CS214: Assignment #2 - 2021

Data Structure Course

Deadline & Submission

At least one team member should submit the compressed group solution as zip file containing the programs on blackboard –>A2 Task(name your assignment file ("A2_ID1_ID2_ID3_ID4_ID5.zip").
Groups must be from the same lab
The deadline for submitting the electronic solution of this assignment is 12/6/2021

About this Assignment

1. This assignment will be solved in teams from 3 to 5.
2. All team members should understand all assignment problems.
3. All code must be standard ANSI C++.
4. Assume any missing details and search for information yourself.
5. Any cheating in any part of the assignment is the responsibility of the whole team and the whole team will be punished.

Section 1:

Name a folder "**Section1**" and put all your section 1 problems folder inside it.

Problem 1:

We know that both Merge and Quick sorts are O (n log n) in average case. But Quick sort is faster. We like to compare their performance and study how faster Quick sort is. Is it two times faster? Or three times faster? Or what? You task will be:

1. Implement quick and merge sort algorithms given in the lecture or book. Understand them very well. Choose a good technique for choosing the pivot of Quick sort.
2. Implement a program that generates arrays (or better vector) of random integer data of size 5,000 items, 10,000 items, etc till 100,000 items.
3. For each data array, run both algorithms and record the time it takes in sec or msec.
4. Plot the performance of the algorithms against each other on the same graph.

Name a folder "**P1**" and put your files inside it (even if it's only one file)

Problem 2:

Insertion sort uses linear search to find the right place for the next item to insert. Would it be faster to find the place using binary search (reduce number of comparisons)? We still have to shift 1 item at a time from the largest till the right place.
Use binary search on the already sorted items to find the place where the new element should go and then shift the exact number of items that need to be shifted and placing the new item in its place. The algorithm works the same, except that instead comparing and shifting item by item, it will compare quickly using binary search but it will still shift one by one till the right place (without comparison).
Plot the performance of the algorithm **against the original** insertion sort.

Name a folder "**P2**" and put your files inside it (even if it's only one file).

Section 2:

Name a folder "**Section2**" and put all your section 2 problems folder inside it.

Linked lists

In this problem, you should develop a linked list class similar to that provided in the C++ STL. The public interface of your class should provide basic insertion and deletion functions. In addition, it should provide an iterator class as an inner class in order to access the data stored in the list. For example, if we have a list of three elements, and want to access the second element, we should declare an iterator and initialize it to the position of the first element, and move to the second position as shown in the code below:

```
list<int> myList;
myList.push_back(1);
myList.push_back(2);
myList.push_back(3);
list<int>::iterator it = myList.begin();
it++;

cout<< *it;
```

notice the usage of the scope operator in the declaration of the iterator, this is because the iterator class is defined as an inner class inside the list class:

```
template<class type>
class myList {
public:

class iterator {
// your code for the iterator class here

};
// your code for the list class her

};
```

Your list class should be a template class.
The list class should have the following public interface:

- `list()` – default constructor.
- `list(type value, int initial_size)` – constructs a list having 'initial_size' elements whose values are 'value'.
- `~list()` – a destructor to clear the list and leave no memory leaks.
- `int size()` – returns the current number of elements in the list.
- `void insert(type value, iterator position)`
  adds an element at position specified by the iterator. For example, if the passed iterator currently points to the second element, this element will be shifted on position, and the new value should be added at the second position.
- `iterator erase(iterator position)` – erases the element specified by the iterator and return an iterator to the next element, throws exception if position points after the last element.
- `list<type>& operator = (list<type> another_list)` – overloads the assignment operator to deep copy a list into another list and return the current list by reference.
- `iterator begin()` – returns an iterator pointing to the first element.
- `iterator end()` – returns an iterator pointing after the last element.
- You should develop an iterator class the following public interface:
  - `void operator ++ ()` – overloads the operator ++, it should advance the iterator one position towards the end of the list, throws exception if it is currently pointing after the last element.
  - `void operator -- ()` – overloads the operator --, it should move the iterator one position toward the beginning of the list, throws exception if it is currently pointing to the first element of the list.
  - `type& operator * ()` – overloads the dereference operator to return the value contained in the current node by reference to allow its modification.
  - `bool operator == (const iterator &)` – overloads the equality comparison operator, should return true if the passed operator points to the same node.
- All node pointers in the list class of the iterator class should be private and inaccessible from outside of the class.
- No memory leaks or dangling pointers.
- It is highly recommended to implement it as a double linked list, however, it is up to you.

As mentioned above, our .end() function should return an iterator pointing to a position after the last element as the STL .end() function does. This can be done easily by having a dummy node after the actual tail, this dummy node contains no data, but mark the end of the list. So, physically, our list is never empty, which will ease the implementation of insertion and remove

operations, as now we don't have to handle an "empty list" case. However, this dummy node should be disregarded when we return the size.

Problem 1: Using your list

In this problem, you must use your linked list. Given two sorted linked lists, merge them in the first one without using an extra list, and without keeping any duplicates.

Write a main function to test.
Name a folder "**P1**" and put your files inside it (even if it's only one file).

Stacks

In this problem, you should develop a stack class similar to that provided in the C++ STL. You can use arrays or your linked list class developed in the previous problem as an underlying data structure, however, you cannot use any of the C++ STL classes in this problem.

- Your stack class should be a template.
- The stack class should have the following public interface:
  - `stack()` – default constructor.
  - `stack(type value, int intial_size)` constructs a stack having 'initial_size' elements whose values are 'value'.
  - `~stack()` – a destructor to clear the stack and leave no memory leaks.
  - `type& top()` – returns the top element by reference.
  - `void pop()` – removes the top element.
  - `void push(type value)` – adds an element to the top of the stack.
  - `int size()` – returns the number of elements in the stack.

Problem 2: Using your stack

In this problem, you must use your stack. Given an input string of brackets '(' and ')', square brackets '*' and '+', curly brackets ',' and '-', and multiple line comment token '/*' and '*/', check if this string is valid or not.

A string is considered valid if and only if:

For each opened bracket, there should be a closing bracket of the same type.
Each closing bracket should close the lastly opened bracket.
Multiple line comment tokens consist of two characters, and any text between them should be ignored. However, they are treated the same way as brackets regarding the rules described above.

`([{}])(){}[]{[]}` – valid.
`({)}` – invalid.
`({/*)}]]]]]]}*/})` – valid.
`({/*[][[]]]]]})` – invalid, the comment is not closed.
`[{/*******/}]` – valid.

Write a main function to test.
Name a folder "**P2**" and put your files inside it (even if it's only one file).

Queues

In this problem, you should develop a queue class similar to that provided in the C++ STL. You can use arrays or your linked list class developed in the previous problem as an underlying data structure, however, you cannot use any of the C++ STL classes in this problem.

- Your queue class should be template.
- The queue class should have the following public interface:
  - `queue()` – default constructor.
  - `queue(type value, int intial_size)` – constructs a queue having 'initial_size' elements whose values are 'value'.
  - `~queue()` – a destructor to clear the queue and leave no memory leaks.
  - `type& front()` – returns the first element by reference.
  - `void pop()` – removes the first element.
  - `void push(type value)` – adds an element to the back of the queue.
  - `int size()` – returns the number of elements in the queue.

Problem 3 Using your queue

In this problem, you must use your queue. Reimplement part of the **stack** data structure using only one queue as an underlying data structure. Reimplement the class for 'int' data type only and with the following functions only:

`int top()` – returns the top element. [3 Points]
`void pop()` – removes the top element. [3 Points]
`void push(int value)` – adds an element to the top of the stack. [4 Points]
Write a main function to test.
Name a folder "**P3**" and put your files inside it (even if it's only one file).

Section 3:

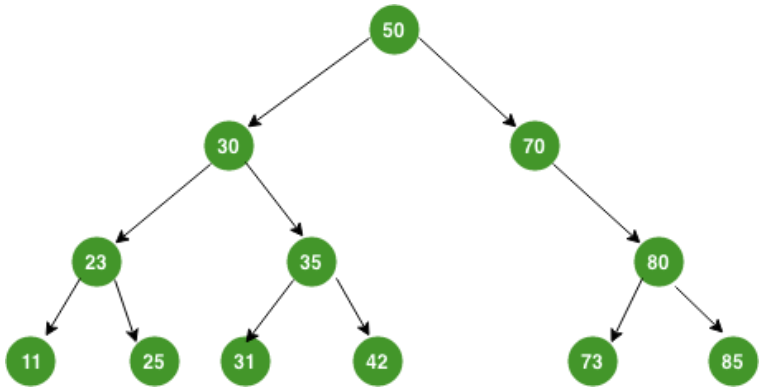
Name a folder "**Section3**" and put all your section 3 problems folder inside it.

Problem 1: BST (draw)

Apply the following operations on the **BST**
Note: The questions are independent on each other for solving any question you should do two steps

Step1: redraw the tree after applying the operation
Step2: explanation for your step

1. Delete the node with value 42
2. Delete the node with value 70
3. Delete the node with value 30



Name a folder "**P1**" and put your files inside it (even if it's only one file).

Problem 2: Implement BST AVL tree (code)

- Insert Element into the tree
- Delete Element from tree
- Display Balanced AVL Tree:
  - InOrder traversal
  - PreOrder traversal
  - PostOrder traversal

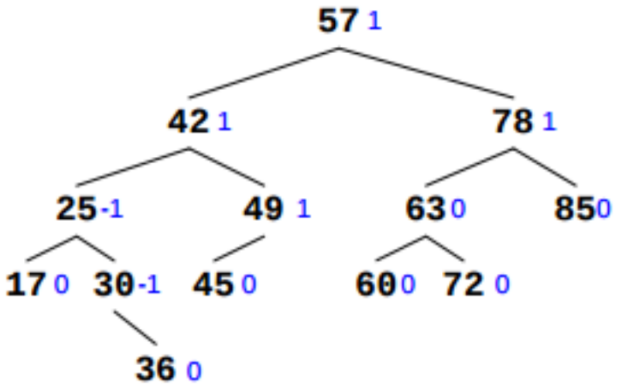
Name a folder "**P2**" and put your files inside it (even if it's only one file).

Problem 3: BST AVL tree (draw)

Explain on the BST AVL what will happen when you add two elements with value 41, 50

Name a folder "**P3**" and put your files inside it (even if it's only one file).

Balance factors shown next to each node in blue



Note: for solve this problem you should view what happen after adding first element then after adding second element and show the changes in balance factors

Rules:

1. Cheating will be punished by giving -2 * assignment mark.
2. Cheating is submitting code or reports taken from any source that you did not fully write yourself (from the net, from a book, from a colleague, etc.)
3. Giving your code to others is also considered cheating both the giver and the taker.
4. People are encouraged to help others fix their code but cannot give them their own code.

5. Do not say we solved it together and we understand it. You can write the algorithm on paper together but each group should implement it alone.
6. If you do not follow the delivery style (time and files names), your assignment will be rejected