Edward Huang
861026626

/*Exercise 1
If the elements of a list are sorted, is an array-based or a
linked-list-based implementation of the list more efficient for binary
search? Explain.

Answer:
An array-based implementation of the list is more efficient for binary
search because you can randomly access elements in an array. By halving
a sorted array, you can find the element in O(logn) time for the worst
case. Whereas for a sorted linked-list you will have to access the
elements sequentially so the worst case would be O(n).
*/

/**Exercise 2
 * a. Write C++ code to implement an integer queue class using linked-list,
 * where the nodes are stored sorted by ascending value of the integer
 * they store. We call this a priority queue. Specifically, implement
 * enqueue and dequeue methods.
 *
 * b. What is the average asymptotic cost per call to enqueue and to dequeue?
 * The average asymptotic cost per call to enqueue is O(n) and to
 * dequeue is O(1)
 *
 * See code below
 *
 * c. What if for each node, in addition to a pointer to the next node,
 * you add a pointer to the 10th next node. Modify your code to take
 * advantage of this.
 *
 * Had problems implementing this, so just left out the code.
 *
 * d. Can the modification in (c) improve the cost (not asymptotic
 * but just execution time) of enqueue? Does it improve the asymptotic cost?
 *
 * I believe that the modification in (c) will improve the cost of enqueue
 * because you can access not just the next and previous elements, but also
 * an element 10 indices away. The asymptotic cost is not improved because
 * you still need to traverse the queue to sort it.
 *
 * e. Is there any disadvantage that modification (c) incurs?
 * The disadvantage that modification (c) incurs is that you have to keep an
 * extra pointer to every 10th next node, which will take up extra memory.
 * The programmer also has to make sure that a 10th next node exists,
 * otherwise they will be pointing to a nonexistant node.
 */

```
#include <iostream>

using namespace std;

class Node {
public:
  Node();
  ~Node();
private:
  int data;
  Node *prev;
  Node *next;
  friend class QueueList;
};

//Node methods
Node::Node()
{}

Node::~Node()
{}

class QueueList
{
  public:
  QueueList();
  ~QueueList();
  void enqueue(int elem);
  void sort(int length);
  void print();
  int dequeue();

  private:
  int size;
  Node* head;
  Node* tail;
};

QueueList::QueueList()
:size(0),head(NULL),tail(NULL)
{
}

QueueList::~QueueList()
{}

void QueueList::enqueue(int elem)
{
```

```
  Node* temp = new Node();
  temp->data = elem;
  temp->next = NULL;
  temp->prev = NULL;

  if(size == 0)
  {
    head = temp;
    tail = temp;
    size++;
  }
  else if(size == 1)
  {
    head->next = temp;
    temp->prev = head;
    temp->next = NULL;
    tail = temp;
    size++;
    sort(size);
  }
  else
  {
    tail->next = temp;
    temp->prev = tail;
    temp->next = NULL;
    tail = temp;
    size++;
    sort(size);
  }
  //size++; //where to put this
  /*
  if(elem < head->data)
  {
    temp->next = head; //puts head in the beginning
    head->prev = temp;
    head = temp;
    size++;
  }

  if(elem
  */
}

void QueueList::sort(int length)
{
  Node* it = head;
  Node* temp = tail; //temporary is tail node
  Node* beforeIt; //beforeIt is it->prev node
  Node* beforeTail = tail->prev; //beforeTail is tail->prev node
```

```
  int tempdata;
  while(it->next != NULL)
  {
   //if data at iterator is smaller than data at tail move the pointer
   if(it->data < tail->data)
   {
    it = it->next;
   }
   //when size is 2 must do something diff.
   else if(size == 2)
   {
    //swap the data
    if(it->data > tail->data)
    {
     tempdata = head->data; //save head's data
     head->data = tail->data; //make head data tail's data
     tail->data = tempdata; //make tails data temps data
     it = it->next;
    }
    //otherwise just return and don't swap
    else
    {
     return;
    }
   }
   //it data will be = or > than tail's data
   else
   {

    beforeIt = it->prev;
    beforeIt->next = temp;
    temp->prev = beforeIt;
    temp->next = it;
    it->prev = temp;

    //cout << "test" << endl;
    tail = beforeTail;

    tail->next = NULL;

    //it = it->next;

   }
  }
}

void QueueList::print()
{
  Node* it = head;
```

```
  if(size != 0)
  {

  while(it->next != NULL)
  {

    cout << it->data << endl;
    it = it->next;
  }
    cout << it->data << endl;
  }
}

int QueueList::dequeue()
{
  int num; //the data I want to return
  if(size == 0)
  {
    cout << "Queue is Empty" << endl;
  }
  if(size == 1)
  {
    num = head->data;
    head = NULL;
    tail = NULL;
    size--;
    //cout << size << endl;
    return num;
  }
  else
  {
    num = head->data;
    //Node* temp = head;
    //delete head;
    head = head->next;
    //delete temp;
    size--;
    return num;
  }
  //Node* temp = head->next;
}

int main()
{
  QueueList test;
  cout << "Test enqueue" << endl;
  test.enqueue(2);
  test.enqueue(1);
  test.enqueue(4);
```

```
  test.enqueue(5);
  test.enqueue(3);
  test.print();
  cout << endl << "Test dequeue" << endl;
  test.dequeue();
  test.dequeue();
  test.print();
  //test.enqueue(3);
  //test.print();
  return 0;
}
```

```
//Exercise 3
/*Write a C++ class that implement two stacks using a single C++ array.
 * That is, it should have functions popFirst(…), popSecond(…),
 * pushFirst(…), pushSecond(…),… When out of space, double the size of
 * the array (similarly to what vector is doing).
 */

#include <iostream>

using namespace std;

//Two stacks on ONE array

//A stack of integers
class Stack
{
  private:
  int size;
  int capacity;
  int topOfStackFirst;
  int topOfStackSecond;
  int* theArray;

  public:
  Stack();
  Stack(int a);
  ~Stack();
  void pushFirst(int dataOne);
  void pushSecond(int dataTwo);
  void popFirst();
  void popSecond();
  void print();
  int expand(); //double size
};

/*
//Default Constructor are the int pointers NULL or -1?
Stack::Stack()
:size(0), sizeSecond(0), capacityFirst(0), capacitySecond(0),
topOfStackFirst(-1), topOfStackSecond(-1), theArrayFirst(NULL),
theArraySecond(NULL)
{
}
*/

/* Constructor with two parameters
 * @param sizeOne: Pass in size of first stack
 * @param sizeTwo: Pass in size of second stack
 */
```

```
Stack::Stack(int a)
:size(0),capacity(a), topOfStackFirst(-1),
topOfStackSecond(a), theArray(new int[a])
{
}

Stack::~Stack()
{
 delete [] theArray;
}

void Stack::pushFirst(int dataOne)
{
 if(size == capacity)
 {
  expand();
 }
 if(topOfStackFirst < topOfStackSecond - 1)
 {
  topOfStackFirst++;
  theArray[topOfStackFirst] = dataOne;
  size++;
 }
}

void Stack::pushSecond(int dataTwo)
{
 if(size == capacity)
 {
  expand();
 }
 if(topOfStackFirst < topOfStackSecond - 1)
 {
  topOfStackSecond--;
  theArray[topOfStackSecond] = dataTwo;
  size++;
 }
}

//these two delete the element at the top
void Stack::popFirst()
{
 if(topOfStackFirst >= 0)
 {
  int temp = theArray[topOfStackFirst];
  topOfStackFirst--;
  size--;
 }
}
```

```
void Stack::popSecond()
{
  if(topOfStackSecond < capacity)
  {
    int temp = theArray[topOfStackSecond];
    topOfStackSecond++;
    size--;
  }
}

void Stack::print()
{
  for(int i = 0; i < capacity; i++)
  {
    cout << theArray[i] << endl;
  }
}

//DOUBLES the size if the stack is full
int Stack::expand()
{
  int newCapacity = capacity * 2;
  int* newArray = new int[newCapacity];
  for(int i = 0; i <= topOfStackFirst; i++)
  {
    newArray[i] = theArray[i];
  }

  for(int j = capacity - 1; j >= topOfStackSecond; j--)
  {
    newArray[newCapacity - 1] = theArray[j];
    newCapacity--;
  }
  capacity = capacity * 2;
  delete [] theArray;
  theArray = new int[capacity];
  for(int k = 0; k < capacity; k++)
  {
    theArray[k] = newArray[k];
  }
  delete [] newArray;
  topOfStackSecond = capacity - topOfStackSecond;
}

int main()
{
  int size = 10;
  Stack a(size);
```

```
  a.pushFirst(4);
  a.pushSecond(2);
  a.popFirst(); //pop does not remove the element, it simply moves pointer
  a.pushFirst(3);
  //a.print();

  cout << endl;

  int size2 = 4;
  Stack b(size2);
  b.pushFirst(3);
  b.pushFirst(2);
  b.pushSecond(1);
  b.pushSecond(5);
  b.pushFirst(10);
  b.pushSecond(6);
  b.pushSecond(7);
  b.pushFirst(13);
  b.pushSecond(100);

  b.popSecond();
  b.pushSecond(14);
  b.print();
  return 0;
}
```