Assignement 2

Priority Queue and Double Stack

**Part of the source code is included here, but all of it is in the zip folder*

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.

Binary search is an algorithm for searching a certain value in a sorted container. For that problem, the array-based implementation would be the most efficient. Binary search requires computing and accessing the middle element of the container at each step, which is performed in constant time for an array $O(1)$ and in linear time $O(n)$ for the linked-list since we have to traverse the whole list to get the middle element. Thus, for the array-based implementation the overall complexity of binary search stays at $O(\log\ \mathrm{n})$ since we are dividing our search by 2 at each step, and $O(n * log\ n)$ for the linked-list implementation since we are searching the middle element at each step in linear time.

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?
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.
d. Can the modification in (c) improve the cost (not asymptotic but just execution time) of enqueue? Does it improve the asymptotic cost?
e. Is there any disadvantage that modification (c) incurs?

B. Hereunder are the implementations of the Enqueue and Dequeue functions:

```
void PriorityQueue::Enqueue(PriorityQueue *newNode)
{
    PriorityQueue *tempNode = this;
    while (tempNode->next && tempNode->next->data < newNode->data)
         tempNode = tempNode->next;
    // enqueing newNode
    PriorityQueue *tempNextNode = tempNode->next;
    newNode->SetNext(tempNextNode);
    tempNode->SetNext(newNode);
}
```

```
void PriorityQueue::Dequeue()
{
    if (this->next)
    {
        PriorityQueue *tempToDelete = this->next;
        this->next = this->next->next;
        tempToDelete->next = nullptr;
        delete tempToDelete;
    }
    else
    {
        cout << "List is empty." << endl;
    }
}
```

For the enqueue function, we are traversing the linked list until we find a spot for the newNode. In the worst case, we could traverse the whole list, so that part runs in $O(n)$ asymptotic time. Setting the next node of tempNext to newNode runs in $O(1)$ so the overall time complexity of Enqueue is $O(n)$.

For the dequeue function, we are assigning next pointer of current node to the second next and deleting the pointer on the dequeued node. Those operations run in constant time thus the complexity is $O(1)$.

D. Code for Enqueue and Dequeue with a pointer to the 10th next node:

```
void PriorityQueue::Enqueue(PriorityQueue *newNode)
{
    // keeping two references for processing
    PriorityQueue *tempNode = this;
    PriorityQueue *temp10 = this;
    int count = 0;
    // while the node has a lower priority
    while (tempNode->next && tempNode->next->data < newNode->data)
    {
        // if we can jump 10 nodes
        if (tempNode->tenthNext && tempNode->tenthNext->data < newNode-
>data)
        {
            // if we know we already have more than ten nodes, the temp10 ju
mps also by 10 nodes
            if (count >= 10)
                temp10 = temp10->tenthNext;
            else
                temp10 = temp10->next;
            tempNode = tempNode->tenthNext;
            count += 10;
        }
```

```
        else
        {
            if (count >= 10)
                temp10 = temp10->next;
            tempNode = tempNode->next;
            count += 1;
        }
    }
    // enqueing newNode
    tempNode->SetNext(newNode);
    // resetting tenthNode pointers
    while (tempNode->next)
    {
        if (count >= 10)
        {
            temp10->next->tenthNext = tempNode->next;
            temp10 = temp10->next;
        }

        tempNode = tempNode->next;
        count++;
    }
}
void PriorityQueue::Dequeue()
{
    if (this->next)
    {
        PriorityQueue *tempToDelete = this->next;
        this->next = this->next->next;
        tempToDelete->next = nullptr;
        tempToDelete->tenthNext = nullptr;
        delete tempToDelete;
    }
    else
    {
        cout << "List is empty." << endl;
    }
}
```

The added 10th pointer improves the execution time for inputs having more than 10 elements. Let's consider that we have a queue with 100000 elements with the lowest priority element being 100000 and we want to insert a node with value 100001. Without the 10th pointer, we would have to traverse the whole list so we would have to make $n = 100000\ iterations$. With the 10th pointer, at each step, we could jump by 10 nodes, we would be making $n/10\ operations$.

But it wouldn't improve the overall asymptotic complexity of the Enqueue method because $n\ operations \sim O(n)\ and\ n/10 \sim O(n)$.

E. The downsides of having a 10th next pointer are three folded. First, the Enqueue function is a bit less self-explanatory, and it's important to keep our primitive functions readable. A more important aspect is that even if we traverse $n/10$ nodes in best case, if we have a worst case where we insert a value in the middle, we would still have to traverse the list another time from the node we just inserted to reset 10th next pointers of the next nodes. The third aspect follows this idea, because choosing the 10th next pointer is an arbitrary choice. What we could do instead is, determine the optimal k parameter corresponding to the kth next pointer as a function of the final size and range of the data to be inserted.

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).

```
#ifndef DOUBLESTACK_H
#define DOUBLESTACK_H

#include <iostream>
#include <string>
#include <vector>

using std::cin;
using std::cout;
using std::endl;
using std::string;
using std::vector;

template <typename T>
class DoubleStack
{
private:
    T *data;
    int firstTop;
    int secondTop;
    int size;
    int firstSize;
    int secondSize;
    void doubleLength();

public:
    DoubleStack();
    ~DoubleStack();
    // accessors
    int GetSize() const;
    int GetFirstSize() const;
    int GetSecondSize() const;
```

```
    const T *GetData() const;

    // member methods
    void pushFirst(T);
    void pushSecond(T);
    void popFirst();
    void popSecond();
    // optionnal
    const T peekFirst() const;
    const T peekSecond() const;
    const bool emptyFirst() const;
    const bool emptySecond() const;

    const void print() const;
};

template <typename T>
DoubleStack<T>::DoubleStack()
{
    this->size = 0;
    this->firstTop = -1;
    this->secondTop = -1;
    this->firstSize = 0;
    this->secondSize = 0;
    this->data = new T[0];
}

template <typename T>
DoubleStack<T>::~DoubleStack()
{
    delete[] this->data;
}

template <typename T>
int DoubleStack<T>::GetSize() const
{
    return this->size;
}

template <typename T>
const T *DoubleStack<T>::GetData() const
{
    return this->data;
}

template <typename T>
int DoubleStack<T>::GetFirstSize() const
```

```
template <typename T>
int DoubleStack<T>::GetFirstSize() const
{
    return this->firstSize;
}

template <typename T>
int DoubleStack<T>::GetSecondSize() const
{
    return this->secondSize;
}

template <typename T>
void DoubleStack<T>::doubleLength()
{
    int newSize = (this->size) * 2;
    T *newArray = new T[newSize];
    this->secondTop = newSize - this->secondSize - 1;

    for (int i = 0; i < newSize; i++)
    {
        if (i < this->firstTop)
            newArray[i] = this->data[i];
        else if (i > this->secondTop)
            newArray[i] = this->data[i - this->size];
        else
            newArray[i] = T();
    }
    this->size = newSize;
    this->data = newArray;
}

// member methods
template <typename T>
void DoubleStack<T>::pushFirst(T value)
{

    if (this->size == 0)
    {
        T *newData = new T[3];
        newData[0] = value;
        newData[1] = T();
        newData[2] = T();
        this->data = newData;
        this->firstTop = 1;
        this->secondTop = 2;
        this->size = 3;
```

```
    }
    else
    {
        if (this->firstSize + this->secondSize + 2 == this->size)
            this->doubleLength();
        int ind = this->firstTop;
        this->data[ind] = value;
        this->firstTop++;
    }
    this->firstSize++;
}

template <typename T>
void DoubleStack<T>::pushSecond(T value)
{
    if (this->size == 0)
    {
        T *newData = new T[3];
        newData[2] = value;
        newData[1] = T();
        newData[0] = T();
        this->data = newData;
        this->secondTop = 1;
        this->firstTop = 0;
        this->size = 3;
    }
    else
    {
        if (this->firstSize + this->secondSize + 2 == this->size)
            this->doubleLength();
        this->data[this->secondTop] = value;
        this->secondTop--;
    }
    this->secondSize++;
}

template <typename T>
void DoubleStack<T>::popFirst()
{
    if (this->firstSize > 0)
    {
        this->firstSize--;
        this->firstTop--;
    }
}
```

```
template <typename T>
void DoubleStack<T>::popSecond()
{
    if (this->secondSize > 0)
    {
        this->secondSize--;
        this->secondTop++;
    }
}

template <typename T>
const void DoubleStack<T>::print() const
{
    for (int i = 0; i < this->size; i++)
        if (i < this->firstTop || i > this->secondTop)
            cout << this->data[i];
        else
            cout << "0";
    cout << endl;
}

template <typename T>
const T DoubleStack<T>::peekFirst() const
{
    if (!this->emptyFirst())
        return this->data[this->firstTop - 1];
    else
        return T();
}

template <typename T>
const T DoubleStack<T>::peekSecond() const
{
    if (!this->emptySecond())
        return this->data[this->secondTop + 1];
    else
        return T();
}

template <typename T>
const bool DoubleStack<T>::emptyFirst() const
{
    return this->firstSize == 0;
}
```

```
template <typename T>
const bool DoubleStack<T>::emptySecond() const
{
    return this->secondSize == 0;
}
#endif
```