Universität des Saarlandes
FR 6.2 Informatik
Prof. Dr. Hans-Peter Lenhof
Dr. Oliver Kohlbacher
Dipl. Inform. Andreas Hildebrandt

Programmierung II, SS 2003
Musterlösung zur 10. Übung

Aufgabe 1: Amortisierte Analyse (10 Punkte)
Beim Einfügen und Löschen von Knoten kann beim 2–5 - Baum ein Knoten mit 6 Kindern oder einer mit nur einem Kind entstehen. Diese Probleme können wir mit den drei Rebalancierungsoperationen *Spalten*, *Stehlen* und *Verschmelzen* beseitigen.
Beim *Spalten* wird ein 6-er Knoten durch zwei 3-er Knoten ersetzt, wodurch offenbar die Kindanzahl des dazugehörigen Vaterknotens um eins erhöht wird. Falls dieser Vaterknoten nun dadurch zum 6-er Knoten wird, wird auch er aufgespalten usw. Dies kann sich bis zur Wurzel hochziehen. Falls wir nun die Wurzel selbst aufspalten müssen setzen wir eine neue Wurzel darüber und sind fertig.
Wie können wir 1-er Knoten reparieren? Dies hängt vom Grad der Geschwisterknoten (d.h. dem entweder linken oder rechten Nachbarn des aktuellen Knotens) ab. Falls das Geschwister ein 2-er Knoten ist, dann *verschmelzen* wir die beiden Knoten zu einem 3-er Knoten. Dadurch verringert sich der Grad des Vaters um eins, und wir müssen unter Umständen wieder bis zur Wurzel hoch rebalancieren. Ist der Geschwisterknoten kein 2-er Knoten, so können wir ihm einfach ein Kind *stehlen* und an den 1-er Knoten anhängen. In diesem Fall sind wir offenbar schon mit der Rebalancierung fertig und müssen nichts mehr nach oben propagieren.

Da sich die Operationen Spalten und Verschmelzen bis zur Wurzel hochziehen können, liegt ihre Laufzeit offenbar in $\mathcal{O}(\log_2(n))$ (denn wir hatten ja auf dem letzten Blatt gezeigt, dass die Höhe des Baumes in $\Theta(\log_2(n))$ liegt). Wenn wir nun n mal Knoten einfügen oder löschen, so könnte man erwarten dass wir eine Laufzeit von $\mathcal{O}(n \log_2(n))$ Rebalancierungsoperationen erhalten könnten. Tatsächlich ist jedoch die worst-case Laufzeit einer solchen Sequenz *erheblich* besser. Daher zeigen wir nun den folgenden

Satz: Wenn man mit einem leeren 2–5 - Baum startet und n Einfüge- oder Löschoperationen durchführt, dann beträgt die Gesamtzahl der Rebalancierungsoperationen höchstens $2n$.
Beweis: Offenbar liegt jede einzelne Rebalancierung in $\mathcal{O}(1)$. Daher können wir zur Vereinfachung die Rebalancierungen als Elementaroperationen mit Kosten von 1 betrachten und die Bankkontomethode zur Analyse der Laufzeit verwenden.
Wir zeigen nun: wenn wir für jedes Einfügen oder Löschen 2 Recheneinheiten einzahlen, dann können wir mit den auf unserem Konto gespeicherten unbenutzten Recheneinheiten alle Rebalancierungen bezahlen. Dazu postulieren und zeigen wir eine Invariante, die zeigt wie sich unser Kontostand zusammensetzt:

Behauptung: Für jeden Knoten des Baumes liegen auf unserem Konto unbenutzte Recheneinheiten gemäß der folgenden Tabelle:

Grad	1	2	3	4	5	6
RE	3	1	0	0	1	3

Beweis: Hinzufügen eines Knotens erhöht den Grad um eins, Löschen vermindert ihn um eins. Wenn wir uns die Tabelle ansehen, so stellen wir fest, dass wir mit den 2 Recheneinheiten die wir für jedes Einfügen oder Löschen zusätzlich zu den schon auf dem Konto gespeicherten Einheiten ausgeben dürfen, die Invariante aufrechterhalten können, denn wir müssen maximal 2 Einheiten einzahlen (und das auch nur, wenn wir von 2 nach 1 oder von 5 nach 6 gehen).
Was passiert nun beim Rebalancieren? Beim *Spalten* wird ein 6-er Knoten zu zwei 3-er Knoten. Es fällt also ein 6-er Knoten weg, und damit werden die für ihn gespeicherten 3 Recheneinheiten frei. Die beiden neuen 3-er Knoten benötigen laut Tabelle überhaupt keine Einheiten, also haben wir nun 3 RE zur Verfügung. Davon müssen wir maximal 2 für das Verändern unseres Vaterknotens aufwenden, nämlich wenn dieser von einem 5-er zu einem 6-er Knoten wird (alle anderen Fälle sind ja "billiger"). Wir haben also auf jeden Fall noch 1 RE übrig, und mit dieser bezahlen wir das Spalten selbst[1].

Analog argumentieren wir im Falle des *Verschmelzens* eines 1-er und eines 2-er Knotens: auf dem Konto liegen 4 RE für diese beiden Knoten bereit, nämlich 3 für den 1-er und 1 für den 2-er. Wir erzeugen einen 3-er Knoten, der selbst keine RE benötigt (laut Tabelle) und vermindern den Grad des Vaterknotens um eins. Dabei müssen wir wieder maximal 2 RE für die Änderung des Vaters bezahlen und haben noch 2 RE zur freien Verfügung. Eine davon geben wir aus um für das Verschmelzen selbst zu bezahlen und eine lassen wir unbenutzt.

Am einfachsten ist natürlich das *Stehlen*. Wir werden auf diese Art einen 1-er Knoten los, der seine 3 RE vom Konto abheben kann, und erzeugen aus ihm einen 2-er Knoten, der von diesen RE eine benötigt. Ausserdem haben wir entweder einen 3-er Knoten zu einem 2-er verändert (kostet 1 RE), einen 4-er zu einem 3-er (kostet gar nichts) oder einen 5-er zu einem 4-er (da wird sogar eine RE frei). Dies kostet uns also maximal eine RE. Damit haben wir noch eine RE übrig mit der wir für das Sthelen selbst bezahlen können.

Und damit sind wir auch schon fertig. Wir konnten zeigen, dass wir den Kontostand der obigen Tabelle aufrechterhalten können, wenn wir für jedes Einfügen und Löschen 2 RE berechnen, und dass wir mit diesem Kontostand alle auftretenden Rebalancierungen bezahlen können. Da wir n Einfüge- oder Löschoperationen durchführen und da jede 2 RE kostet, erhalten wir die gesuchten Gesamtkosten von $2n$. ■

Aufgabe 2: d-näre Heaps (10 Punkte)
Die d-nären Heaps sind den in der Vorlesung vorgestellten Heaps sehr ähnlich.

(a) Die Repräsentation läuft analog der für binäre Heaps: wir speichern Ebene für Ebene von links nach rechts ab. Wenn wir das Array $A := [a_0, \ldots, a_{n-1}]$ dafür verwenden, dann kommen wir folgendermassen zum i-ten Kind eines Knotens mit Index k:

[1] Wir haben also gerade gezeigt, dass wir mit den auf dem Konto vorhandenen Recheneinheiten alle beim Spalten eines Knotens auftretenden Elementaroperationen bezahlen können! Dabei entsteht u.U. ein 6-er - Vaterknoten, der nun selbst wieder für sein eigenes Spalten bezahlen kann usw.

nehmen wir an, k sei der m-te Knoten auf Ebene l. Dann sind *vor* k die $l-1$ letzten Ebenen gespeichert, d.h. schon mal $\sum_{i=0}^{l-1} d^i$ Elemente. Da k das m-te Kind sein soll, und da wir die Indizierung des Arrays bei der 0 beginnen, bedeutet das, dass $k = \sum_{i=0}^{l-1} d^i + m - 1 = d^1 + \cdots + d^{l-1} + m$. Um zu seinem i-ten Kind zu kommen, müssen wir nun zuerst die l-te Ebene finden, die offenbar beim Index $d^1 + \cdots + d^l$ beginnt, und noch die jeweils d Kinder der $m-1$ linken Geschwisterknoten von k überspringen. Wir landen also bei $d^1 + \cdots + d^l + (m-1)d + i$. Dies können wir vereinfachen zu:

$$\begin{aligned} d^1 + \cdots + d^l + (m-1)d + i &= d + d(d^1 + \cdots + d^{l-1}) + md - d + i \\ &= d(d^1 + \cdots + d^{l-1} + m) + i \\ &= kd + i \end{aligned}$$

Also finden wir das Kind des Knotens `A[k]` an der Position `A[kd + i]`. Dies können wir auch leicht umkehren: den Vater eines Knotens mit Index k erhalten wir durch `A[`$\lfloor\frac{k-1}{d}\rfloor$`]`.

(b) In der Ebene 0 ist 1 Knoten gespeichert, in der Ebene 1 sind d Knoten gespeichert, ..., in der Ebene $h_d - 1$ sind d^{h-1} Knoten gespeichert und in der letzten Ebene (h_d) sind - je nach Füllstand des Heaps - zwischen 1 und d^h Elemente gespeichert. Es gilt also:

$$\begin{aligned} & 1 + d^1 + \cdots + d^{h_d - 1} + 1 \leq n \leq 1 + d + \cdots + d^{h_d} \\ \Leftrightarrow & \sum_{i=0}^{h_d - 1} d^i + 1 \leq n \leq \sum_{i=0}^{h_d} d^i \\ \Leftrightarrow & \frac{d^{h_d} - 1}{d-1} + 1 \leq n \leq \frac{d^{h_d+1} - 1}{d-1} \\ \Rightarrow & d^{h_d} \leq n(d-1) + 1 \leq d^{h_d+1} \\ \Leftrightarrow & h_d \leq \log_d(n(d-1)+1) \leq h+1 \\ \Rightarrow & h_d \in \Theta(\log_d(n)) \blacksquare \end{aligned}$$

(denn $\log_d(nd) = \log_d(n) + log_d(d) = \log_d(n) + 1$).

(c) Wenn man sich die Implementierung von `insert` bei binären Heaps ansieht, dann stellt man fest, dass nirgendwo davon Gebrauch gemacht wird, dass es sich um einen binären Baum handelt. Das einzige, was angepasst werden muss, ist die Funktion, die den Index des Vaterknotens zurückliefert. Diese haben wir im ersten Teil schon bestimmt und sind damit damit fertig. Die Laufzeit ist damit – genau wie bei den Binärheaps – in $\mathcal{O}(h_d(n))$ und damit nach Teil 2 in $\mathcal{O}(\frac{\log_2(n)}{\log_2(d)}) = \mathcal{O}(\log_2(n))$.

Aufgabe 3: Binärheaps (10 Punkte)
Bei Interpretation als max-Heap lautet das Ergebnis $[84, 22, 19, 10, 3, 17, 6, 5, 9]$, bei Interpretation als min-Heap $[3, 5, 6, 9, 84, 19, 17, 22, 10]$.
Teil (b):

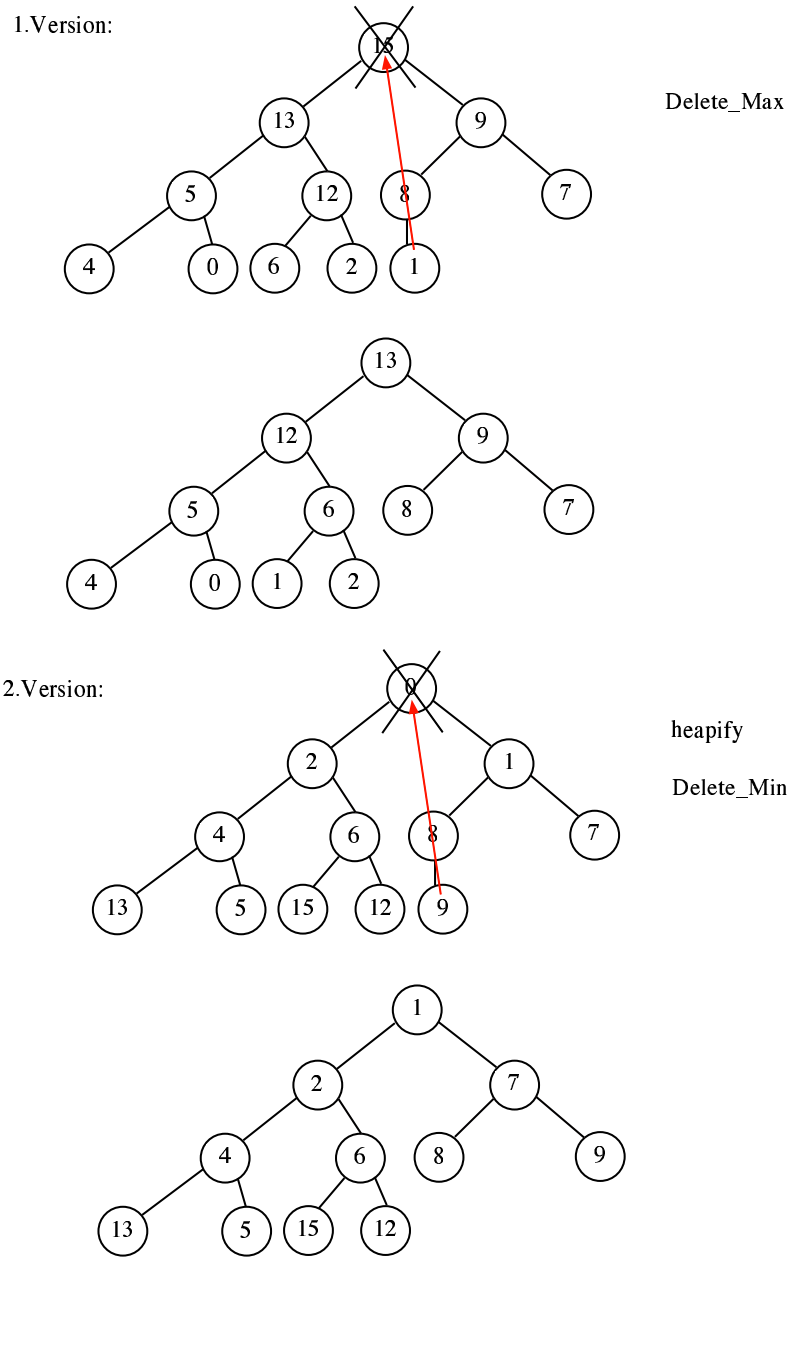

Abbildung 1: (Teil b)

Aufgabe 4: Binärheaps, Implementierung (10 Punkte)
Der folgende Code befindet sich in der Datei `BinaryHeap.h`:

```
#include <vector>
#include <algorithm>

using namespace std;

// Die Heap-Klasse
template <typename Key, typename Inf>
class BinaryHeap
{
  public:
    // Wir wollen uns ein wenig Tipparbeit sparen und den Code
    // lesbarer machen...
    typedef pair<Key, Inf> Item;

    // Der Defaultkonstruktor erzeugt einen leeren Heap
    BinaryHeap();

    // Dieser Konstruktor nimmt einen vector<Item> und bastelt daraus
    // einen passenden Heap
    BinaryHeap(const vector<Item>& data);

    // Dieser Operator erlaubt den transparenten Zugriff auf die im
    // intern verwendeten Array enthaltenen Daten und kann natuerlich
    // die Heap - Eigenschaft zerstoeren!
    Item& operator [] (int i);

    // Den Zugriffs-Operator brauchen wir immer auch in einer const - Variante,
    // ansonsten koennen wir ihn in einer Variable "const BinaryHeap<T, T2> bla"
    // nicht verwenden. In diesem Operator kann der Heap - Eigenschaft natuerlich
    // nichts geschehen.
    const Item& operator [] (int i) const;

    // Anfuegen eines Elementes an das intern verwendete Array. Diese
    // Funktion kann die Heap - Eigenschaft zerstoeren!
    void push_back(const Item& item);

    // Entfernen des letzten Blatts.
    void pop_back();

    // Ueberprueft, ob der Heap leer ist
    bool empty() const;

    // Gibt die Groesse des Heaps zurueck
    size_t size() const;
```

```
    // push_up schiebt ein Element nach oben
    void push_up(int i);

    // push_down schiebt ein Element nach unten
    void push_down(int i);

    // Gibt den Index des Vaterknotens zurueck. Falls i schon die Wurzel ist, wird
    // -1 zurueckgeliefert.
    int parent_index(int i) const;

    // Gibt den Index des linken Kindes zurueck. Falls i kein linkes Kind besitzt
    // wird -1 zurueckgeliefert.
    int left_child(int i) const;

    // Gibt den Index des rechten Kindes zurueck. Falls i kein rechtes Kind besitzt
    // wird -1 zurueckgeliefert.
    int right_child(int i) const;

    // Macht aus data_ einen Heap.
    void heapify();

    std::vector<Item>& data();
    const std::vector<Item>& data() const;

  protected:

    // Das Array, mit dem wir unseren Heap simulieren.
    vector<Item> data_;
};

// Die Implementierung unserer Methoden

// Defaultkonstruktor.
template <typename Key, typename Inf>
BinaryHeap<Key, Inf>::BinaryHeap()
{
}

// Alternativer Konstruktor.
template <typename Key, typename Inf>
BinaryHeap<Key, Inf>::BinaryHeap(const vector<Item>& data)
  : data_(data)
{
  // Wir muessen das Heap-basteln ja nicht doppelt implementieren und ziehen uns
  // deshalb auf die Methode create_heap() zurueck
  heapify();
}
```

```
// Die Anfuege-Funktion
template <typename Key, typename Inf>
void BinaryHeap<Key, Inf>::push_back(const Item& item)
{
  data_.push_back(item);
}

// Die Anfuege-Funktion
template <typename Key, typename Inf>
void BinaryHeap<Key, Inf>::pop_back()
{
  data_.pop_back();
}

// Durchreichen der Daten
template <typename Key, typename Inf>
typename BinaryHeap<Key, Inf>::Item& BinaryHeap<Key, Inf>::operator [] (int i)
{
  return data_[i];
}

// Durchreichen der Daten, const - Version
template <typename Key, typename Inf>
const typename BinaryHeap<Key, Inf>::Item& BinaryHeap<Key, Inf>::operator [] (int i) con
{
  return data_[i];
}

// Durchreichen der Daten, const - Version
template <typename Key, typename Inf>
const typename std::vector<typename BinaryHeap<Key, Inf>::Item>& BinaryHeap<Key, Inf>::d
{
  return data_;
}

// Durchreichen der Daten, non-const - Version
template <typename Key, typename Inf>
typename std::vector<typename BinaryHeap<Key, Inf>::Item>& BinaryHeap<Key, Inf>::data()
{
  return data_;
}

// Leerer Heap?
template <typename Key, typename Inf>
bool BinaryHeap<Key, Inf>::empty() const
{
  return data_.empty();
```

```
}

//  Groesse des Heaps
template <typename Key, typename Inf>
size_t BinaryHeap<Key, Inf>::size() const
{
  return data_.size();
}

// Vaterindex
template <typename Key, typename Inf>
int BinaryHeap<Key, Inf>::parent_index(int i) const
{
  // Sind wir an der Wurzel? Dann gibt's natuerlich keinen Vaterknoten
  if (i == 0)
  {
    return -1;
  }

  // OK. Da ja fuer einen Knoten mit Index i gilt: linkes Kind = 2i+1, rechtes Kind = 2i
  // koennen wir den Vater eines Knotens mit Index j einfach bestimmen durch Abrunden
  // von (j-1)/2. Da C++ Divisionen bei Integerzahlen automatisch abrundet muessen wir
  // uns darum auch nicht mehr kuemmern.
  return (i-1)/2;
}

// Linkes Kind
template <typename Key, typename Inf>
int BinaryHeap<Key, Inf>::left_child(int i) const
{
  int child_index = 2*i+1;

  if (child_index >= (int) size())
  {
    return -1;
  }

  return child_index;
}

// Rechtes Kind
template <typename Key, typename Inf>
int BinaryHeap<Key, Inf>::right_child(int i) const
{
  int child_index = 2*i+2;

  if (child_index >= (int) size())
  {
```

```
    return -1;
  }

  return child_index;
}

// Die push_up Funktion vertauscht ein Element mit dem "darueberliegenden" Element
template <typename Key, typename Inf>
void BinaryHeap<Key, Inf>::push_up(int i)
{
  int j = parent_index();

  // Wenn wir schon ganz oben angekommen sind, machen wir nichts.
  if (j != -1)
  {
    Item tmp = data_[i];
    data_[i] = data_[parent_index];
    data_[parent_index] = tmp;
  }
}

// Die push_down - Funktion verschiebt ein Element von oben nach unten an seinen
// Platz im Heap. Hier gilt es zu beachten, dass die in data_ gespeicherten Werte
// vom Typ "Item", also ein pair aus "Key" und "Inf" sind. Zum Vergleichen muessen
// wir natuerlich die Schluessel heranziehen, die wir mit "first" aus dem Paar
// erhalten
template <typename Key, typename Inf>
void BinaryHeap<Key, Inf>::push_down(int i)
{
  bool finished = false;
  int current_left_child, current_right_child;
  int k;

  while (!finished)
  {
    current_left_child = left_child(i);
    current_right_child = right_child(i);

    // Gibt es ueberhaupt noch ein rechtes Kind?
    if (current_right_child != -1)
    {
      // Falls ja, dann suchen wir das kleinere der beiden Kinder
      k = (data_[current_right_child].first < data_[current_left_child].first) ? current
    }
    else
    {
      // Gibt es denn wenigstens noch ein linkes Kind?
      if (current_left_child != -1)
```

```
      {
        // Dann faellt die Wahl leicht...
        k = current_left_child;
      }
      else
      {
        // Tja, viel koennen wir dann nicht mehr machen... Wir sind ja schon ganz unten
        return;
      }
    }

    // In k ist jetzt der Index des kleinsten Kindes des Knotens mit Index i. Muessen wi
    // i noch weiter nach unten druecken? Offensichtlich muessen wir dies genau dann tue
    // die Heap - Eigenschaft noch verletzt ist!
    if (data_[i].first > data_[k].first)
    {
      // Der Knoten hat einen groesseren Wert als sein kleinstes Kind! Wir muessen die b
      // miteinander vertauschen.
      Item tmp = data_[i];
      data_[i] = data_[k];
      data_[k] = tmp;

      // Und in der naechsten Iteration muessen wir mit der neuen Position des herunterz
      // Knotens starten
      i = k;
    }
    else
    {
      // Wie schoen, wir sind fertig!
      finished = true;
    }
  }
}

// Wir waehlen hier die triviale O(n log(n)) - Methode zum Erstellen des Heaps
template <typename Key, typename Inf>
void BinaryHeap<Key, Inf>::heapify()
{
  for (int i = (int)size() - 1; i >= 0; i--)
  {
    push_down(i);
  }
}
```

Und in der Datei `heaptest.C` finden sich ein paar kleine Tests der Implementierung:

```
#include <iostream>
#include <vector>
#include <string>
```

```
#include "BinaryHeap.h"

using namespace std;

int main(int argc, char **argv)
{
  vector<pair<int, string> > daten;

  daten.resize(10);

  srand(42);

  for (unsigned int i = 0; i<daten.size(); i++)
  {
    daten[i].first = rand();
    daten[i].second = i;
  }

  BinaryHeap<int, string> binheap(daten);

  for (unsigned int i = 0; i<binheap.size(); i++)
  {
    cout << i << " " << binheap[i].first << " " << binheap[i].second << endl;
  }

  cout << endl << endl;

  vector<int> parents;
  parents.push_back(0);

  cout << binheap[0].first << endl;

  bool finished = false;

  while (!finished)
  {
    vector<int> p2;

    for (int i=0; i<(int)parents.size(); i++)
    {

      int lc = binheap.left_child(parents[i]);
      int rc = binheap.right_child(parents[i]);

      p2.push_back(lc);
      p2.push_back(rc);
      if (lc != -1)
      {
```

```
        cout << binheap[lc].first << " ";
      }
      else
      {
        finished = true;
      }
      if (rc != -1)
      {
        cout << binheap[rc].first << " ";
      }
      else
      {
        finished = true;
      }
    }
    cout << endl;
    parents = p2;
  }

  return 0;
}
```

Aufgabe 5. (Prioritätswarteschlangen, Implementierung Bonuspunkte)
10
Den folgenden Code finden Sie in der Datei `PriorityQueue.h`

```
// Priority Queue implementation based on binary heap

#include <BinaryHeap.h>

#include <vector>
#include <exception>
#include <algorithm>

template <typename Key, typename Inf>
class PriorityQueue
{
  public:
  /// type definitions
  typedef std::pair<Key, Inf> Item;

  class QueueEmpty : std::exception
  {
    public:
    virtual const char* what() { return "Queue is empty!"; }
```

```
};

class NoDecrease : std::exception
{
  public:
  virtual const char* what() { return "Queue is empty!"; }
};

class OutOfRange : std::exception
{
  public:
  virtual const char* what() { return "Queue is empty!"; }
};

/// Default ctor
PriorityQueue() {}
/// Dtor
virtual ~PriorityQueue() {}

/** Insert a new item into the queue
*/
void insert(const Item& item);

/** Return the current minimum of the queue.
    @exception QueueEmpty if the queue is empty
*/
const Item& find_min() const;

/** Remove the minimum key from the queue.
    @exception QueueEmpty if the queue is empty
*/
void delete_min();

/** Merge the contents of another queue into this one.
*/
void merge(const PriorityQueue& pq);

/** Decrease the key and reorganize the heap.
*/
void decrease_key(int index, const Key& key);

/** Create a priority queue from an unsorted array
*/
void create(const std::vector<Item>& array);

/** Empty predicate
*/
bool empty() const { return heap.empty(); }
```

```
  /// The heap
  BinaryHeap<Key, Inf> heap;
};

template <typename Key, typename Inf>
void PriorityQueue<Key, Inf>::insert(const typename PriorityQueue<Key, Inf>::Item& item)
{
  // Store in the next empty tree leaf
  heap.push_back(item);
  // Restore heap property
  heap.push_up(heap.size() - 1);
}

template <typename Key, typename Inf>
void PriorityQueue<Key, Inf>::create(const std::vector<Item>& array)
{
  heap.data() = array;
  heap.heapify();
}

template <typename Key, typename Inf>
const typename PriorityQueue<Key, Inf>::Item& PriorityQueue<Key, Inf>::find_min() const
{
  // Consistency check -- throw exception if there's no minimum
  if (heap.empty()) throw QueueEmpty();

  // Return the root of the heap.
  return heap[0];
}

template <typename Key, typename Inf>
void PriorityQueue<Key, Inf>::delete_min()
{
  // Consistency check -- throw exception if there's no minimum
  if (heap.empty()) throw QueueEmpty();

  // Copy last element to the root, reduce heap size by one.
  heap[0] = heap[heap.size() - 1];
  heap.pop_back();

  // Restore heap propoerty.
  heap.push_down(0);
}

template <typename Key, typename Inf>
void PriorityQueue<Key, Inf>::decrease_key(int index, const Key& key)
{
```

```
  // Consistency check -- throw exception if there's no minimum.
  if (heap.empty()) throw QueueEmpty();

  // Consistency check -- throw exception if the key was not decreased.
  if (heap[index].first <= key) throw NoDecrease();

  // Consistency check -- throw exception if the index is out of range.
  if ((index <= 0) || (index >= heap.size())) throw OutOfRange();
  heap[index].first = key;

  // Restore heap property.
  heap.push_up(index);
}

template <typename Key, typename Inf>
void PriorityQueue<Key, Inf>::merge(const PriorityQueue& pq)
{
  // Allocate a new vector of items of appropriate size.
  std::vector<Item> tmp(heap.size() + pq.heap.size());

  // Copy the current data elements into it.
  std::copy(heap.data.begin(), head.data.end(), tmp.begin());

  // Append the contents of the second queue.
  std::copy(pq.head.data.begin(), pq.head.data.end(), tmp.begin() + heap.data.size());

  // Create a pq from the array.
  create(tmp);
}
```

Und ein paar kleine Tests in der Datei `priority_queue_test.C`

```
#include <iostream>

#include <PriorityQueue.h>


typedef PriorityQueue<int, int> PQ;
typedef PQ::Item Item;

int main()
{
  std::vector<Item> items(500);
  for (size_t i = 0; i < items.size(); i++)
  {
    items[i].first = rand();
  }

  PQ p;
  p.create(items);
```

```
  while (!p.empty())
  {
    std::cout << p.find_min().first << std::endl;
    p.delete_min();
  }

  return 0;
}
```

Abgabe: 05.07.2003