COMP9024 19T3

Week 8 Problem Set
Self-adjusting Search Trees

Data Structures and Algorithms

1. (Insertion at root)

   a. Consider an initially empty BST and the sequence of values

   ```
   1 2 3 4 5 6
   ```

   - Show the tree resulting from inserting these values "at leaf". What is its height?
   - Show the tree resulting from inserting these values "at root". What is its height?
   - Show the tree resulting from alternating between at-leaf-insertion and at-root-insertion. What is its height?

   b. Complete this week's Binary Search Tree ADT (`BST.h`, `BST.c`) from the lecture by an implementation of the function:

   ```
   Tree insertAtRoot(Tree t, Item it) { ... }
   ```

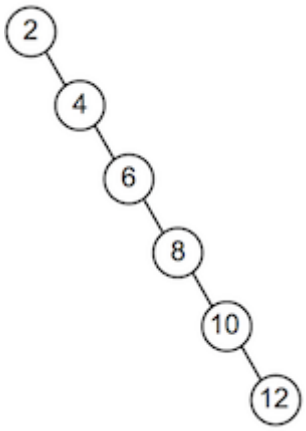
   We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program that corresponds to this exercise. It expects to find the file named* `BST.c` *in the current directory with your implementation of the function* `insertAtRoot()`.

   You can use dryrun as follows:

   ```
   prompt$ 9024 dryrun BST
   ```

2. (Rebalancing)

   Trace the execution of `rebalance(t)` on the following tree. Show the tree after each rotate operation.



3. (Splay trees)

   a. Show how a Splay tree would be constructed if the following values were inserted into an initially empty tree in the order given:

   ```
   3 8 5 7 2 4
   ```

   b. Let `t` be your answer to question a., and consider the following sequence of operations:

```
SearchSplay(t,3)
SearchSplay(t,5)
SearchSplay(t,6)
```

Show the tree after each operation.

4. (AVL trees)

   Note: You should answer the following question without the help of the `treeLab` *program from the lecture.*

   Show how an AVL tree would be constructed if the following values were inserted into an initially empty tree in the order given:

   ```
   58 26 12 37 43 40
   ```

5. (Lazy deletion)

   There are (at least) two approaches to dealing with deletions from binary search trees. The first, as used in lectures, is to remove the tree node containing the deleted value and re-arrange the pointers within the tree. The second is to not remove nodes, but simply to mark them as being "deleted".

   For this question, assume that we are going to re-implement Binary Search Trees so that they use mark-as-deleted rather than deleting any nodes. Under this scheme, no nodes are ever removed from the tree; instead, when a value is deleted, its node remains (and continues to retain the same value) but is marked so that it can be recognised as deleted.

   a. Suggest a modification to the BST data structure (week 7 lecture) to implement deleted values.

   b. Modify the search algorithm (week 7 lecture) for a "conventional" binary search tree to take into account deleted nodes.

   c. One significant advantage of deletion-by-marking is that it makes the deletion operation simpler. All that deletion needs to do is search for a node containing the value to be deleted. If it finds such a node, it simply "marks" it as deleted. If it does not find such a node, the tree is unchanged.

      Write a deletion algorithm that takes a BST `t` and a value `v` and returns a new tree which does not contain an undeleted node with value `v`.

   d. The most problematic aspect of deletion-by-marking is insertion. If handled naively, the tree grows as if it contains *n*+*d* values, where *n* is the number of nodes containing undeleted values and *d* is the number of nodes containing deleted values. If many values are deleted, then the tree becomes significantly larger than necessary.

      A more careful approach to insertion can help to limit the growth of the tree by re-using nodes containing deleted values. Modify the algorithm for AVL tree insertion from the lecture to re-use deleted nodes where possible without causing an imbalance.

6. (2-3-4 trees)

   Show how a 2-3-4 tree would be constructed if the following values were inserted into an initially empty tree in the order given:

   ```
   1 2 3 4 5 8 6 7 9 10
   ```

   Once you have built the tree, count the number of comparisons needed to search for each of the following values in the tree:

```
1  7  9  13
```

7. (Red-black trees)

   a. Show how a red-black tree would be constructed if the following values were inserted into an initially empty tree in the order given:

   ```
   1 2 3 4 5 8 6 7 9 10
   ```

   Once you have built the tree, compute the cost (#comparisons) of searching for each of the following values in the tree:

   ```
   1  7  9  13
   ```

   b. Consider the following high-level description from the lecture of an algorithm for inserting items into a red-black tree:

   ```
   insertRB(tree,item,inRight):
      if tree is empty then
         return newNode(item)
      else if item=data(tree) then
         return tree
      end if
      if left(tree) and right(tree) are RED then
         split 4-node
      end if
      recursive insert, re-arrange links/colours after insert
      return modified tree

   insertRedBlack(tree,item):
      tree=insertRB(tree,item,false)
      color(tree)=BLACK
      return tree
   ```

   Implement this algorithm in the Red-Black Tree ADT (RBTree.h, RBTree.c) from the lecture as the function:

   ```
   Tree TreeInsert(Tree t, Item it) { ... }
   ```

We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program that corresponds to this exercise. It expects to find the file named* `RBTree.c` *in the current directory with your implementation of the function* `TreeInsert()`.

You can use dryrun as follows:

```
prompt$ 9024 dryrun RBTree
```

8. **Challenge Exercise**

Extend the BST ADT from the lecture (BST.h, BST.c) by an implementation of the function

```
deleteAVL(Tree t, Item it)
```

to properly delete an element from an AVL tree (as opposed to lazy deletion, cf. Exercise 5) while maintaining balance.

Assessment

Group submissions will not be allowed. Your program must be entirely your own work. Plagiarism detection software will be used to compare all submissions pairwise.

Submit your solutions to **Exercise 1 and 7** using the following **give** command:

```
prompt$ give cs9024 week8 BST.c RBTree.c
```

Make sure you spell the filenames correctly. You can run **give** multiple times. Only your last submission will be marked.

The deadline for submission is **Monday, 11 November 11:11:11am**.

Each program is worth 1 mark. Total marks = 2. Auto-marking will be run by the lecturer several days after the submission deadline using different test cases than `dryrun` does.

Hints:

- Programs will not be manually marked.
- It is important that the output of your program follows exactly the format shown above, otherwise auto-marking will result in 0 marks.
- Ensure that your program compiles on a CSE machine with the standard options `-Wall -Werror -std=c11`. Programs that do not compile will receive 0 marks.
- `dryrun` and auto-marking also check the correct usage of dynamic memory allocation and pointers as well as the absence of memory leaks.
- Do your own testing in addition to running `dryrun`.