Jacob Ciurca

Dr. Lu

Lab 8

Cosc 320

11/12/2020

**Pre Lab:** Reviewed priority queue and understood the two given header files

**Lab:**

**D\_pqueue.h:** had to change the template <typename T, typename Compare = greater<T> >

To template <typename T, typename Compare = less<T> > for program to work properly

#ifndef MINI\_PRIORITY\_QUEUE\_CLASS

#define MINI\_PRIORITY\_QUEUE\_CLASS

#include <vector>

#include "d\_heap.h"

#include "d\_except.h"

using namespace std;

// maintain a priority queue containing elements of data type

// T using a comparison function object of type Compare

template <typename T, typename Compare = less<T> >

class miniPQ

{

public:

miniPQ();

// create empty priority queue

int size() const;

// return the number of elements in the priority queue

bool empty() const;

// is the priority queue empty?

void push(const T& item);

// insert item into the priority queue

// Postcondition: the heap size increases by 1

void pop();

// remove the element of highest priority.

// Precondition: the priority queue is not empty.

// if condition fails, the function throws the

// underflowError exception.

// Postcondition: the heap decreases by 1

T& top();

// return the element of highest priority

// Precondition: the priority queue is not empty.

// if the condition fails, the function throws the

// underflowError exception

const T& top() const;

// constant version

private:

vector<T> pqList;

// pqList holds the priority queue elements

Compare comp;

// function object used for comparisons

};

// constructor. create empty priority queue

template <typename T, typename Compare>

miniPQ<T,Compare>::miniPQ()

{}

// return the size of the priority queue

template <typename T, typename Compare>

int miniPQ<T,Compare>::size() const

{

return pqList.size();

}

// return true if the priority queue is empty and false

// otherwise

template <typename T, typename Compare>

bool miniPQ<T,Compare>::empty() const

{

return pqList.empty();

}

// insert a new item in the priority queue

template <typename T, typename Compare>

void miniPQ<T,Compare>::push(const T& item)

{

// insert the item at the end of the vector

// call pushHeap() to restore the heap condition.

pqList.push\_back(item);

pushHeap(pqList,pqList.size(), comp);

}

// remove the element of highest priority,

template <typename T, typename Compare>

void miniPQ<T,Compare>::pop()

{

// check for an empty priority queue

if (pqList.empty())

throw underflowError("miniPQ pop(): empty list");

// call popHash() to put element at back of the vector

popHeap(pqList, pqList.size(), comp);

// delele element from back of pqList

pqList.pop\_back();

}

template <typename T, typename Compare>

T& miniPQ<T,Compare>::top()

{

// check for an empty heap

if (pqList.empty())

throw underflowError("miniPQ top(): empty list");

// return the root of the heap

return pqList[0];

}

template <typename T, typename Compare>

const T& miniPQ<T,Compare>::top() const

{

// check for an empty heap

if (pqList.empty())

throw underflowError("miniPQ top(): empty list");

// return the root of the heap

return pqList[0];

}

#endif // MINI\_PRIORITY\_QUEUE\_CLASS

**Preqrec.h:**

#ifndef PROCESSREQUESTRECORD\_CLASS

#define PROCESSREQUESTRECORD\_CLASS

#include <iostream>

#include <string>

using namespace std;

class procReqRec

{

public:

// default constructor

procReqRec()

{}

// constructor

procReqRec(const string& nm, int p){

name = nm;

priority = p;

}

// access functions

int getPriority(){

return priority;

}

string getName(){

return name;

}

// update functions

void setPriority(int p){

priority = p;

}

void setName(const string& nm){

name = nm;

}

// for maintenance of a minimum priority queue

friend bool operator< (const procReqRec& left, const procReqRec& right);

// output a process request record in the format

// name: priority

friend ostream& operator<< (ostream& ostr, const procReqRec& obj);

private:

string name; // process name

int priority; // process priority

};

bool operator< (const procReqRec& left, const procReqRec& right){

if(left.priority < right.priority){

return true;

}

else{

return false;

}

}

ostream& operator<< (ostream& ostr, const procReqRec& obj){

ostr << obj.name << ": " << obj.priority;

return ostr;

}

#endif // PROCESSREQUESTRECORD\_CLASS

**Lab\_08.cpp:**

#include <iostream>

#include <stdlib.h>

#include "preqrec.h"

#include "d\_except.h"

#include "d\_pqueue.h"

#include "d\_heap.h"

using namespace std;

int main()

{

int f = rand()%39;

int f1 = rand()%39;

int f2 = rand()%39;

int f3 = rand()%39;

int f4 = rand()%39;

int f5 = rand()%39;

int f6 = rand()%39;

int f7 = rand()%39;

int f8 = rand()%39;

int f9 = rand()%39;

miniPQ<procReqRec> pq;

procReqRec p("Process A", f);

procReqRec p1("Process B", f1);

procReqRec p2("Process C", f2);

procReqRec p3("Process D", f3);

procReqRec p4("Process E", f4);

procReqRec p5("Process F", f5);

procReqRec p6("Process G", f6);

procReqRec p7("Process H", f7);

procReqRec p8("Process I", f8);

procReqRec p9("Process J", f9);

pq.push(p);

pq.push(p1);

pq.push(p2);

pq.push(p3);

pq.push(p4);

pq.push(p5);

pq.push(p6);

pq.push(p7);

pq.push(p8);

pq.push(p9);

while(pq.empty() == false){

cout << pq.top() << endl;

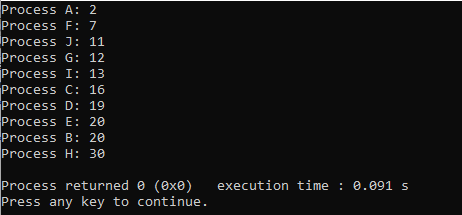
pq.pop();

}

return 0;

}

**Sample Output:**



**Post Lab:** This lab was good at helping us get practice with priority queues and took me about 1 hour 30 minutes to complete and I completed this lab by myself with no help