Priority Queue ADT

Definition

A *priority queue* P is a data type for ordering a set (collection) S of elements each with an associated value called a *key*. The keys can be put in order from the smallest to largest key (or from the largest to smallest key).

Operations on priority queues:

- `insertItem(k, x)` inserts a key `k` and element `x` from S into P.
- `minElement()` returns an element of S with the smallest key in P (but it does not remove it).
- `removeMin()` removes an element of S with the smallest key in P.
- `minKey()` returns the smallest key in P.
- `isEmpty( )` tests whether P is empty.

Note that more than one element of S can have the same key in P. The last two operations (`minKey()` and `isEmpty()`) are secondary functions and are not always implemented.

Priority queues are used to schedule some tasks (or events) with different priorities where priorities are usually natural numbers. Often the task with the smallest (or largest) priority number is the most important.

Using a heap to implement a priority queue is very efficient. Below is one of possible implementations of a priority queue using heap data structure. We assume that P is a one-dimensional array of length n with the heap property with respect to keys. The set S is not always explicitly stated and is created when P is created.

The Composition Pattern

The composition pattern defines a single object that is composed of other objects. In the priority queue we use a special composition pattern called pairs. A *pair* (k, e) is composed of two objects, k and e. To implement a pair (or, in general, more complex composition patterns such as triples, quadruples and so on) we define a class called `Item` that stores these two objects as member variables and that provides functions to access and update these variables. We assume that key is an integer, but element can be of any type (therefore we use template).

If we want to use keys which are not integers we need to choose an appropriate type for keys, but also we need to use a *comparator class* with a special *comparison function* for comparing keys. It makes implementation of the priority queue more complex but also more general.

```
template<typename ElemType>
class Item {
private:
   int key;
   ElemType elem;
public:
   Item(const int k=0, const ElemType& e=ElemType())
      : key(k), elem(e) { }  //constructor
   const int getKey() const { return key; }
   const ElemType& getElem() const { return elem; }
   void setKey(const int k) { key = k; }
   void setElem(const ElemType& e) { elem = e; }
};
```

The Comparator Pattern

An important issue in the priority queue ADT is how to compare keys. There are a few design choices. We choose a more general approach where we define a *comparator function*. A convenient way to provide such a function in C++ is to define a *comparator class*. Typically such a class has no data members and no other member functions. To define a comparison member function we overload the "`()`" operator. The resulting function takes two operands, a and b and returns an integer i such that $i < 0$, $i = 0$, or $i > 0$, depending on whether $a < b$, $a = b$, $a > b$, respectively. Below is an example of such a class where the arguments are of class `Item<ElemType>` and `ElemType` is any C++ type.

```
template<typename ElemType>
class PQComp {
public:
   int operator()(const Item<ElemType>& e1,
                  const Item<ElemType>& e2)
   {
      return e1.getKey() - e2.getKey();
   }
};
```

The Templated Class BinaryHeap

Our implementation of the priority queue ADT is based on the heap ADT. Therefore we need a templated class `BinaryHeap`.

```
template<typename ElemType, typename Comp>
class BinaryHeap {
private:
   Comp comp; // comparator class: comp(a,b) compares a and b
   int curSize; //number of elements in heap
   ElemType *array; //(dynamic) heap array
   int length; //the length of the array
   static const int DEF_SIZE = 8;
   void getNewArray(int newSize) {
      array = new ElemType[newSize];
      length = newSize;
   }
   void checkSize(); //double the size of array when full
```

```
public: // templated class BinaryHeap (cont)
   BinaryHeap(int size = DEF_SIZE) { //constructor
      curSize = 0;
      getNewArray(size);
   }
   ElemType& findMin() throw(EmptyHeap) {
      if ( isEmpty() ) throw EmptyHeap();
      return array[0];
   }
   bool isEmpty( ) const { return curSize == 0; }
   void buildHeap();
   void insert(const ElemType& x);
   void deleteMin() throw(EmptyHeap);
};
```

Implementation

We present only three templated functions: `insert`, `walkDown` and `deleteMin`. The other `BinaryHeap` functions can be changed in a similar way.

```
template<typename ElemType, typename Comp>
void BinaryHeap<ElemType,Comp>::insert(const ElemType& x)
{
   checkSize( );
   //walk up (establish heap order now)
   int hole = curSize++;
   for ( ; hole > 0 && comp(array[(hole-1)/2], x) > 0;
           hole = (hole-1)/2)
      array[hole] = array[(hole-1)/2];
   array[hole] = x;
}
```

```
template<typename ElemType, typename Comp>
void BinaryHeap<ElemType, Comp>::deleteMin()
     throw(EmptyHeap)
{
   array[0] = array[ --curSize ]; //decrease size
   walkDown(0);
}
```

```
template<typename ElemType, typename Comp>
void BinaryHeap<ElemType, Comp>::walkDown(int hole)
{
   int child;  ElemType key = array[hole];
   for ( ; 2*hole+1 < curSize; hole = child) {
      child = 2*hole+1;
      if (child != curSize-1 &&
          comp(array[child], array[child+1]) > 0)
         child++; // right child = 2*hole+2
      if (comp(key, array[child]) > 0)
         array[hole]=array[child];
      else break;
   }
   array[hole] = key;
}
```

The Class PriorityQueue

```
#include "BinaryHeap.h"
// include the templated class Item here
// include the templated class PQComp here
template<typename ElemType>
class PriorityQueue {
protected: typedef Item<ElemType> _Item;
   typedef PQComp<ElemType> _PQComp;
private: BinaryHeap<_Item, _PQComp> T;
   static const int DEF_SIZE = 8;
```

```
public: // class PriorityQueue cont.
   PriorityQueue(int size = DEF_SIZE) : T(size) { }
   bool isEmpty() const { return T.isEmpty(); }
   void insertItem(const int k, const ElemType& e)
   {
      T.insert(_Item(k, e));
   }
   const ElemType& minElement()
         throw(EmptyPriorityQueue)
   {
      if (T.isEmpty()) throw EmptyPriorityQueue();
      return T.findMin().getElem();
   }
```

```
    // class PriorityQueue cont.
    const int minKey() throw(EmpyPriorityQueue)
    {
       if (T.isEmpty()) throw EmptyPriorityQueue();
       return T.findMin().getKey();
    }
    void removeMin() throw(EmptyPriorityQueue)
    {
       if (T.isEmpty()) throw EmptyPriorityQueue();
       T.deleteMin();
    }
}; // end of class PriorityQueue
```

Example

This is a simple example showing how to use the class `PriorityQueue`.

```
#include "PriorityQueue.h"
int main() {
   PriorityQueue<string> pq;
   string elem;
   elem = "abc"; pq.insertItem(3, elem);
   elem = "bbc"; pq.insertItem(7, elem);
   elem = "cnn"; pq.insertItem(8, elem);
   cout << "minElement: " << pq.minElement() << endl;
   cout << "minKey: " << pq.minKey() << endl;
   cout << "Min element removed." << endl;
   pq.removeMin();
}
```

Running Time

The running time of:

- `minElement()` and `minKey()` is $\Theta(1)$
- `removeMin()` is $O(\lg n)$
- `insertItem()` is $O(\lg n)$