/\* Assignment: Heap Program

Author: Ryan Wood

Created On: April 1, 2018

Requirements: Create a priority heap class and

test it using the program specified

in the document Heap Program handed out

in class March 27, 2018

\*/

/\*\*\*\*\*\*\*\*\*Main file\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <fstream>

#include "../List/List.h"

#include "Heap.h"

#include "Customer.h"

using namespace std;

const int CAPACITY = 105;

Heap<Customer, CAPACITY> customerHeap;

ifstream inFile("HeapPriortyNbrs.dat");

ofstream outFile("HeapOutput.out");

/\* Function: readRequests(int)

Purpose: reads as many items from the input file

as specified by the given integer value,

reading the id followed by a dash, followed

by the priority and adding customer class

instances to the heap based on the data

found in the file. -1 may be passed to simply

read all data items that remain in the file

Parameters: the number of customer reauests to read

or -1 if all data items should be read

\*/

void readRequests(int numRequests)

{

int count = 0;

Customer \*cust = NULL;

char dash = '-';

cust = new Customer;

while((count < numRequests || numRequests == -1) && !inFile.eof())

{

inFile >> cust->id;

inFile >> dash;

inFile >> cust->priority;

customerHeap.push(\*cust);

count++;

}

delete cust;

}

/\* Function: popCustomers(int, int)

Purpose: prints the number of customers from the heap as given

as the first integer argument up to the number of items

given as the second argument, which is the number of items

to pop. Pops items off the heap, printing them as it goes

until the given numToPop has been reached. Ceases to print

to the console or file once the given numToPrint integer

value has been reached

Parameters: the number of values to print,

the number of values to pop from the stack

\*/

void popCustomers(int numToPrint, int numToPop)

{

Customer \*cust = NULL;

int count = 0;

int printed = 0;

cust = new Customer;

while((count < numToPop && customerHeap.size() > 0) ||

(numToPop == -1 && customerHeap.size() > 0))

{

\*cust = customerHeap.top();

customerHeap.pop();

if(count < numToPrint)

{

printed++;

if(printed > 1)

{

outFile << ", " << \*cust;

cout << ", " << \*cust;

}

else

{

outFile << \*cust;

cout << \*cust;

}

if(printed %10 == 0)

{

outFile << endl;

cout << endl;

printed = 0;

}

}

count++;

}

outFile << endl;

cout << endl;

}

int main()

{

if(!inFile)

{

cout << "Bad input file name." << endl;

return -1;

}

readRequests(100);

popCustomers(20, 50);

readRequests(50);

popCustomers(30, 50);

readRequests(-1);

popCustomers(40, -1);

return 0;

}

#ifndef CUSTOMER\_H

#define CUSTOMER\_H

/\* Class: Customer

Purpose: Specialized class created to

demonstrate the priority heap

Author: Ryan Wood

Created On: April 7, 2018

\*/

class Customer

{

public:

int id;

int priority;

/\* Function: operator >(Customer)

Purpose: returns whether the given customer has a

lower priority than the priority value

of this class instance

Parameters: the customer to compare

Return: whether the customer has a lower priorty

\*/

bool operator >(Customer custA)

{

if(priority > custA.priority)

return true;

return false;

}

/\* Function: operator <(Customer)

Purpose: determines whether the given customer has

a higher priority than the priority of

the current Customer class instance.

Paramters: the customer to compare

Return: whether the given customer has a higher priority

\*/

bool operator <(Customer custA)

{

if(priority < custA.priority)

return true;

return false;

}

};

/\* Function: operator <<(ostream&, Customer&)

Purpose: outputs the given customer's priority followed by

a dash, followed by the customer's ID to the console

Parameters: the output stream, the customer

Return: the output stream

\*/

std::ostream& operator<<(std::ostream& out, Customer &cust)

{

out << cust.priority << " - " << cust.id;

return out;

}

/\* Function: operator <<(ofstream&, Customer&)

Purpose: outputs the given customer's priority followed by

a dash, followed by the customer's ID to the file stream

Parameters: the output file stream, the customer

Return: the output file stream

\*/

std::ofstream& operator<<(std::ofstream& out, Customer &cust)

{

out << cust.priority << " - " << cust.id;

return out;

}

#endif

/\* Assignment: Heap Program

Author: Ryan Wood

Created On: April 1, 2018

Requirements: Create a priority heap class and

test it using the program specified

in the document Heap Program handed out

in class March 27, 2018

\*/

/\*\*\*\*\*\* Heap Header file \*\*\*\*\*\*\*/

#ifndef HEAP\_H

#define HEAP\_H

#include <iostream>

#include "../List/List.h"

template <typename et, size\_t CAPACITY>

class Heap

{

public:

/\* Function: Heap();

Purpose: cunstructor. Creates a List with the

capacity of the Heap

\*/

Heap();

/\* Function: ~Heap();

Purpose: destroys the heap, deleting the list pointer

\*/

~Heap();

/\* Function: push(et)

Purpose: pushes the given value into the heap, making it the

root if it is greater than the rest of the elements

in the heap. Shifts elements depending on the new value

Parameters: the value

\*/

void push(et val);

/\* Function: pop()

Purpose: deletes the largest element in the heap, the root,

moving the element at the end of the list to the

top and then walking that value down until a new

root is placed and the element is back at a reasonable

position in the heap

\*/

void pop();

/\* Function: top()

Purpose: retrieves the item from the top of the heap, which

is the largest item in the heap

Return: the largest item in the heap

\*/

et top();

/\* Function: size() const

Purpose: retrieves the size of the heap, ie the number

of elements of the heap that are in use

Return: the number of used elements in the heap

\*/

size\_t size() const;

/\* Function: capacity() const

Purpose: retrieves the number of elements that the heap may hold

before it is full

Return: the capacity of the heap

\*/

size\_t capacity() const;

/\* Function: full() const

Purpose: determines whether the heap is unable to take on any

new elements due to size reaching capacity

Return: whether the heap is full

\*/

bool full() const;

/\* Function: print() const

Purpose: prints the heap's elements. Actually, prints out

the List, which includes the heap's unused element 0.

\*/

void print() const;

private:

List<et, CAPACITY> \*list;

};

template<typename et, size\_t CAPACITY>

inline Heap<et, CAPACITY>::~Heap()

{

delete list;

}

template<typename et, size\_t CAPACITY>

inline Heap<et, CAPACITY>::Heap()

{

et nothing;

list = new List<et, CAPACITY>;

list->insertAfter(nothing);//fill array position. We won't need it

}

template<typename et, size\_t CAPACITY>

inline void Heap<et, CAPACITY>::pop()

{

bool isHeap = true;

int curIndex = 0;

int lIndex = 0;

int childIndex = 0;

int rIndex = 0;

et lastVal;

if(list->size() <= 1)

return;

//get the last value

lastVal = list->getElement();

list->erase();

if(list->getPos() == 0)

return;

//move the current index to the first value, the big one

list->setPos(1);

//replace the value

list->replace(lastVal);

curIndex = list->getPos();

do

{

isHeap = true;

lIndex = curIndex \* 2;

rIndex = (curIndex \*2) + 1;

childIndex = list->max(lIndex, rIndex);

if(list->max(curIndex, childIndex) == childIndex)

{

list->swap(curIndex, childIndex);

curIndex = childIndex;

isHeap = false;

}

}while(!isHeap);

list->last();

}

template<typename et, size\_t CAPACITY>

inline void Heap<et, CAPACITY>::push(et val)

{

bool isHeap = true;

int parentIndex = 0;

int childIndex = 0;

et parent;

et child;

list->insertAfter(val);

childIndex = list->getPos();

do

{

isHeap = true;

if(childIndex > 1)

{

parentIndex = childIndex/2;

parent = list->getElement(parentIndex);

if(list->max(parentIndex, childIndex) == childIndex)

{

list->swap(childIndex, parentIndex);

childIndex = parentIndex;

isHeap = false;

}

}

}while(!isHeap);

list->last();

}

template<typename et, size\_t CAPACITY>

inline et Heap<et, CAPACITY>::top()

{

if(list->size() > 1)

return list->getElement(1);

else

return list->getElement(0);

}

template<typename et, size\_t CAPACITY>

inline size\_t Heap<et, CAPACITY>::size() const

{

return list->size()-1;

}

template<typename et, size\_t CAPACITY>

inline size\_t Heap<et, CAPACITY>::capacity() const

{

return CAPACITY;

}

template<typename et, size\_t CAPACITY>

inline bool Heap<et, CAPACITY>::full() const

{

return size() < capacity();

}

template <typename et, size\_t CAPACITY>

inline void Heap<et, CAPACITY>::print() const

{

std::cout << \*list << std::endl;

}

#endif

/\* Assignment: General List Program

Author: Ryan Wood

Created On: 1/28/2018

Last Modified: 2/24/2018

\*/

/\*List definition file\*/

#ifndef LIST\_H

#define LIST\_H

#include <iostream>

#include <fstream>

#include <string>

template <typename et, size\_t CAPACITY>

class List

{

public:

/\* Function: List()

Purpose: default constructor.

Creates a List with a capacity of 20

Return: an empty List

\*/

List();

/\* Function: List()

Purpose: copy constructor

Paramters: the List to copy

Return: a new List with the same values as

the one given

\*/

List(const List &lstA);

/\* Function: emtpy()

Purpose: determines whether the list has no values

Return: whether elements are used

\*/

bool empty() const;

/\* Function: full()

Purpose: determines whether the list is unable

to take on any new values

Return: whether used is equal to CAPACITY

\*/

bool full() const;

/\* Function: first()

Purpose: sets the position to the first

element of the array, 0

\*/

void first();

/\* Function: last()

Purpose: sets the position of the array to last element

in the array, used-1

\*/

void last();

/\* Function: prev()

Purpose: sets the position back by one index

\*/

void prev();

/\* Function: next()

Purpose: sets the position forward by one index

\*/

void next();

/\* Function: getPosition() const

Purpose: retrieves the current position index

Return: the index

\*/

int getPos() const;

/\* Funcrtion: setPos(int)

Purpose: sets the current position to the given

index if it is valid

Parameters: an index less than the size of the List

\*/

void setPos(int);

/\* Function: insertBefore(et)

Purpose: inserts the given value before the current

position in the list

Parameters: the value

\*/

void insertBefore(et);

/\* Function: insertAfter(et)

Purpose: inserts the given value after the

current position in the List

Parameters: the value

\*/

void insertAfter(et);

/\* Function: getElement() const

Purpose: retrieves the value at the current position

in the list

Return: the value at the current index

\*/

et getElement() const;

/\* Function: getElement(int)

Purpose: retrieves the value at the given index to the List

if it is valid for the list

Parameters: the index

Return: the value

\*/

et getElement(int pos) const;

/\* Function: size()

Purpose: gets the size of the list, the number of

elements that have been used

Return: the number of elements used

\*/

size\_t size() const;

/\* Function: replace(et)

Purpose: replaces the value at the current index

to the list with the value given

Parameters: the value

\*/

void replace(et);

/\* Function: erase()

Purpose: erases the element at the current index to

the list, shifting all other elements down

to take the empty place

\*/

void erase();

/\* Function: clear()

Purpose: deletes all elements in the list, setting

pos and used both to 0

\*/

void clear();

/\* Function: swap(int, int)

Purpose: swaps the element at the given first index

with the element at the given second index

if neither index is greater than the current

number of values used

Parameters: the first index, the second index

\*/

void swap(int indexA, int indexB);

/\* Function: max(int, int)

Purpose: determines which of the two given indexes

to the list contains the largest value,

returning the index with the largest value.

If one index has a null value, returns the

other. If they are both null returns -1

Parameters: the first index, the second index

Return: the index holding the larger value

\*/

int max(int indexA, int indexB);

/\* Function: getCapacity()

Purpose: returns the maximum number of elements the

list is able to hold

Return: capacity

\*/

size\_t getCapacity() const;

private:

/\* Function: copy(List&)

Purpose: copies the given list to the current list,

element by element, copying the values of

the array and the value of pos and used

Parameters: a valid list

\*/

void copy(const List &lstA);

et \*arry;//the list itself

int used;//how many elements have been used

int pos;//the current pointer to the list

};//List class

template <typename et, size\_t CAPACITY>

inline List<et, CAPACITY>::List()

{

used = 0;

pos = 0;

arry = new et[CAPACITY];

}

template <typename et, size\_t CAPACITY>

inline List<et, CAPACITY>::List(const List &lstA)

{

copy(lstA);

}

/\* Function: operator<<(ostream&, List&)

Purpose: overloads the stream insertion operator using the given

output to the console

Parameters: the console output stream, the List to print

Return: a reference to the output stream

\*/

template <typename et, size\_t CAPACITY>

std::ostream& operator <<(std::ostream &out, const List<et, CAPACITY> &lst);

/\* Function: operator <<(ofstream&, List&)

Purpose: overloads the given file output stream and prints the

given list to the file

Parameters: a reference to the file output stream,

a reference to the LIst to print

Return: a reference to the file output stream

\*/

template <typename et, size\_t CAPACITY>

std::ofstream& operator <<(std::ofstream &out, const List<et, CAPACITY> &lst);

/\* Function: operator==(List&, List&)

Purpose: determines whether the given lists are equal

Parameters: the first list, the second list

Return: whether the lists have the same values

\*/

template <typename et, size\_t CAPACITY>

bool operator ==(const List<et, CAPACITY> &lstA, const List<et, CAPACITY> &lstB);

/\* Function: operator !=(List&, List&)

Purpose: determines whether the given lists are not equal

Parameters: the first list, the second list

Return: whether the two lists do not have the same values

\*/

template <typename et, size\_t CAPACITY>

bool operator !=(const List<et, CAPACITY> &lstA, const List<et, CAPACITY> &lstB);

/\* Function: operator +(List&, List&)

Purpose: adds the values of the second list into the remaining values

of the first list, creating a new list with the results

Parameters: the first list, the second List

Return: a list with values of both lists

\*/

template <typename et, size\_t CAPACITY>

List<et, CAPACITY> operator +(const List<et, CAPACITY> &lstA, const List<et, CAPACITY> &lstB);

template<typename et, size\_t CAPACITY>

inline bool List<et, CAPACITY>::empty() const

{

return(used == 0);

}

template<typename et, size\_t CAPACITY>

inline bool List<et, CAPACITY>::full() const

{

return(used == CAPACITY);

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::first()

{

pos = 0;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::last()

{

pos = used-1;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::prev()

{

if(pos > 0)

pos--;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::next()

{

if(pos < CAPACITY-1 && pos < used-1)

pos++;

}

template<typename et, size\_t CAPACITY>

inline int List<et, CAPACITY>::getPos() const

{

return pos;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::setPos(int val)

{

if(val < used && val > -1)

pos = val;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::insertBefore(et val)

{

et \*current;

et \*tmp;

et \*next;

//case used = 0

if(used < CAPACITY)

{

if(used == 0)

{

arry[pos] = val;

}

else

{

//set current to position address

current = &arry[pos];

//set a temp var to the next to last value

tmp = &arry[used-1];

next = tmp+1;

//loop backward up to our current position

while(tmp >= current)

{

//set the next value to be the value before it

\*next = \*tmp;

tmp--;

next--;

}

//now set the current value to the given value

\*current = val;

}

//increment how many values there are

used++;

}

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::insertAfter(et val)

{

et \*current;

et \*tmp;

if(used < CAPACITY)

{

//case used = 0;

if(used == 0)

arry[pos] = val;

else

{

current = &arry[pos];

tmp = &arry[used-1];

//from the last item to the current item, loop

//backward through array, moving elements forward by one

while(tmp >= current)

{

\*(tmp+1) = \*tmp;

tmp--;

}

current++;

\*current = val;

pos++;

}

used++;

}

}

template<typename et, size\_t CAPACITY>

inline et List<et, CAPACITY>::getElement(int ePos) const

{

et val;

if(pos >= 0 && pos < used)

{

val = arry[ePos];

}

return val;

}

template<typename et, size\_t CAPACITY>

inline et List<et, CAPACITY>::getElement() const

{

et val;

if(used > 0)

{

val = arry[pos];

}

return val;

}

template<typename et, size\_t CAPACITY>

inline size\_t List<et, CAPACITY>::size() const

{

return used;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::replace(et val)

{

arry[pos] = val;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::erase()

{

int count = 0;

if(used > 0)

{

count = pos;

while(count < used)

{

arry[count] = arry[count+1];

count++;

}

used--;

pos--;

}

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::swap(int indexA, int indexB)

{

et tmp;

if(indexA <= used && indexB <= used)

{

tmp = arry[indexA];

arry[indexA] = arry[indexB];

arry[indexB] = tmp;

}

}

template<typename et, size\_t CAPACITY>

inline int List<et, CAPACITY>::max(int indexA, int indexB)

{

int maxIndex = -1;

et valA;

et valB;

if(indexA >= used || indexB >= used)

{

if(indexA >= used && indexB >= used)

maxIndex = -1;

else if(indexA >= used)

maxIndex = indexB;

else

maxIndex = indexA;

}

else

{

valA = arry[indexA];

valB = arry[indexB];

if(valA > valB)

maxIndex = indexA;

else

maxIndex = indexB;

}

return maxIndex;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::clear()

{

int count = 0;

et null;

for(count = 0; count < used; count++)

{

arry[count] = null;

}

used = 0;

pos = 1;

}

template<typename et, size\_t CAPACITY>

inline void List<et, CAPACITY>::copy(const List &lstA)

{

int count = 0;

et \*iter = NULL;

CAPACITY = lstA.getCapacity();

arry = new et[CAPACITY];

used = 0;

pos = 0;

if(!lstA.empty())

{

iter = arry;

for(count = 0; count < lstA.size(); count++)

{

\*iter = lstA.getElement(count);

iter++;

}

}

pos = lstA.getPos();

used = lstA.size();

}

template<typename et, size\_t CAPACITY>

inline size\_t List<et, CAPACITY>::getCapacity() const

{

return CAPACITY;

}

template<typename et, size\_t CAPACITY>

inline std::ostream& operator <<(std::ostream &out, const List<et, CAPACITY> &lst)

{

int count = 0;

for(count = 0; count < lst.size(); count++)

{

if(count == 0)

out << lst.getElement(count);

else

out << ", " << lst.getElement(count);

}

return out;

}

template<typename et, size\_t CAPACITY>

inline std::ofstream& operator <<(std::ofstream &out, const List<et, CAPACITY> &lst)

{

int count = 0;

for(count = 0; count < lst.size(); count++)

{

if(count == 0)

{

out << lst.getElement(count);

}

else

out << ", " << lst.getElement(count);

}

return out;

}

template<typename et, size\_t CAPACITY>

inline bool operator ==(const List<et, CAPACITY> &lstA, const List<et, CAPACITY> &lstB)

{

int count = 0;

if(lstA.size() != lstB.size())

return false;

for(count = 0; count < lstA.size(); count++)

{

if(lstA.getElement(count) != lstB.getElement(count))

return false;

}

return true;

}

template<typename et, size\_t CAPACITY>

inline bool operator !=(const List<et, CAPACITY> &lstA, const List<et, CAPACITY> &lstB)

{

return(!(lstA == lstB));

}

template<typename et, size\_t CAPACITY>

inline List<et, CAPACITY> operator +(const List<et, CAPACITY> &lstA, const List<et, CAPACITY> &lstB)

{

int count = 0;

List<et, CAPACITY> lstC;

for(count = 0; count < lstA.size(); count++)

{

lstC.insertAfter(lstA.getElement(count));

}

for(count = 0; count < lstB.size(); count++)

{

lstC.insertAfter(lstA.getElement(count));

}

return lstC;

}

#endif

24 - 120, 24 - 224, 24 - 172, 23 - 214, 22 - 212, 22 - 206, 22 - 159, 22 - 155, 22 - 291, 22 - 136

21 - 136, 21 - 245, 21 - 167, 21 - 208, 21 - 223, 21 - 130, 21 - 288, 21 - 238, 21 - 152, 21 - 105

24 - 254, 24 - 298, 24 - 116, 24 - 108, 23 - 287, 23 - 203, 23 - 126, 23 - 192, 22 - 257, 22 - 122

21 - 289, 21 - 188, 21 - 298, 20 - 157, 19 - 109, 19 - 116, 18 - 100, 18 - 291, 18 - 279, 18 - 181

17 - 232, 17 - 285, 17 - 289, 17 - 186, 16 - 121, 16 - 289, 16 - 140, 16 - 104, 16 - 201, 16 - 224

24 - 124, 24 - 249, 24 - 159, 24 - 123, 23 - 295, 23 - 253, 23 - 192, 23 - 170, 23 - 170, 23 - 166

22 - 180, 22 - 259, 22 - 139, 22 - 108, 22 - 116, 21 - 229, 21 - 163, 21 - 171, 21 - 205, 20 - 160

19 - 174, 19 - 210, 19 - 201, 18 - 168, 18 - 118, 18 - 256, 18 - 255, 18 - 182, 17 - 229, 17 - 205

16 - 249, 16 - 122, 16 - 282, 15 - 255, 15 - 149, 15 - 234, 14 - 137, 14 - 296, 14 - 148, 14 - 122