Problem 8.1:

For full implementation: see Stack.h

```
template <class T> class StackQueue;
template <class T> class Stack {

private:
        struct StackNode { // linked list
                T data;
                std::shared_ptr<StackNode> next;
        };
        std::shared_ptr<StackNode> top; // top of stack
        int size; // -1 if not set, value otherwise
        int current_size; // unused if size = -1
public:

        void push(T x){
                if (current_size == size && size > 0){
                        std::cout<<"stack overflow!\n";
                        //throw std::logic_error("stack overflow!");
                        return;
                }

                std::shared_ptr<StackNode> newel = std::make_shared<StackNode>();
                if (!newel)
                        throw std::bad_alloc();
                newel->data = x;
                newel->next = top;
                top = newel;
                current_size++;
                //std::cout<<"inserting element "<<x<<" in stack!\n";
        }

        bool pop(T& out){
                if (current_size > 0) {
                        if (top == nullptr){
                                std::cout<<"null pointer! stack underflow!";
                                return false;
                        }
                        out = top->data;
                        std::shared_ptr<StackNode> next = top->next;
                        current_size--;
                        top = next;
                        //std::cout<<"popping element "<< out << " from stack\n";

                }
                else{
```

```
                        std::cout<<"stack underflow!\n";

                        //throw std::logic_error("stack underflow!");
                        return false;
                }
                return true;
        }

        bool isEmpty(){
                return current_size == 0;
        } // true if empty, false otherwise
        //Stack(int new_size)

        Stack(){
                size = -1;
                current_size = 0;
                top = nullptr;
        }
        Stack(int max){
                size = max;
                current_size = 0;
                top = nullptr;
        }

        friend class StackQueue<T>;

};
```

(a) push: $T(n) = \Theta(1)$. The time taken to push element is constant since it simply requires inserting it at the top of the stack. No loops nor recursive calls are needed.

Pop: $T(n) = \Theta(1)$. The operation of popping the stack is constant because it requires no loops or recursive calls, it simply removes the top of the stack, which is directly accessible via the top pointer.

isEmpty: $T(n) = \Theta(1)$: checking whether a stack is empty simply checks whether the current_size variable equals zero.

Constructors: $T(n) = \Theta(1)$: constructors simply initialise variables

StackQueue

```
template <class T> class StackQueue{
private:
        Stack<T> exit;
        Stack<T> entrance;
        int current_size;
        int max_size;
public:
        StackQueue(){
                exit = Stack<T>();
                entrance = Stack<T>();

                current_size =0;
```

```
                max_size= -1;
        }

        StackQueue(int max){
                exit = Stack<T>(max);
                entrance = Stack<T>(max);
                current_size =0;
                max_size= max;
        }
        void enqueue(T& x){
                if(current_size < max_size || max_size < 0) {
                        T trash;
                        while (!entrance.isEmpty()){
                                exit.push(entrance.top->data);
                                entrance.pop(trash);
                        }
                        entrance.push(x);
                        while(!exit.isEmpty()){
                                entrance.push(exit.top->data);
                                exit.pop(trash);
                        }
                        current_size++;
                }
                else{
                        std::cout<<"queue overflow!\n";
                }
        }
        bool dequeue(T& out){
                if (!entrance.isEmpty()) {
                        entrance.pop(out);
                        current_size--;
                        return true;

                }
                else{
                        std::cout<<"queue underflow!\n";
                        return false;
                }
        }
        bool isEmpty(){
                return current_size == 0;
        }


};
```

Enqueue: $T(n) = \Theta(n)$: inserting an element requires passing all of the elements of one stack to the other one, inserting the new element in the now empty stack, and then pushing all of the elements of the auxiliary stack back into the main stack. The time complexity of this operation is dependent on the number of elements in the stack due to the existence of a while loop, making the complexity $\Theta(n)$.

Dequeue: $T(n) = \Theta(1)$: after the insertion process is completed, the main stack acts like a queue in the sense that the element popped from the stack is the first element to have "entered" the queue-stack structure. Therefore, the time complexity is $\Theta(1)$ as this is a simple pop() operation.

Problem 8.2:

(a) (3 points) Write down the pseudocode for an in-situ algorithm that reverses a linked list of n elements in Θ(n). Explain why it is an in-situ algorithm

```
Procedure reversify(A)
        if A.start == A.end
                return
        cursor2 = A.start
        cursor1 = nil;
        cursor3 = nil
        while cursor2 != nil
                cursor3 = cursor2.next
                cursor2.next = cursor1

                cursor1 = cursor2
                cursor2 = cursor3
        A.head = cursor1
```

This algorithm is in situ because it does not need auxiliary memory to reverse the list proportional to the size of the list. The amount of auxiliary memory needed is constant and is equivalent to the three cursor pointers.

(b) For implementation, see LinkedLists.h . The conversion algorithm is implemented as constructor of a Linked List taking a simple BST.

c) For the bonus problem, I implemented a red-black binary tree, which guarantees the time complexity of search methods(not implemented yet) to be O(logn). Since for every element of the Linked List, the element must be inserted in the binary tree via a for loop, and insertion has a time complexity of $O(\log(n))$, then the complexity of this procedure is $O(n \log n)$.