Programming Assignment #1

1 Rules

This programming assignment attempts to help you go through the implementation of several data structures: singly linked list, doubly linked list, and red-black tree.

1. Implement the data structures yourself. The modern programming language shall already include the basic data structures. For instance, in C++, you can directly create a doubly linked list by using `std::list`. This assignment is not about using those interfaces. In the assignment, the implementation shall be written on simple data types, like primitive data types and pointers. Our target is to implement the data structures and algorithms learned from the class.
2. Write comments and use meaningful naming. Commenting on your codes and naming the variables based on their functionality is critical for software engineering as they makes the codes understandable. The rule of thumb is that when the others are reading your codes, they will need to feel comfortable and your codes tractable. We may deduct points if the codes are hard to understand.
3. Use your favorite programming language. You are allowed to use your preferred language, including but not limited to C, C++, and Python. But be sure to follow the (1).
4. Test your program. We will have several required tasks to test your data structures. Test your program yourself and include the results in the report.

2 Part 1: Implementation of data structures

In this part, you shall implement several data structures. Let's assume the data (key) to be stored is `int`. Let's also assume that the keys are unique. In other words, there will be no identical numbers. (Bonus: try making your codes generic so that they can store any type of data.)

2.1 Singly linked list

Singly linked list shall have the following operations.

1. `list_search(key)`: find the element with the key value.
2. `insert(x, key)`: insert a new node storing `key` after the node `x`.
3. `prepend(key)`: insert a new node storing `key` at the front of the list.
4. `append(key)`: insert a new node storing `key` at the back of the list.
5. `delete(x)`: delete the node `x` from the list.

2.2 Doubly linked list

The doubly linked list shall support the same set of operations as the singly linked one.

2.3 Red Black tree

The RBT shall have the following operations.

1. `find(key)`: find the node with the key value.
2. `insert(key)`: insert a new node storing `key` in the RBT.
3. `delete(key)`: find the node storing `key` and delete it from the RBT.
4. `successor(key)`: find the node with the key value and return the successor node in the RBT.

The implementation of RBT is not trivial. You are encouraged to find some reference tutorial to help you understand the procedures. But be sure to write your own program and make it well commented.

2.4 Create a "Set" abstract data type

Now you can embed the three data structures under the same interface, the "Set" abstract data type. Let's assume it takes a set of integers and shall supporting the inserted integers. We can throw an error or trigger a failure assertion if two duplicate numbers are inserted. Notice that it is not trivial for the linked lists to check if it stores duplicate value. **Let's skip the checking and assume the inserted numbers can never be duplicate.** Our Set ADT shall support the following operations.

1. `find(key)`: find the element with the key value and return the pointer to it.
2. `insert(key)`: insert a new node storing `key` in the Set.
3. `delete(key)`: find the node storing `key` and delete it from the Set.

You can link your data structures with the Set interface using approaches like `template` and `inheritance`. You can also just create separate structs/classes of `Set_SLL`, `Set_DLL` and `Set_RBT`. What is important is to see how our data structures do in the Set ADT.

3 Part 2: Test your Sets

Test each data structure you have written in the **Part 1** as follows.

3.1 Prepare the data

Let's generate the following three vectors (list in Python) of numbers.

1. -10, -9, -8, ... , 0, 1, 2, ... 1000.
2. -100, -99, -98, ..., 0, 1, 2, ... 100K.
3. -500, -499, ... , 0, 1, 2, ... 500K.
4. -1000, -999, ... , 0, 1, 2, ... 1M.

Then shuffle the numbers. You can use `std::shuffle` in C++, and `random.shuffle` in Python.

We will do the following experiments *five* times for each vectors of numbers. The repetitive experiments can help us reduce the randomness and shows us the average runtime.

In other words, you need do $3 \times 4 \times 5 = 60$ experiments (in three data structures, four testing vectors, and five repetitive experiments). Write your programs in loops so that you don't need manually adjust your experiments.

3.2 Insert the data in your data structures.

Insert the numbers one by one in your three Sets operations using `insert(key)`.

Record the total runtime in inserting the numbers.

Check the sizes of the elements stored are corrected.

.

3.3 Delete the negative numbers.

Delete all the negative numbers one by one from your Sets using `delete(key)`.

Record the total runtime in erasing the numbers.

3.4 Search 10% of the positive numbers

Now test the `find` function of your data structures.

For each data structure, try to search the first 10% of positive numbers one by one. For example, for the first round of experiments where the integers currently stored shall be 0, 1, 2, 3, ..., 1000. Try to first find *0*, then *1*, until *99*. Pick the first 100, 10K, 50K and 100K in each round of the experiments. Since the order of numbers is shuffled, this experiment gives you statistics of how fast your data structures do the `search` on average.

Record the total runtime in searching the numbers.

3.5 Plot the runtime.

Draw two plots for each of the three operations (`insert`, `delete` and `find`). In total, six plots shall be drawn. Let x-axis be the size of the input problem, y-axis be the runtime. In the first plot, draw the total runtime of the operations. In the second plot, draw the average runtime per operation (divide the total time by number of operations you actually executed). Use different color of lines and legends for different data structures (singly linked list, doubly linked list and red-black tree).

Analyzing the plots. Are they following your expectations? Why?

4 What to submit

You need to submit a `.zip` file for source codes and a `.pdf` file for a report.

4.1 Source codes:

Please wrap all your source codes into a `.zip` file and submit it to the class.

4.2 Report

The report shall include two parts.

In **Part 1**, show your source codes for the data structures implementation and briefly describe how to implement the operations.

In **Part 2**, show your source codes for the test program and report what is asked in Sec. 3.