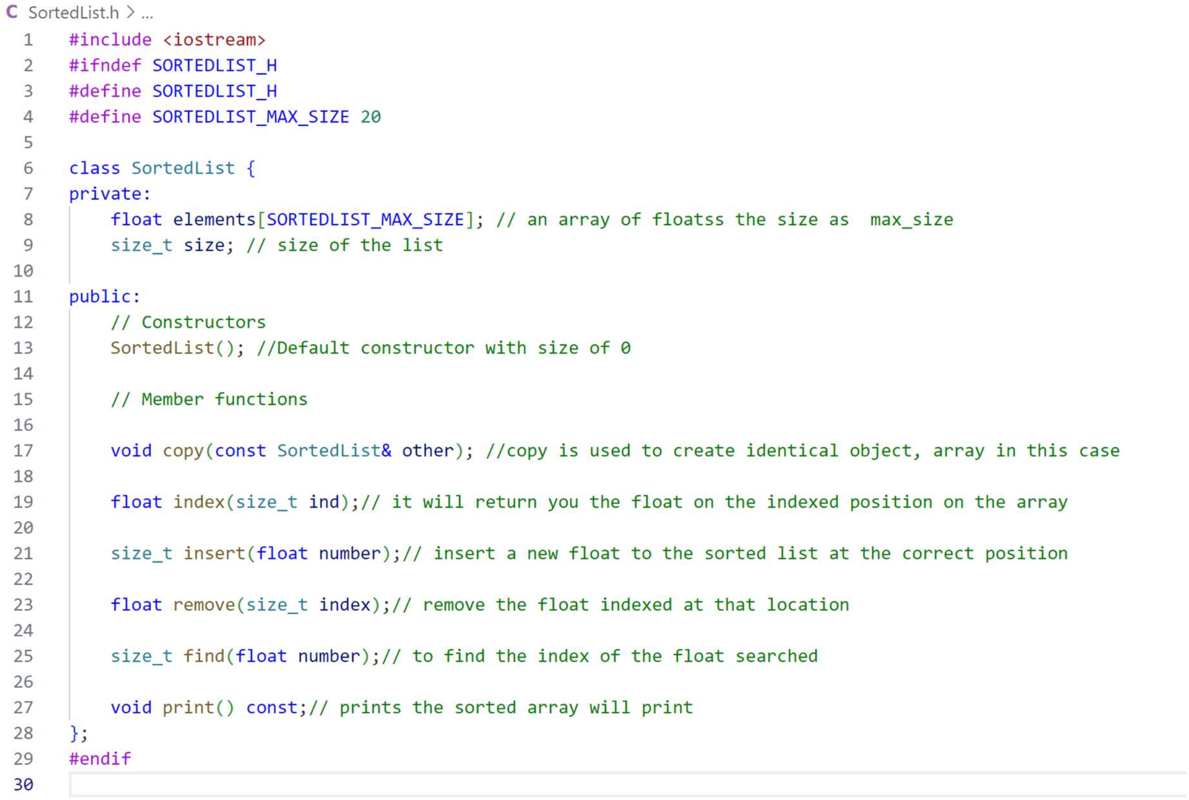
Date: 08/11/2024

Programing Assignment 1 EE441

Mert Ekren

Q1



SortedList.h > ...

```cpp
#include <iostream>
#ifndef SORTEDLIST_H
#define SORTEDLIST_H
#define SORTEDLIST_MAX_SIZE 20

class SortedList {
private:
    float elements[SORTEDLIST_MAX_SIZE]; // an array of floatss the size as  max_size
    size_t size; // size of the list

public:
    // Constructors
    SortedList(); //Default constructor with size of 0

    // Member functions

    void copy(const SortedList& other); //copy is used to create identical object, array in this case

    float index(size_t ind);// it will return you the float on the indexed position on the array

    size_t insert(float number);// insert a new float to the sorted list at the correct position

    float remove(size_t index);// remove the float indexed at that location

    size_t find(float number);// to find the index of the float searched

    void print() const;// prints the sorted array will print
};
#endif

```

Q2

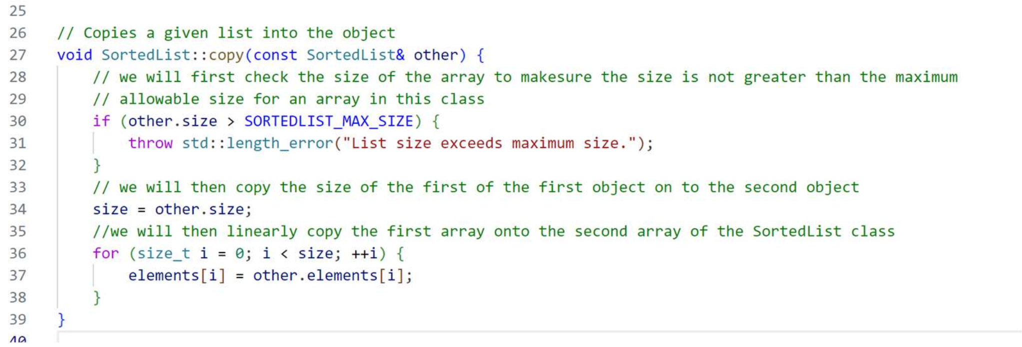
The default constructor initializes an empty list by setting the size attribute to 0. This means the list starts with no elements.

```cpp
// Constructors

SortedList::SortedList() : size(0) {} // default constructer will create with the size 0
```

Q3

The copy function replicates the contents of another SortedList instance into the current object. It first checks that the size of other does not exceed the maximum allowable size. If valid, it copies both the size and elements from other to the current list.

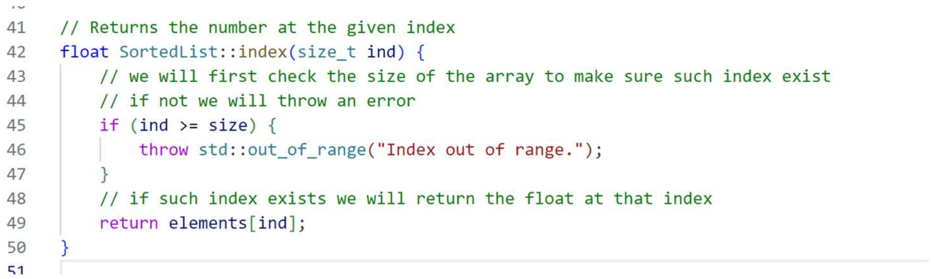
Date: 08/11/2024



```cpp

// Copies a given list into the object
void SortedList::copy(const SortedList& other) {
    // we will first check the size of the array to makesure the size is not greater than the maximum
    // allowable size for an array in this class
    if (other.size > SORTEDLIST_MAX_SIZE) {
        throw std::length_error("List size exceeds maximum size.");
    }
    // we will then copy the size of the first of the first object on to the second object
    size = other.size;
    //we will then linearly copy the first array onto the second array of the SortedList class
    for (size_t i = 0; i < size; ++i) {
        elements[i] = other.elements[i];
    }
}
```

Q4

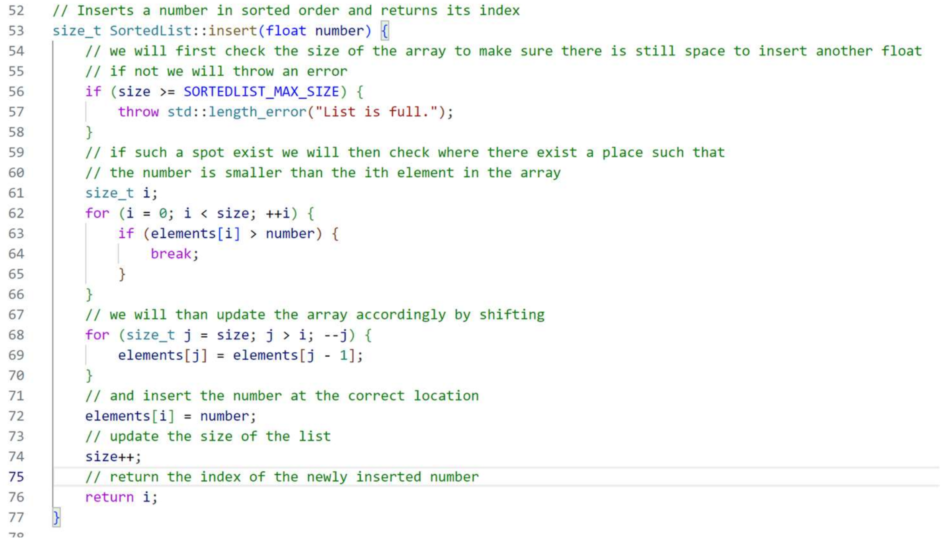
The index function returns the element at a specific index. It first checks if the provided index is less than the list's current size. If the index is valid, it returns the element; otherwise, it throws an exception.



```cpp
// Returns the number at the given index
float SortedList::index(size_t ind) {
    // we will first check the size of the array to make sure such index exist
    // if not we will throw an error
    if (ind >= size) {
        throw std::out_of_range("Index out of range.");
    }
    // if such index exists we will return the float at that index
    return elements[ind];
}
```

Q5

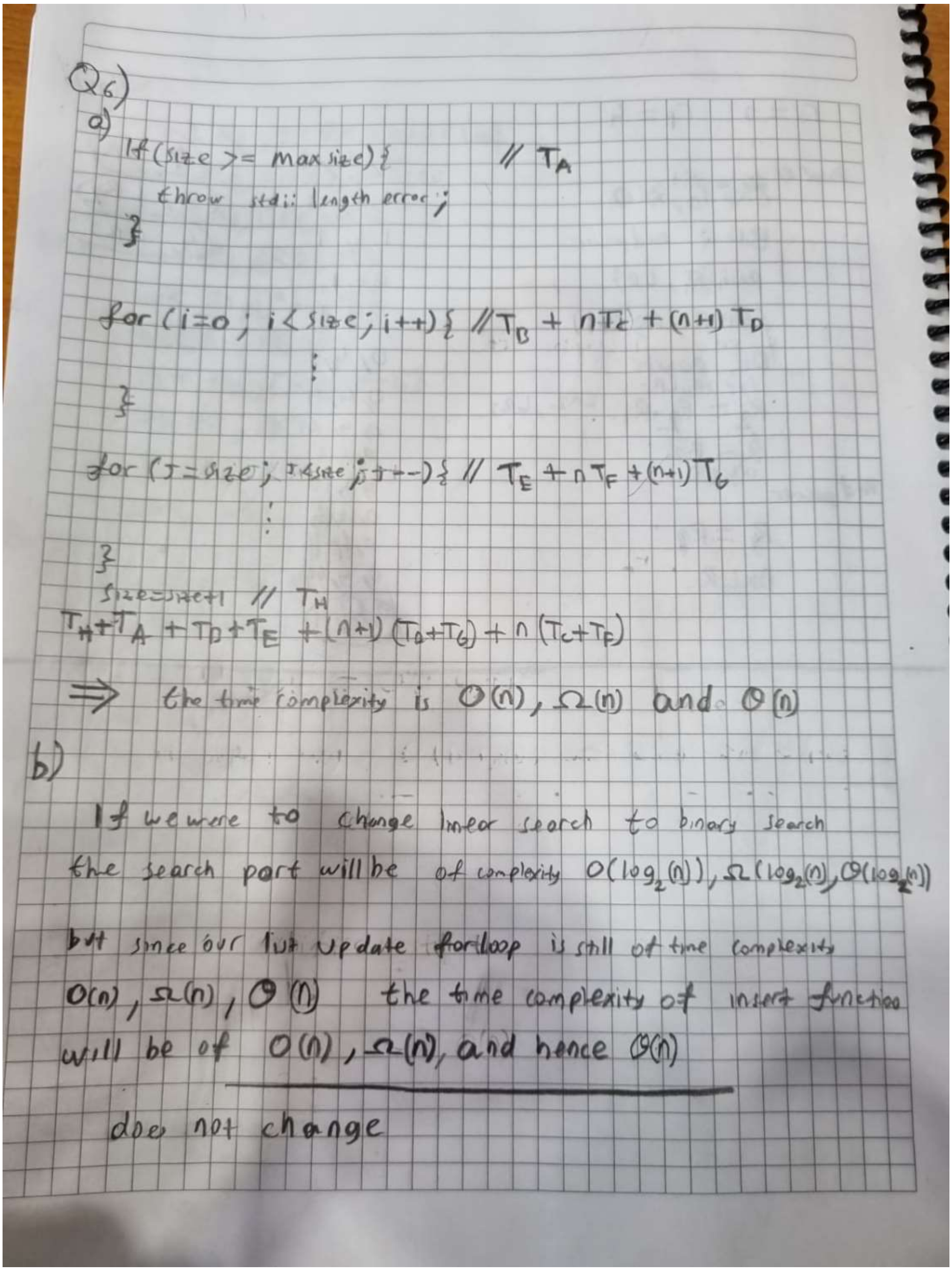
The insert function adds a new float to the list in a way that maintains the sorted order. It first ensures there is space for a new element by checking against SORTEDLIST_MAX_SIZE. The function finds the appropriate position for the new element, shifts existing elements to make space, inserts the new element, increments the size, and returns the new element's index.



```cpp
// Inserts a number in sorted order and returns its index
size_t SortedList::insert(float number) {
    // we will first check the size of the array to make sure there is still space to insert another float
    // if not we will throw an error
    if (size >= SORTEDLIST_MAX_SIZE) {
        throw std::length_error("List is full.");
    }
    // if such a spot exist we will then check where there exist a place such that
    // the number is smaller than the ith element in the array
    size_t i;
    for (i = 0; i < size; ++i) {
        if (elements[i] > number) {
            break;
        }
    }
    // we will than update the array accordingly by shifting
    for (size_t j = size; j > i; --j) {
        elements[j] = elements[j - 1];
    }
    // and insert the number at the correct location
    elements[i] = number;
    // update the size of the list
    size++;
    // return the index of the newly inserted number
    return i;
}
```

Date: 08/11/2024

Q6



Q6)

a)

```
If(size >= max size){            // T_A
    throw std:: length error;
}

for (i=0; i<size; i++){ // T_B + nT_C + (n+1) T_D
        :
}

for (j=size; j<size ; j--){ // T_E + n T_F + (n+1) T_G
        :
}
size=size+1  // T_H
```

$T_H + T_A + T_B + T_E + (n+1)(T_D + T_G) + n(T_C + T_F)$

$\Rightarrow$ the time complexity is $O(n)$, $\Omega(n)$ and $\Theta(n)$

b)

If we were to change linear search to binary search the search part will be of complexity $O(\log_2(n))$, $\Omega(\log_2(n)$, $\Theta(\log_2(n))$

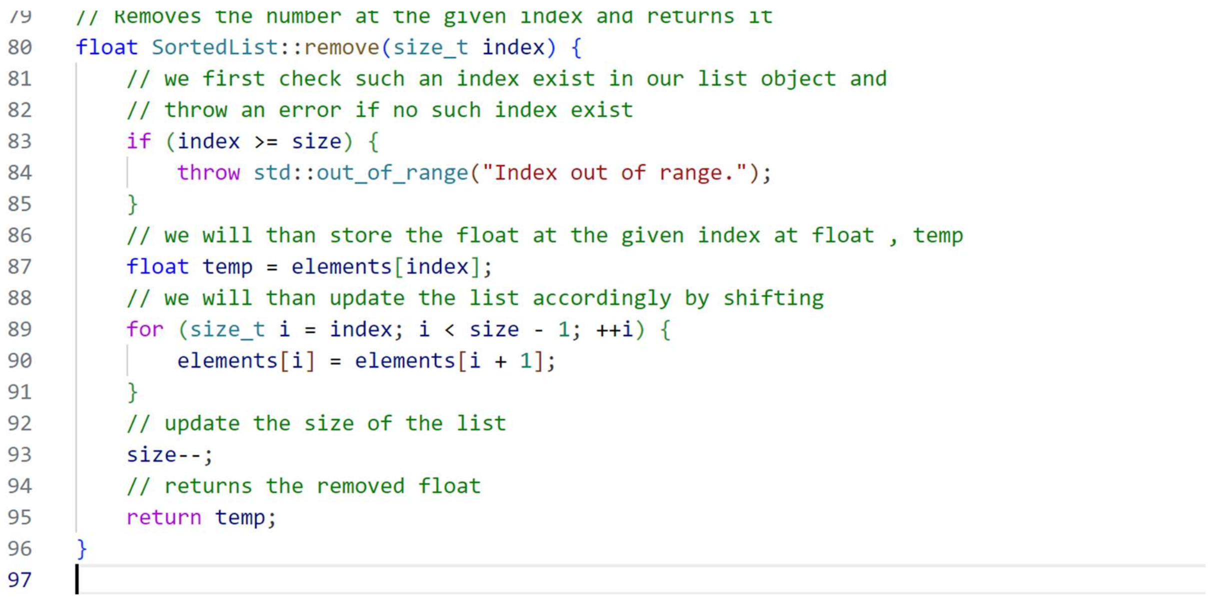
but since our list update forloop is still of time complexity $O(n)$, $\Omega(n)$, $\Theta(n)$ the time complexity of insert function will be of $O(n)$, $\Omega(n)$, and hence $\Theta(n)$

does not change

Q7

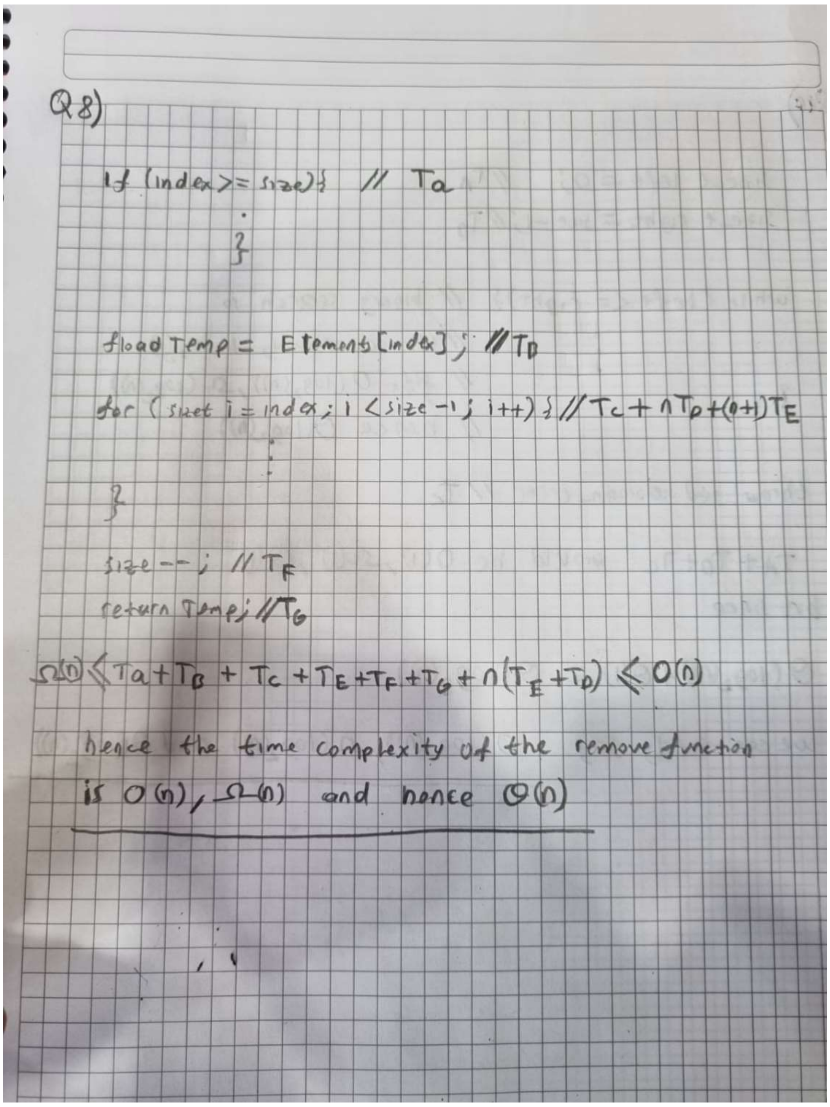
The remove function deletes an element at a specified index and returns it. It first checks that the index is within range; if valid, it stores the element at index, shifts subsequent elements to fill the gap, decrements the size, and returns the removed value. An std::out_of_range exception is thrown if the index is doesn't exist or is invalid.

Date: 08/11/2024



```cpp
// Removes the number at the given index and returns it
float SortedList::remove(size_t index) {
    // we first check such an index exist in our list object and
    // throw an error if no such index exist
    if (index >= size) {
        throw std::out_of_range("Index out of range.");
    }
    // we will than store the float at the given index at float , temp
    float temp = elements[index];
    // we will than update the list accordingly by shifting
    for (size_t i = index; i < size - 1; ++i) {
        elements[i] = elements[i + 1];
    }
    // update the size of the list
    size--;
    // returns the removed float
    return temp;
}

```

Q8



Q8)

```
If (index >= size){   // Ta
        .
        }

float Temp =  Elements[index]; // TB

for ( sizet i = index; i < size -1; i++) { // Tc + nTD + (n+1)TE
        :
}

size--; // TF
return Temp; // TG
```

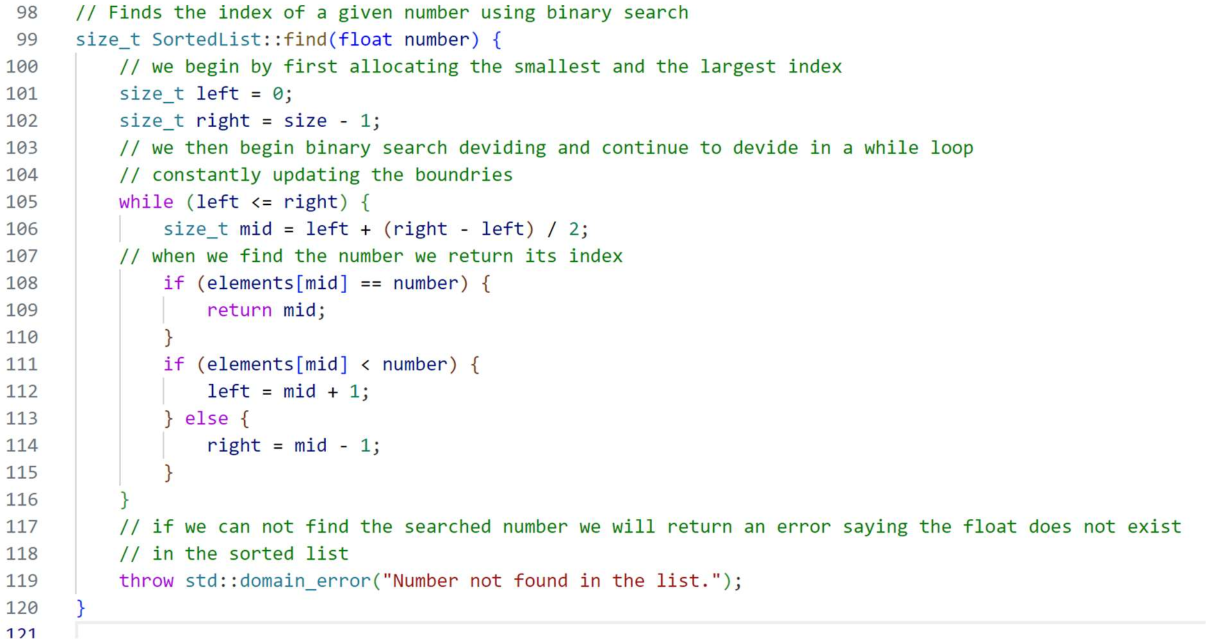
$\Omega(n) \leqslant Ta + T_B + T_c + T_E + T_F + T_G + n(T_E + T_D) \leqslant O(n)$

hence the time complexity of the remove function is $O(n)$, $\Omega(n)$ and hence $\Theta(n)$

Date: 08/11/2024

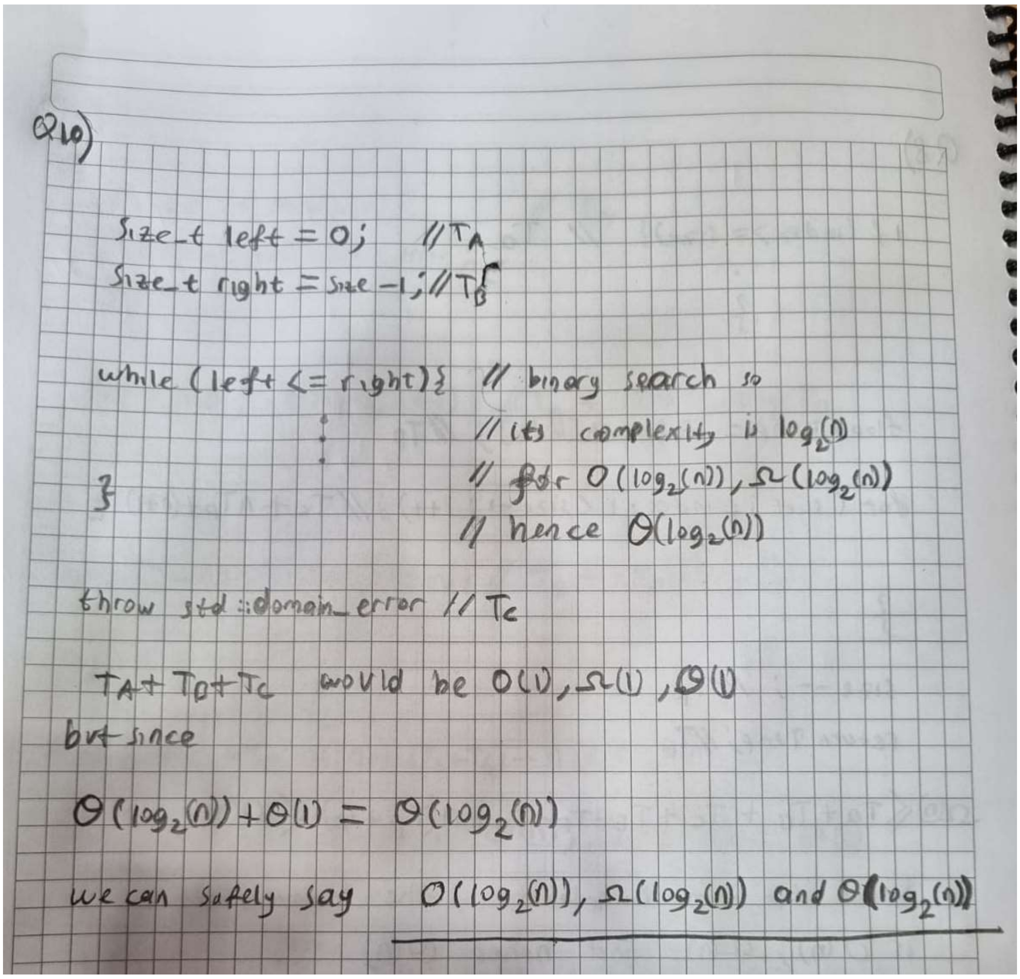
Q9

This function searches for a given number using binary search, which is really efficient on a sorted list. It sets up the initial search boundaries and iteratively narrows them until it either finds the number (returning its index) or it cant find the number, throwing a std::domain_error.



```
// Finds the index of a given number using binary search
size_t SortedList::find(float number) {
    // we begin by first allocating the smallest and the largest index
    size_t left = 0;
    size_t right = size - 1;
    // we then begin binary search deviding and continue to devide in a while loop
    // constantly updating the boundries
    while (left <= right) {
        size_t mid = left + (right - left) / 2;
    // when we find the number we return its index
        if (elements[mid] == number) {
            return mid;
        }
        if (elements[mid] < number) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    // if we can not find the searched number we will return an error saying the float does not exist
    // in the sorted list
    throw std::domain_error("Number not found in the list.");
}
```

Q10



Q10)

Size_t left = 0; // T_A

Size_t right = size -1; // T_B

while (left <= right){ // binary search so

// its complexity is $\log_2(n)$

} // for $O(\log_2(n))$, $\Omega(\log_2(n))$

// hence $\Theta(\log_2(n))$

throw std::domain_error // T_c

$T_A + T_B + T_c$ would be $O(1)$, $\Omega(1)$, $\Theta(1)$

but since

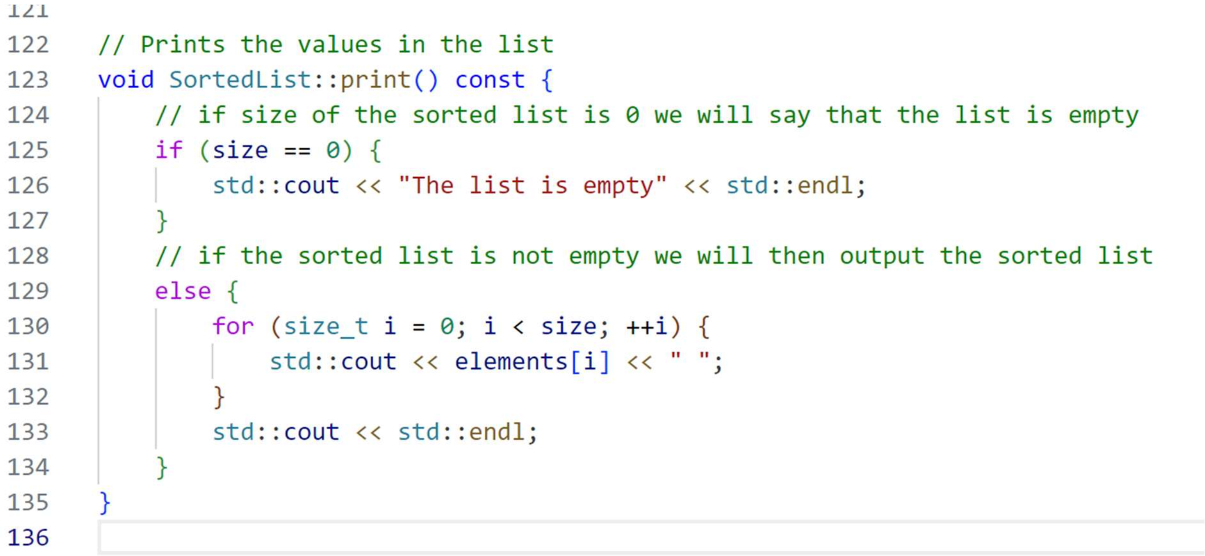
$\Theta(\log_2(n)) + \Theta(1) = \Theta(\log_2(n))$

we can safely say $O(\log_2(n))$, $\Omega(\log_2(n))$ and $\Theta(\log_2(n))$

Date: 08/11/2024

Q11

The print function outputs the elements in the list. If the list is empty, it prints a message indicating this. If it has elements in it it iterates over the array, printing each element in sorted order.

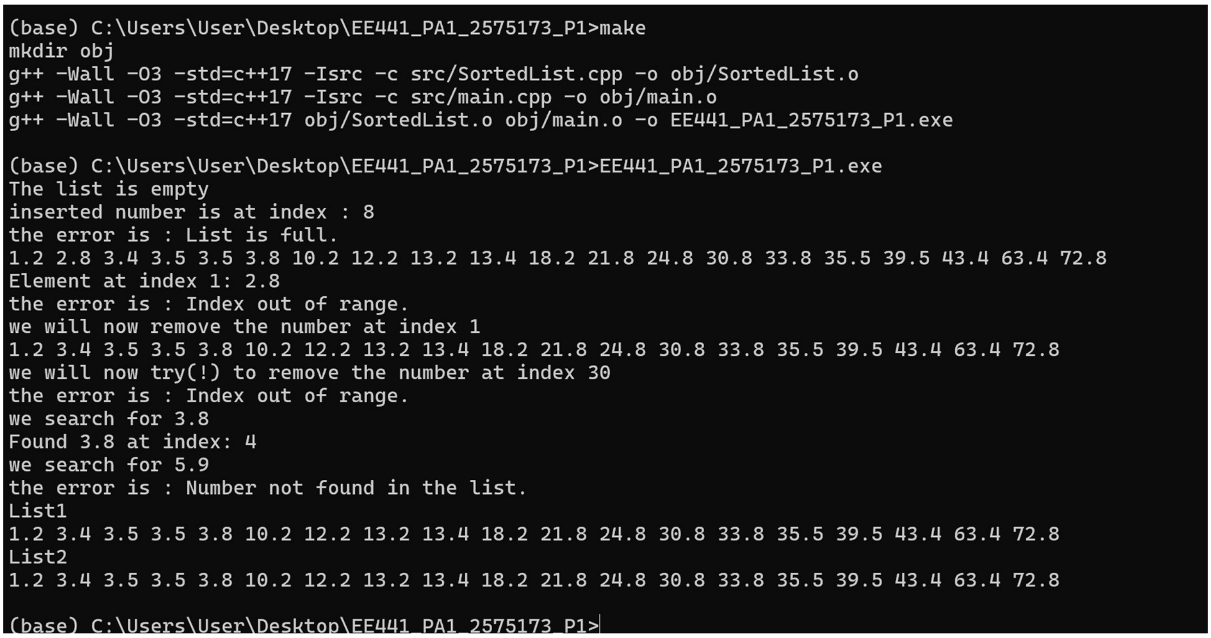


```
// Prints the values in the list
void SortedList::print() const {
    // if size of the sorted list is 0 we will say that the list is empty
    if (size == 0) {
        std::cout << "The list is empty" << std::endl;
    }
    // if the sorted list is not empty we will then output the sorted list
    else {
        for (size_t i = 0; i < size; ++i) {
            std::cout << elements[i] << " ";
        }
        std::cout << std::endl;
    }
}
```

Q12

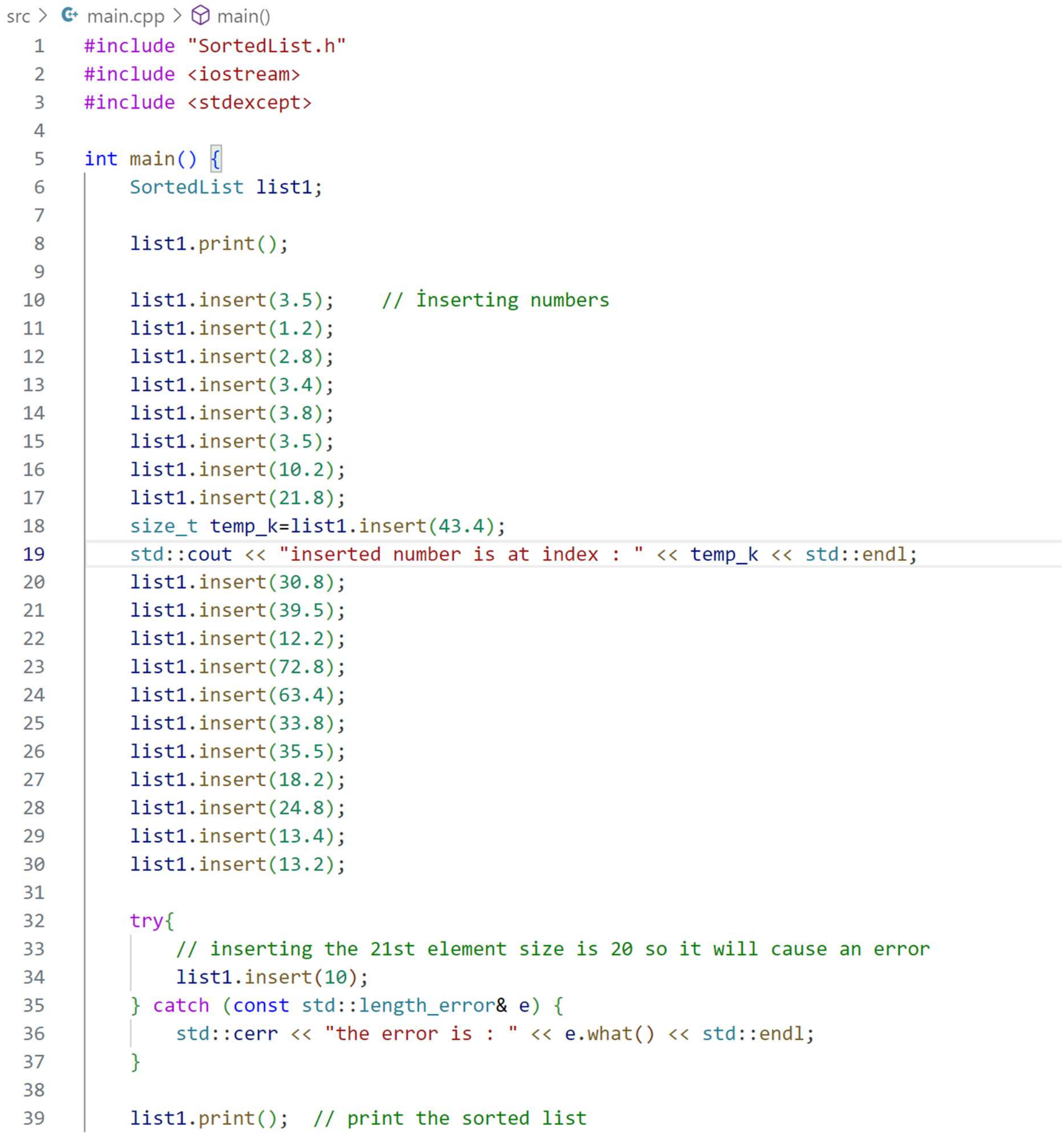
In the main function we began by creating a list then we insert some random numbers then observe that it is sorted in ascending values we then try the member functions index, remove, find, copy and print. And observe those functions also work as expected.

As we can see the Class performs all the required functions and constructors as expected.



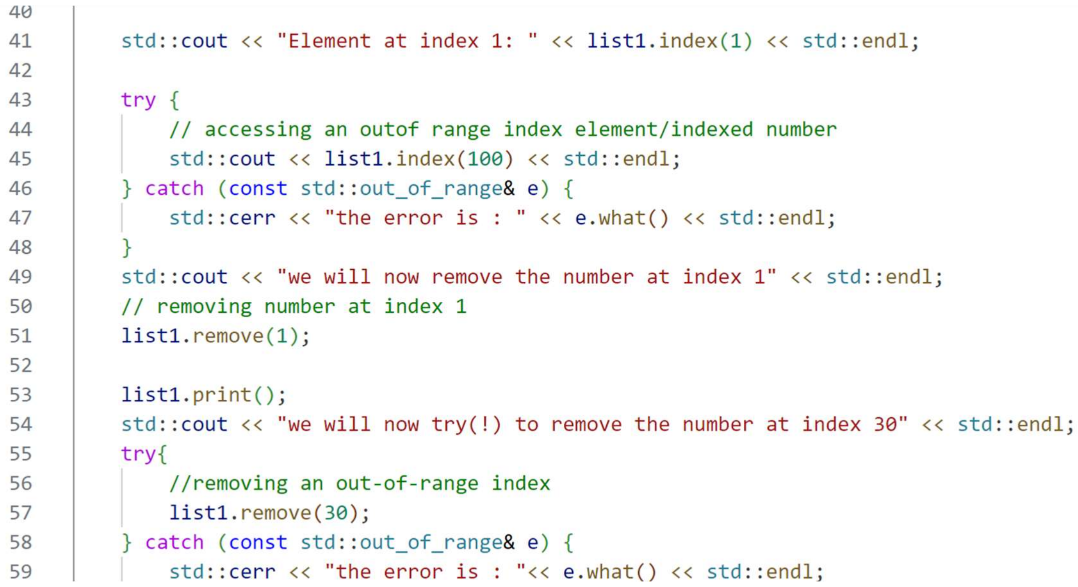
```
(base) C:\Users\User\Desktop\EE441_PA1_2575173_P1>make
mkdir obj
g++ -Wall -O3 -std=c++17 -Isrc -c src/SortedList.cpp -o obj/SortedList.o
g++ -Wall -O3 -std=c++17 -Isrc -c src/main.cpp -o obj/main.o
g++ -Wall -O3 -std=c++17 obj/SortedList.o obj/main.o -o EE441_PA1_2575173_P1.exe

(base) C:\Users\User\Desktop\EE441_PA1_2575173_P1>EE441_PA1_2575173_P1.exe
The list is empty
inserted number is at index : 8
the error is : List is full.
1.2 2.8 3.4 3.5 3.5 3.8 10.2 12.2 13.2 13.4 18.2 21.8 24.8 30.8 33.8 35.5 39.5 43.4 63.4 72.8
Element at index 1: 2.8
the error is : Index out of range.
we will now remove the number at index 1
1.2 3.4 3.5 3.5 3.8 10.2 12.2 13.2 13.4 18.2 21.8 24.8 30.8 33.8 35.5 39.5 43.4 63.4 72.8
we will now try(!) to remove the number at index 30
the error is : Index out of range.
we search for 3.8
Found 3.8 at index: 4
we search for 5.9
the error is : Number not found in the list.
List1
1.2 3.4 3.5 3.5 3.8 10.2 12.2 13.2 13.4 18.2 21.8 24.8 30.8 33.8 35.5 39.5 43.4 63.4 72.8
List2
1.2 3.4 3.5 3.5 3.8 10.2 12.2 13.2 13.4 18.2 21.8 24.8 30.8 33.8 35.5 39.5 43.4 63.4 72.8

(base) C:\Users\User\Desktop\EE441_PA1_2575173_P1>
```

Date: 08/11/2024

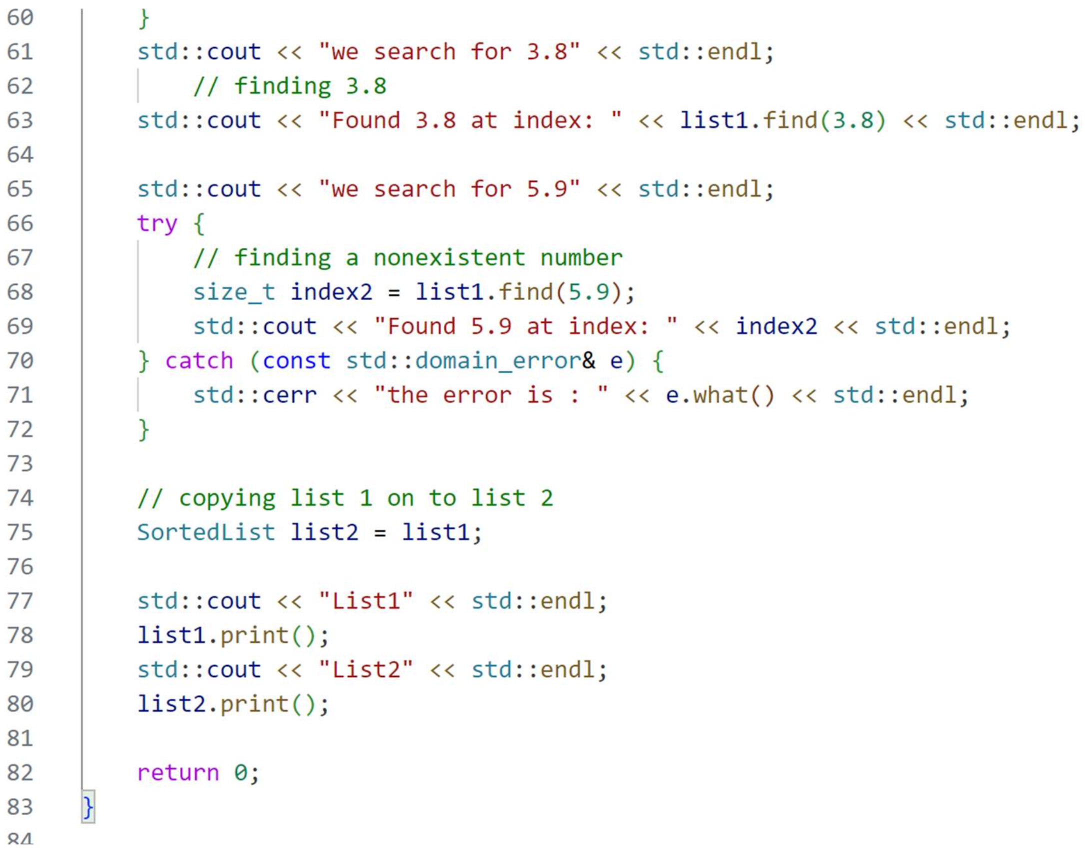


```cpp
#include "SortedList.h"
#include <iostream>
#include <stdexcept>

int main() {
    SortedList list1;

    list1.print();

    list1.insert(3.5);    // İnserting numbers
    list1.insert(1.2);
    list1.insert(2.8);
    list1.insert(3.4);
    list1.insert(3.8);
    list1.insert(3.5);
    list1.insert(10.2);
    list1.insert(21.8);
    size_t temp_k=list1.insert(43.4);
    std::cout << "inserted number is at index : " << temp_k << std::endl;
    list1.insert(30.8);
    list1.insert(39.5);
    list1.insert(12.2);
    list1.insert(72.8);
    list1.insert(63.4);
    list1.insert(33.8);
    list1.insert(35.5);
    list1.insert(18.2);
    list1.insert(24.8);
    list1.insert(13.4);
    list1.insert(13.2);

    try{
        // inserting the 21st element size is 20 so it will cause an error
        list1.insert(10);
    } catch (const std::length_error& e) {
        std::cerr << "the error is : " << e.what() << std::endl;
    }

    list1.print();  // print the sorted list
```

Date: 08/11/2024



```
    std::cout << "Element at index 1: " << list1.index(1) << std::endl;

    try {
        // accessing an outof range index element/indexed number
        std::cout << list1.index(100) << std::endl;
    } catch (const std::out_of_range& e) {
        std::cerr << "the error is : " << e.what() << std::endl;
    }
    std::cout << "we will now remove the number at index 1" << std::endl;
    // removing number at index 1
    list1.remove(1);

    list1.print();
    std::cout << "we will now try(!) to remove the number at index 30" << std::endl;
    try{
        //removing an out-of-range index
        list1.remove(30);
    } catch (const std::out_of_range& e) {
        std::cerr << "the error is : "<< e.what() << std::endl;
```



```
    }
    std::cout << "we search for 3.8" << std::endl;
        // finding 3.8
    std::cout << "Found 3.8 at index: " << list1.find(3.8) << std::endl;

    std::cout << "we search for 5.9" << std::endl;
    try {
        // finding a nonexistent number
        size_t index2 = list1.find(5.9);
        std::cout << "Found 5.9 at index: " << index2 << std::endl;
    } catch (const std::domain_error& e) {
        std::cerr << "the error is : " << e.what() << std::endl;
    }

    // copying list 1 on to list 2
    SortedList list2 = list1;

    std::cout << "List1" << std::endl;
    list1.print();
    std::cout << "List2" << std::endl;
    list2.print();

    return 0;
}
```