Data Structures and Algorithms II
Fall 2016
Programming Assignment #2

You are going to create a class called "heap" that provides programmers with the functionality of a priority queue using a binary heap implementation. Each item inserted into the binary heap will specify a unique string id, an integer key, and optionally any pointer. The implementation of the class should use pointers to void in order to handle pointers to any type of data. When a heap is declared, a capacity will be passed to its constructor representing the maximum number of items that may be in the heap at one time; the heap will never be allowed to grow past its initial capacity (although it is not difficult to implement a resize operation).

I have written a program that uses my own implementation of the class. I will provide you with that program (useHeap.cpp) and with a Makefile. *You should not change my source code file!* You are allowed to add c++11 flags to the Makefile if you need to. Both files will be discussed in class and can be obtained from the course home page. Your heap will also make use of the hash table class that you created for the previous programming assignment.

This assignment asks you to fill in the missing heap.cpp and heap.h files, and to correct or add to your hash.cpp file if necessary, so that everything works. This implies that your heap class must include at least the following: a *constructor* that accepts an integer representing the capacity of the binary heap; a public member function, *insert*, used to insert a new item into the heap; a public member function, *deleteMin*, that removes the item with the lowest key from the heap; a public member function, *setKey*, providing both increaseKey and decreaseKey functionality; and a public member function, *remove*, that allows the programmer to delete an item with a specified id from the heap. In class we will discuss the parameters of these member functions and their return values. In addition, your class should contain private data members and private member functions that allow you to elegantly and efficiently implement the required public member functions. I will discuss my own implementation in class, and it is described on the next page.

In class, we will look at a sample run of the program and discuss the provided code. This program only passes string ids and integer keys to the insert member function of the heap class, but again, the insert member function should also optionally accept any pointer that can be stored and associated with the id. In the future, you will be using the class you write for this assignment in order to implement an algorithm involving graph data structures, and this functionality will be necessary. Also note that the integer keys will not necessarily be positive integers.

All operations should be implemented using average-case logarithmic time (or better) algorithms. In order to achieve setKey and remove in average-case logarithmic time, your program needs to be able to map an id to a node quickly. Since each id can be any arbitrary string, a hash table is useful for this purpose. Searching a heap to find an item with a particular id would require linear time, but a hash table in which each hash entry includes a pointer to the associated node in the heap allows you to find the item in constant average time. Apart from the calls to the hash table member functions, which are worst-case linear time but average-case constant time operations, all heap operations should use worst-case logarithmic time algorithms, and the insert operation should use an average-case constant time algorithm.

My heap class contains four private data members. Two are simple integers representing the capacity and the current size of the heap. The third is a vector of node objects containing the actual data of the heap; each node contains a string id, an integer key, and a pointer to void that can point to anything. (I have made "node" a private nested class within the heap class.) The fourth private data member is a pointer to a hash table (the actual hash table is allocated in the heap's constructor). Since the constructor is provided with the maximum size of the heap, you may allocate the hash table to be large enough such that there is a small likelihood of a rehash, but that is up to you. (Note that since items get removed from the heap, but only lazily deleted from the hash table, it is still possible that a rehash of the hash table will be necessary.)

Your heap.h file should contain the declaration of your class along with the declarations of its public and private data members and member functions. The heap.cpp file should contain the implementation of the class. I don't think you should need to implement any functions other than the member functions of the class itself in this file (I did not).

As usual, you will be graded not only on the correctness of your program, but also on the appropriateness of the decisions that you make, the elegance (and perhaps the formatting) of your code, and on the appropriate use of C++ concepts and routines.

The following page shows the declarations of the constructor and the public member functions of my heap class along with my comments describing their functionalities, parameters, and return values. I am not showing the declarations of my private data members or private member functions here, but this will be discussed further in class.

E-mail me (*CarlSable.Cooper@gmail.com*) your program, including all source code files, head files, and your Makefile (including any provided files that you use without making changes). Your program must compile and run using either Ubuntu or Cygwin. The program is due before midnight on the night of Tuesday, October 18.

```
//
// heap - The constructor allocates space for the nodes of the heap
// and the mapping (hash table) based on the specified capacity
//
heap(int capacity);

//
// insert - Inserts a new node into the binary heap
//
// Inserts a node with the specified id string, key,
// and optionally a pointer. They key is used to
// determine the final position of the new node.
//
// Returns:
//   0 on success
//   1 if the heap is already filled to capacity
//   2 if a node with the given id already exists (but the heap
//     is not filled to capacity)
//
int insert(const std::string &id, int key, void *pv = NULL);

//
// setKey - set the key of the specified node to the specified value
//
// I have decided that the class should provide this member function
// instead of two separate increaseKey and decreaseKey functions.
//
// Returns:
//   0 on success
//   1 if a node with the given id does not exist
//
int setKey(const std::string &id, int key);

//
// deleteMin - return the data associated with the smallest key
//             and delete that node from the binary heap
//
// If pId is supplied (i.e., it is not NULL), write to that address
// the id of the node being deleted. If pKey is supplied, write to
// that address the key of the node being deleted. If ppData is
// supplied, write to that address the associated void pointer.
//
// Returns:
//   0 on success
//   1 if the heap is empty
//
int deleteMin(std::string *pId = NULL, int *pKey = NULL, void *ppData = NULL);

//
// remove - delete the node with the specified id from the binary heap
//
// If pKey is supplied, write to that address the key of the node
// being deleted. If ppData is supplied, write to that address the
// associated void pointer.
//
// Returns:
//   0 on success
//   1 if a node with the given id does not exist
//
int remove(const std::string &id, int *pKey = NULL, void *ppData = NULL);
```

My nested class:

```
class node { // An inner class within heap
public:
  std::string id; // The id of this node
  int key; // The key of this node
  void *pData; // A pointer to the actual data
};
```

The declarations of my data vector and hash table pointer:

```
std::vector<node> data; // The actual binary heap
hashTable *mapping; // maps ids to node pointers
```

Private member functions of my heap:

```
void percolateUp(int posCur);
void percolateDown(int posCur);
int getPos(node *pn);
```

Part of my heap constructor:

```
data.resize(capacity+1);
mapping = new hashTable(capacity*2);
```

A simple getPos implementation:

```
int pos = pn - &data[0];
return pos;
```

An example of a call to the hash table's setPointer member function:

```
mapping->setPointer(data[posCur].id, &data[posCur]);
```

An example of a call to the hash table's getPointer member function:

```
node *pn = static_cast<node *> (mapping->getPointer(id, &b));
```

Filling in ppData in deleteMin:

```
*(static_cast<void **> (ppData)) = data[1].pData;
```