegen aufwendiger (O(N)). Im Fall von Change und Remove liegt der Grund darin, daß die Position eines Elements im Heap nicht bekannt ist. Daher muß das Feld linear durchsucht werden.

Für eine effiziente Implementierung von Change (z.B. für Dijkstra) und Delete muß daher eine Suchstruktur integriert werden.

Ansatz: Index-Heap

- Daten liegen in einem Feld außerhalb der Vorrangswarteschlange (client array).
- Der Heap verwaltet nur Indizes der Feldeinträge
- Zusätzlich wird die Position jedes Feldeintragindexs im Heap in einem Feld gespeichert (für Change und Remove)

Index-Heap (1)

PriorityQueueIndexHeap.h

```
template <class Comparable>
class PriorityQueueIndexHeap {
public:
      PriorityQueueIndexHeap(vector<Comparable>& a); // a ist die Adresse des client arrays
      void insert(int ind);          // bisher wurde hier ein Comparable& übergeben
      int deleteMin();               // gibt Index des gelöschten Minimums zurück
      void change(int ind);          // Achtung: Aufruf, nachdem Client Eintrag im client array
                                     // geändert hat

private:
      int currentSize;                        // wie bisher: Heapgröße
      vector<int> index;                      // Heap enthält Indizes der entsprechenden Einträge im
                                              // client array, nicht die Einträge selbst wie bisher
      vector<int> heapPos;                    // Position eines Index im Heap
      vector<Comparable>& clientArray;        // Zeiger zum Client array
      void swap(int i, int j) {               // vertausche 2 Knoten im Heap
            int tmp = index[i]; index[i] = index[j]; index[j] = tmp;
            heapPos[index[i]] = i; heapPos[index[j]] = j;
      }
      bool less(int i, int j) {          // Vergleich der durch i und j indizierten Client-Array-Einträge
            return clientArray[index[i]] < clientArray[index[j]];
      }
      void swim(int hole);               // Upheap
      void sink(int hole);               // Downheap
};
```

Index-Heap (2)

PriorityQueueIndexHeap.cpp

```
template <class Comparable>
void PriorityQueueIndexHeap<Comparable>::swim(int hole) {
      while(hole > 1 && less(hole, hole / 2)) {    // solange Knoten kleiner als Vorfahr u. Wurzel nicht
            swap(hole, hole/2);                    // erreicht, tausche Position mit Vorfahr
            hole = hole/2;
      }
}

template <class Comparable>
void PriorityQueueIndexHeap<Comparable>::insert(int ind) {
      index[currentSize++] = ind;
      heapPos[ind] = currentSize;
      swim(currentSize);
}

template <class Comparable>
void PriorityQueueIndexHeap<Comparable>::sink(int hole) {
      while(2*hole <= currentSize) { // solange Blattebene noch nicht erreicht ist
            int child = hole * 2;
            if( child < currentSize && less(child + 1, child) ) child++; // suche kleineres Kind
            if( ! less(hole, child) ) break;                  // Abbruch falls Heapbedingung erfüllt
            swap(hole, child);                                // falls nicht, tausche Position mit Kind
            hole = child;
      }
}
```

Index-Heap (3)

PriorityQueueIndexHeap.cpp

```
template <class Comparable>
int PriorityQueueIndexHeap<Comparable>::deleteMin() {
      swap(1, currentSize--);          // bringe Minimum an letzte Stelle
      sink(1);                         // stelle Heapordnung wieder her
      return index[currentSize + 1];   // gib Index des gelöschten Elements zurück
}

template <class Comparable>
void PriorityQueueIndexHeap<Comparable>::change(int ind) {
      swim(heapPos[ind]);    // Index des geänderten Elements schwimmt nach oben , falls kleiner
      sink(heapPos[ind]);    // sinkt nach unten, falls größer
}
```

main.cpp

```
:
vector<Comparable> a;
PriorityQueueIndexHeap<Comparable> pq(a);
:
// Remove-Operation: remove(17)
a[17] = minval;     // setze a[17] auf -∞
pq.change(17);      // update Index-Heap => Index 17 wird neues Minimum
pq.deleteMin();     // entferne Minimum
:
```

Vorrangswarteschlangen

- Elementare Implementierungen
- Binäre Heaps
- Index-Heaps
- Binomial Queues

Binomial Queues

	insert	delMin	remove	change	merge
Heap	log N	log N	N	N	N
Index-Heap	log N	log N	log N	log N	N

Index-Heaps erledigen auch Change und Delete in O(log N), aber Merge ist immer O(N).

- Eine effiziente Merge-Operation läßt sich nicht mit Felddarstellungen realisieren.
- Verkettete, ungeordnete Listen erlauben zwar Merge in O(1), aber DeleteMin ist O(N).
- Die Bedingungen der Heapordnung und Vollständigkeit für den Binärbaum verhindern eine effiziente Merge-Operation.

Kann man flexiblere Strukturen finden, die alle Operationen in O(log N) bewältigen, ohne daß eine vollständige Suchbaumstruktur implementiert werden muß?

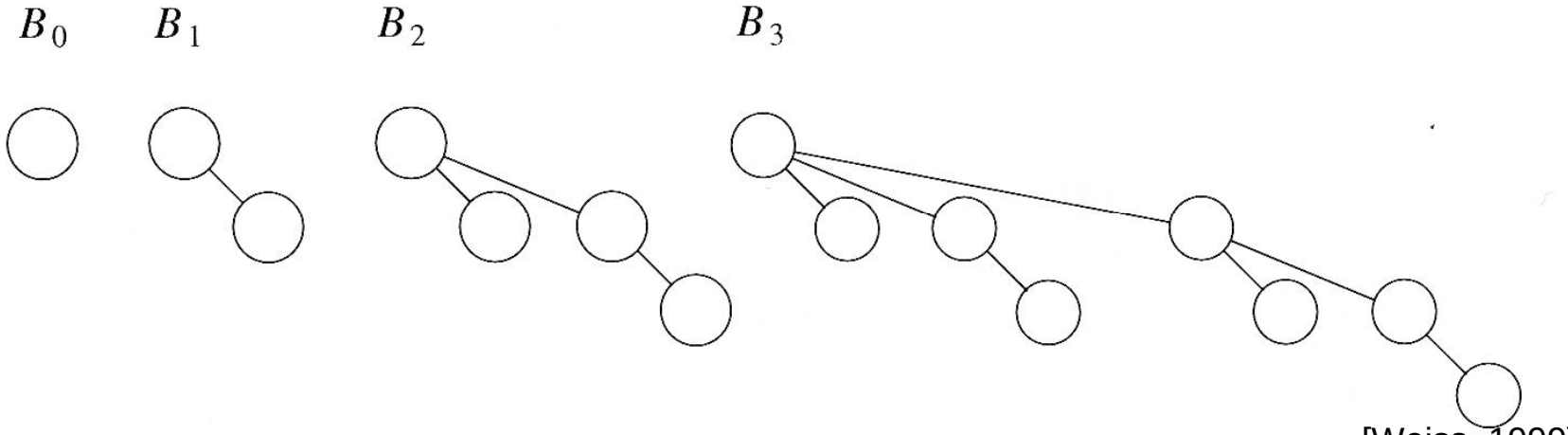
Antwort ist Ja: Binomial Queues erledigen alle Operationen der Vorrangswarteschlange in schlimmstenfalls O(log N), Insert im Mittel O(1), ohne daß die Suchbaumbedingung eingehalten werden muß.

Binomial Queues - Struktur (1)

- Binomial Queues sind eine Kollektion von Bäumen (d.h. ein Wald), die jeweils der Heap-Bedingung genügen: für jedes Kind z von p gilt p.priority < z.priority.

- Jeder der heapgeordneten Bäume hat die Struktur eines sogenannten Binomialbaums:

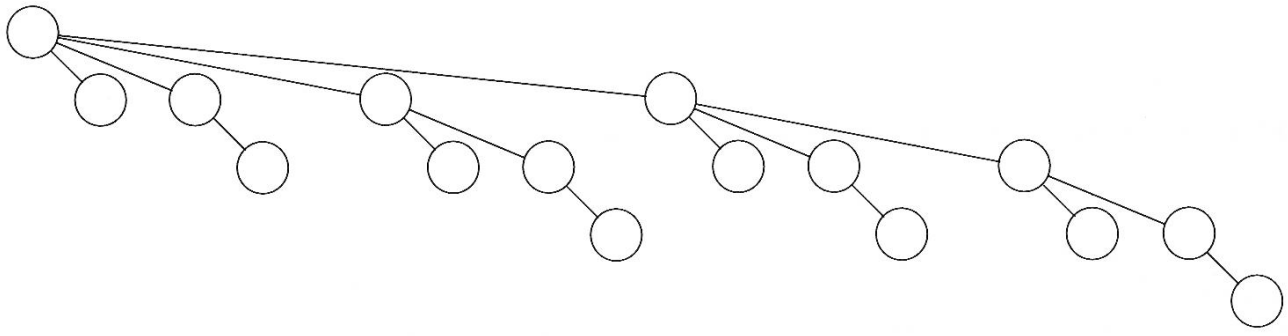
  - ein Binomialbaum der Höhe 0 ist ein Baum mit einem Knoten
  - ein Binomialbaum der Höhe k ensteht durch das Anhängen eines Binomialbaumes der Höhe k - 1 an die Wurzel eines anderen Binomialbaumes der Höhe k - 1.



[Weiss, 1999]

Binomial Queues - Struktur (2)

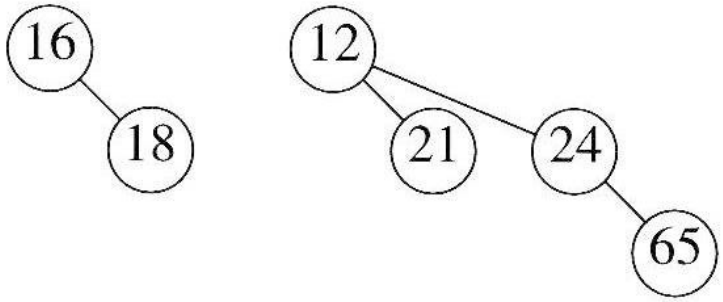
B_4



[Weiss, 1999]

- In einer Binomial Queue gibt es maximal einen Binomialbaum für jede Höhe.

H_1:



[Weiss, 1999]

Eigenschaften:

- Die Wurzel eines Binomialbaums der Höhe k hat k - 1 Kinder.
- Binomialbäume der Höhe k besitzen 2^k Knoten.
- Die Anzahl der Knoten in Tiefe d ist $\binom{k}{d}$.
- Jede Vorrangswarteschlange kann als Binomial Queue repräsentiert werden.

Operationen auf Binomial Queues (1)

- FindMin: Minimum wird durch Absuchen aller Wurzeln der Binomial Queue gefunden. Da es maximal log N unterschiedliche Bäume gibt, ist T(N) = O(log N).
- Alternativ läßt sich zusätzlich die Position des aktuellen Minimums speichern, Aktualisierung bei jeder Veränderung der Anzahl oder des Inhalts der Wurzeln. Hier gilt T(N) = O(1).

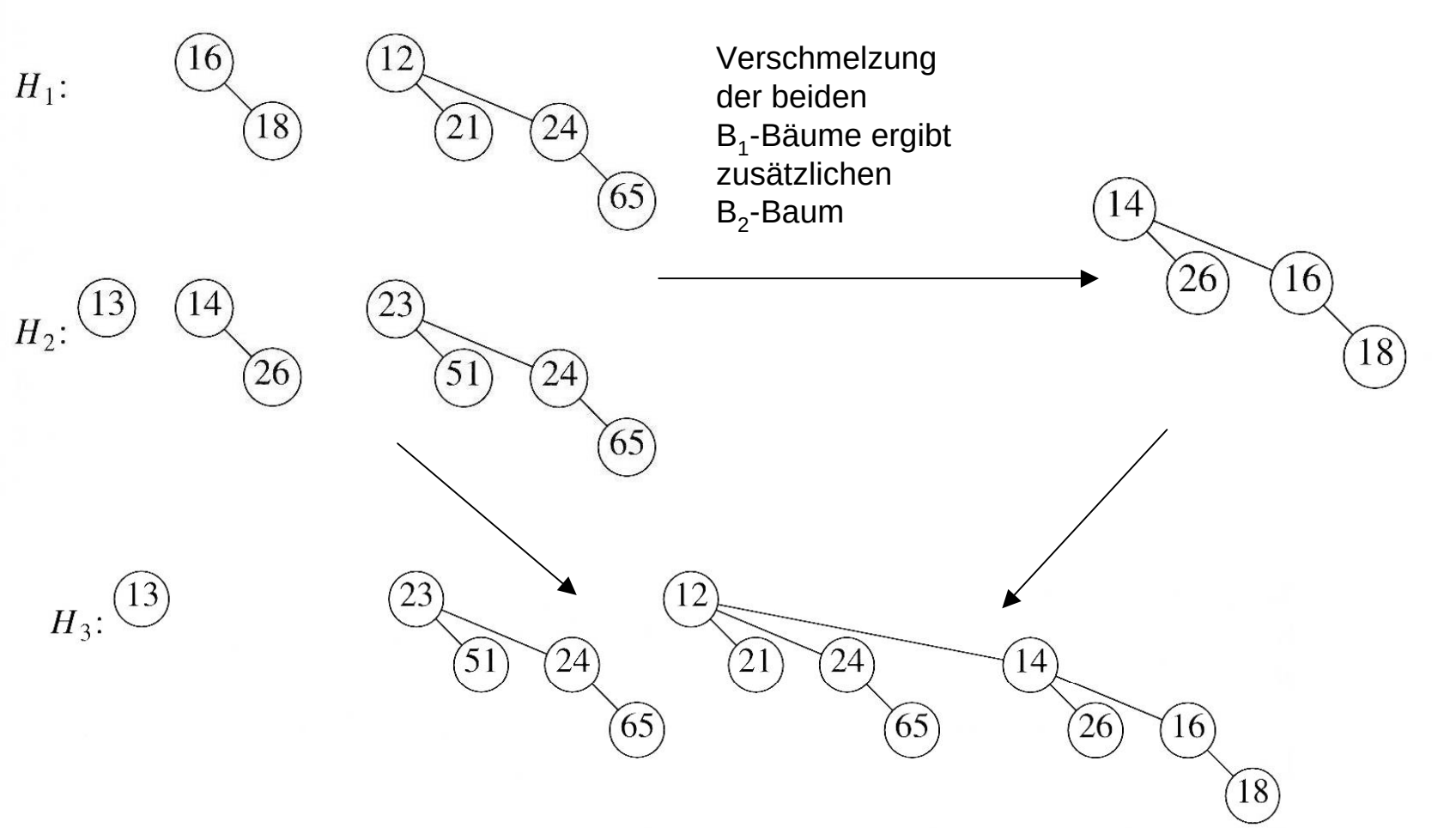
- Merge: „Addition zweier Binomial Queues“
  - Falls ein Baum der Höhe k nur in einer Queue vorkommt, wird er unverändert übernommen.

  - Kommt eine Baumhöhe in beiden Queues vor, wird die größere Wurzel als Unterbaum an die kleinere Wurzel gehängt.

  - Gibt es bereits einen Baum der resultierenden Höhe, werden auch diese verschmolzen.

  - Gibt es bereits zwei Bäume der resultierenden Höhe, wird einer beibehalten und zwei verschmolzen.

Operationen auf Binomial Queues (2)



[Weiss, 1999]

Da die Verschmelzung zweier Binomialbäume O(1) dauert und es maximal O(log N) Bäume gibt, ist die Merge-Operation O(log N).

Operationen auf Binomial Queues (3)

• Insert: Spezialfall von Merge, bei dem eine Queue mit einer einelementigen Queue vereinigt wird. Damit läuft auch diese Operation schlimmstenfalls in O(log n). Eine Analyse des Durchschnittsfalls ergibt sogar O(1).

• DeleteMin:
1. Aufsuchen der kleinsten Wurzel (O(log N))
2. Entfernen des zugehörigen Baums aus der Queue
3. Löschen der Wurzel des herausgenommenen Baums der Höhe k ergibt k - 1 Teilbäume in einer neuen Binomial Queue.
4. Beide resultierenden Queues werden über eine Merge-Operation in O(log N) vereinigt.

Insgesamt ergibt sich O(log N).

• Change: Da die Knoten in der Binomial Queue als verkettete Struktur von Objekten angelegt wird, kann bei jedem Insert ein Zeiger auf das Objekt übergeben werden, mit dem direkt auf das Element zugegriffen werden kann. Danach muß jeweils eine „swim“- und „sink“-Operation durchgeführt werden, um die Heapordnung wiederherzustellen, d.h. O(log N).

• Remove: 1. Change in -∞, 2. DeleteMin => O(log N).

Implementierung von Binomial Queues

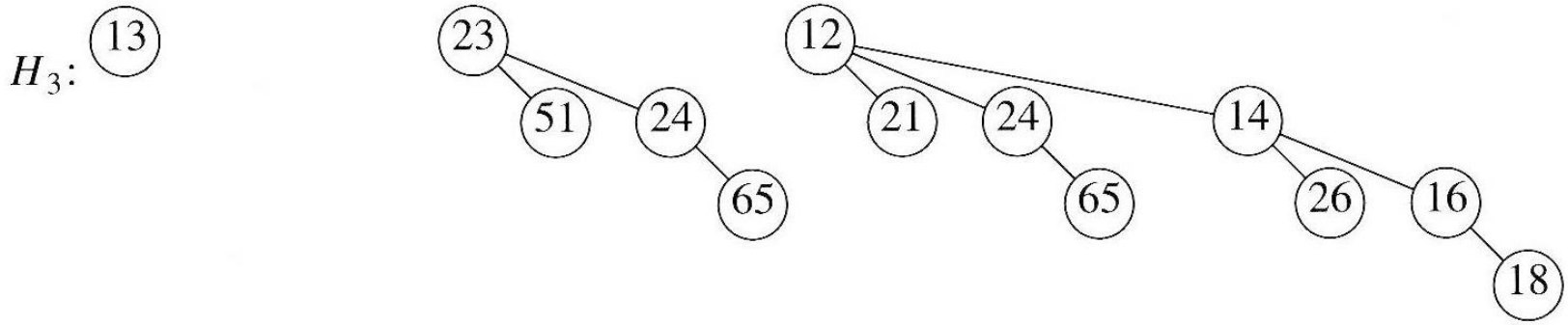
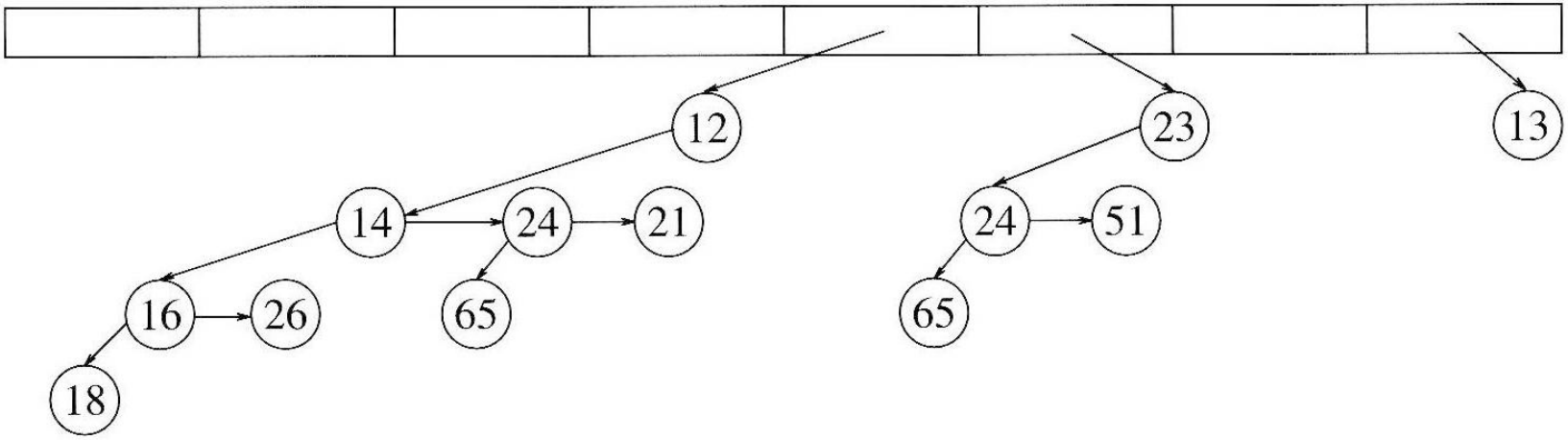


Figure 6.50 Binomial queue H_3 drawn as a forest



[Weiss, 1999]

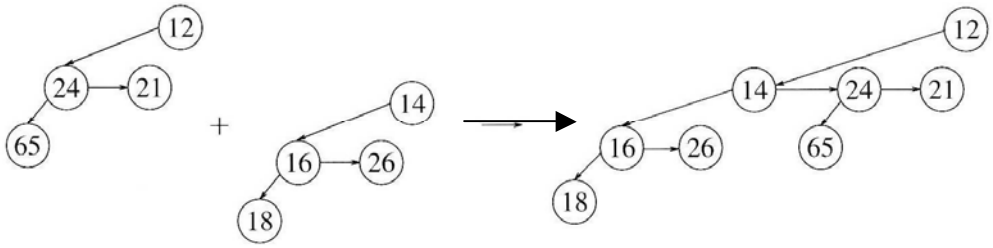
Binomial Queues: Klassenschnittstelle

```
template <class Comparable>
class BinomialQueue;

template <class Comparable>
class BinomialNode
{
    Comparable    element;
    BinomialNode *leftChild;
    BinomialNode *nextSibling;

    BinomialNode( const Comparable & theElement,
                  BinomialNode *lt, BinomialNode *nt )
      : element( theElement ), leftChild( lt ), nextSibling( nt ) { }
    friend class BinomialQueue<Comparable>;
};

template <class Comparable>
class BinomialQueue
{
  public:
    BinomialQueue( );
    BinomialQueue( const BinomialQueue & rhs );
    ~BinomialQueue( );

    bool isEmpty( ) const;
    bool isFull( ) const;
    const Comparable & findMin( ) const;

    void insert( const Comparable & x );
    void deleteMin( );
    void deleteMin( Comparable & minItem );
    void makeEmpty( );
    void merge( BinomialQueue & rhs );

    const BinomialQueue & operator=( const BinomialQueue & rhs );

  private:
    int currentSize;                // Number of items in the priority queue
    vector<BinomialNode<Comparable> *> theTrees;  // An array of tree roots

    int findMinIndex( ) const;
    int capacity( ) const;
    BinomialNode<Comparable> * combineTrees( BinomialNode<Comparable> *t1,
                                             BinomialNode<Comparable> *t2 ) const;
    void makeEmpty( BinomialNode<Comparable> * & t ) const;
    BinomialNode<Comparable> * clone( BinomialNode<Comparable> * t ) const;
};
```

[Weiss, 1999]

Binomial Queues: Merge (1)

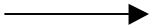
```
/**
 * Return the result of merging equal-sized t1 and t2.
 */
template <class Comparable>
BinomialNode<Comparable> *
BinomialQueue<Comparable>::combineTrees( BinomialNode<Comparable> *t1,
                                         BinomialNode<Comparable> *t2 ) con
{
    if( t2->element < t1->element )
        return combineTrees( t2, t1 );
    t2->nextSibling = t1->leftChild;
    t1->leftChild = t2;
    return t1;
}
```




```
/**
 * Merge rhs into the priority queue.
 * rhs becomes empty. rhs must be different from *this.
 * Throw Overflow if result exceeds capacity.
 */
template <class Comparable>
void BinomialQueue<Comparable>::merge( BinomialQueue<Comparable> & rhs )
{
    if( this == &rhs )    // Avoid aliasing problems
        return;

    if( currentSize + rhs.currentSize > capacity( ) )
        throw Overflow( );

    currentSize += rhs.currentSize;

    BinomialNode<Comparable> *carry = NULL;
    for( int i = 0, j = 1; j <= currentSize; i++, j *= 2 )
    {
        BinomialNode<Comparable> *t1 = theTrees[ i ];
        BinomialNode<Comparable> *t2 = rhs.theTrees[ i ];
        int whichCase = t1 == NULL ? 0 : 1;
        whichCase += t2 == NULL ? 0 : 2;
        whichCase += carry == NULL ? 0 : 4;

        switch( whichCase )
        {
          case 0: /* No trees */
          case 1: /* Only *this */
            break;
          case 2: /* Only rhs */
            theTrees[ i ] = t2;
            rhs.theTrees[ i ] = NULL;
            break;
          case 4: /* Only carry */
            theTrees[ i ] = carry;
            carry = NULL;
            break;
```

[Weiss, 1999]

Binomial Queues: Merge (2)



```
            case 3: /* *this and rhs */
              carry = combineTrees( t1, t2 );
              theTrees[ i ] = rhs.theTrees[ i ] = NULL;
              break;
            case 5: /* *this and carry */
              carry = combineTrees( t1, carry );
              theTrees[ i ] = NULL;
              break;
            case 6: /* rhs and carry */
              carry = combineTrees( t2, carry );
              rhs.theTrees[ i ] = NULL;
              break;
            case 7: /* All three */
              theTrees[ i ] = carry;
              carry = combineTrees( t1, t2 );
              rhs.theTrees[ i ] = NULL;
              break;
        }
    }

    for( int k = 0; k < rhs.theTrees.size( ); k++ )
        rhs.theTrees[ k ] = NULL;
    rhs.currentSize = 0;
}
```

[Weiss, 1999]