COMP9024 18s2

Week 10 Problem Set

Data Structures and Algorithms

Self-adjusting Trees

1. (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:

```
5 3 8 7 4
```

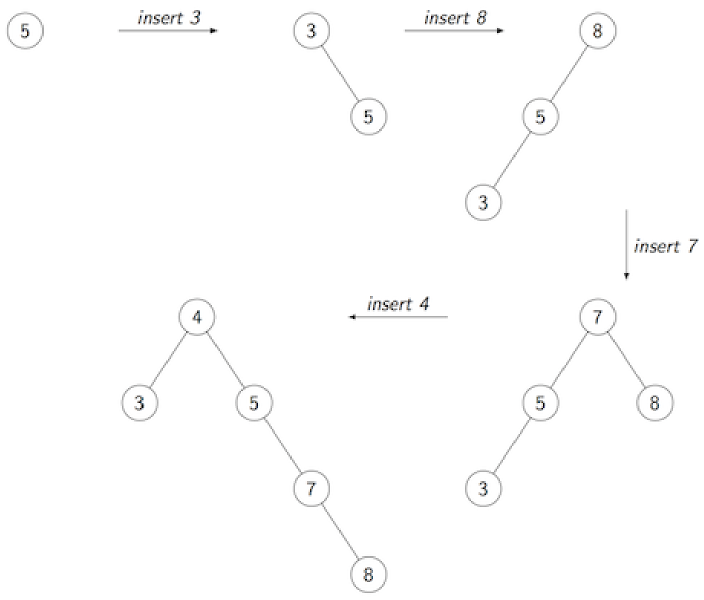
   b. Let `t` be your answer to question a., and consider the following sequence of operations:

```
SearchSplay(t,7);
SearchSplay(t,8);
SearchSplay(t,6);
```

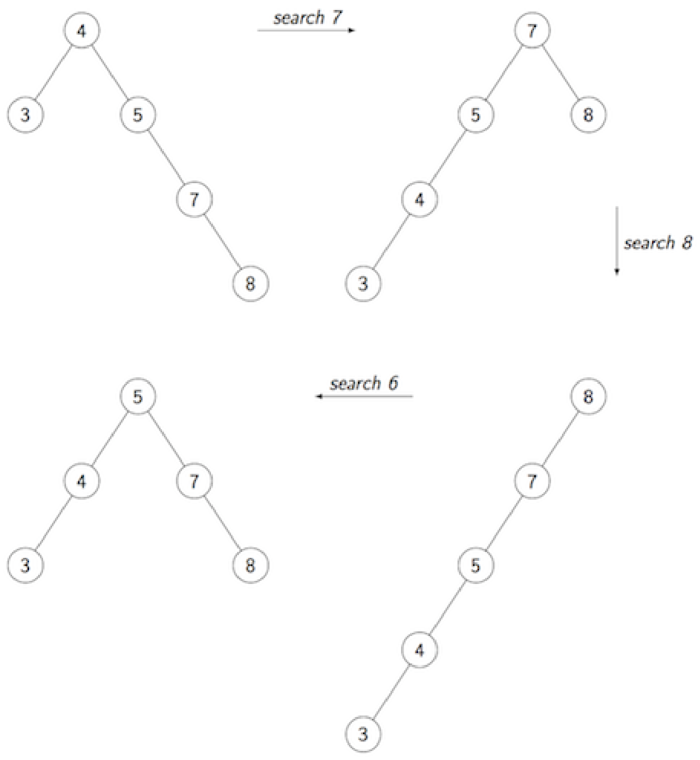
   Show the tree after each operation.

Answer:

a. The following diagram shows how the tree grows:



b. The following diagram shows how the tree changes with each search operation:



2. (AVL trees)

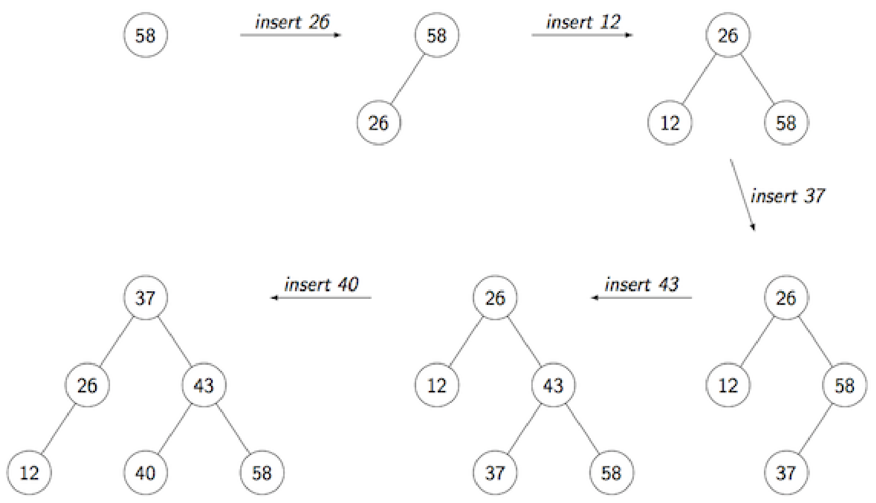
   Answer the following question without the help of the `treeLab` program from the lecture.

   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 ~~39~~ 40
```

   Answer:

   The following diagram shows how the tree grows:



   Imbalances happen:
   - when 12 is inserted, which triggers a right rotation at the root;
   - when 43 is inserted, which triggers a left rotation at 37 followed by a right rotation at 58;
   - when 40 is inserted, which triggers a right rotation at 43 followed by a left rotation at the root.

3. (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 from the lecture to implement deleted values.

   b. Modify the search algorithm from the 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.

   Answer:

   a. The simplest modification is to add another field to the node to indicate that the node has been deleted:

```
typedef struct Node {
   bool deleted;
   int  data;
   Tree left, right;
} Node;
```

      Note that it is not possible to come up with a "distinguished" value to put in the `value` field to represent "deleted", since all possible values of `value` are valid.

   b. The following algorithm ignores nodes that have been deleted:

```
TreeSearch(tree,item):
|  Input  tree, item
|  Output true if item found in tree, false otherwise
|
|  if tree is empty then
|     return false
|  else if item<data(tree) then
|     return TreeSearch(left(tree),item)
|  else if item>data(tree) then
|     return TreeSearch(right(tree),item)
|  else      // found => check if node has been deleted
|     if tree.deleted return false
|                else return true
|  end if
```

c. The following simple function implements deletion-by-marking:

```
TreeDelete(t,v):
|  Input  tree t, value v
|  Output t with v deleted
|
|  if t is not empty then      // nothing to do if tree is empty
|  |  if v<data(t) then        // delete v in left subtree
|  |     left(t)=TreeDelete(left(t),item)
|  |  else if v>data(t) then   // delete v in right subtree
|  |     right(t)=TreeDelete(right(t),item)
|  |  else                     // mark 't' as deleted
|  |     t.deleted=true
|  |  end if
|  end if
|  return t
```

d. The algorithm below implements the following strategy:

- if the value is already in the AVL tree, the tree is unchanged
- if there are no deleted nodes on the insertion path, insert as a leaf as normal and rebalance if necessary
- if a deleted node is found containing the value to be inserted, un-mark it
- if a deleted node is on the insertion path and if the value to be inserted is between the inorder predecessor and the inorder successor of the deleted node, then replace the value in that node by the new value, and un-mark it

```
min(t):
|  Input  tree t
|  Output minimum value in t
|
|  while left(t) not empty do
|     t=left(t)
| return data(t)

max(t):
|  Input  tree t
|  Output maximum value in t
|
|  while right(t) not empty do
|     t=right(t)
| return data(t)

insertAVL(t,v):
|  Input  tree t, value v
|  Output t with item AVL-inserted
|
|  if t is empty then
|     return new node containing v
|  else if v=data(t) then
|     return t
|  else if t.deleted and max(left(t))<v<min(right(t)) then
|     data(t)=v
|     t.deleted=false
|  else
|  |  if v<data(t) then
|  |     left(t)=insertAVL(left(t),v)
|  |  else if v>data(t) then
|  |     right(t)=insertAVL(right(t),v)
|  |  end if
|  |  if height(left(t))-height(right(t)) > 1 then
|  |     if v>data(left(t)) then
|  |        left(t)=rotateLeft(left(t))
|  |     end if
|  |     t=rotateRight(t)
|  |  else if height(right(t))-height(left(t)) > 1 then
|  |     if v<data(right(t)) then
|  |        right(t)=rotateRight(right(t))
|  |     end if
|  |     t=rotateLeft(t)
|  |  end if
|  |  return t
|  end if
```

4. (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:

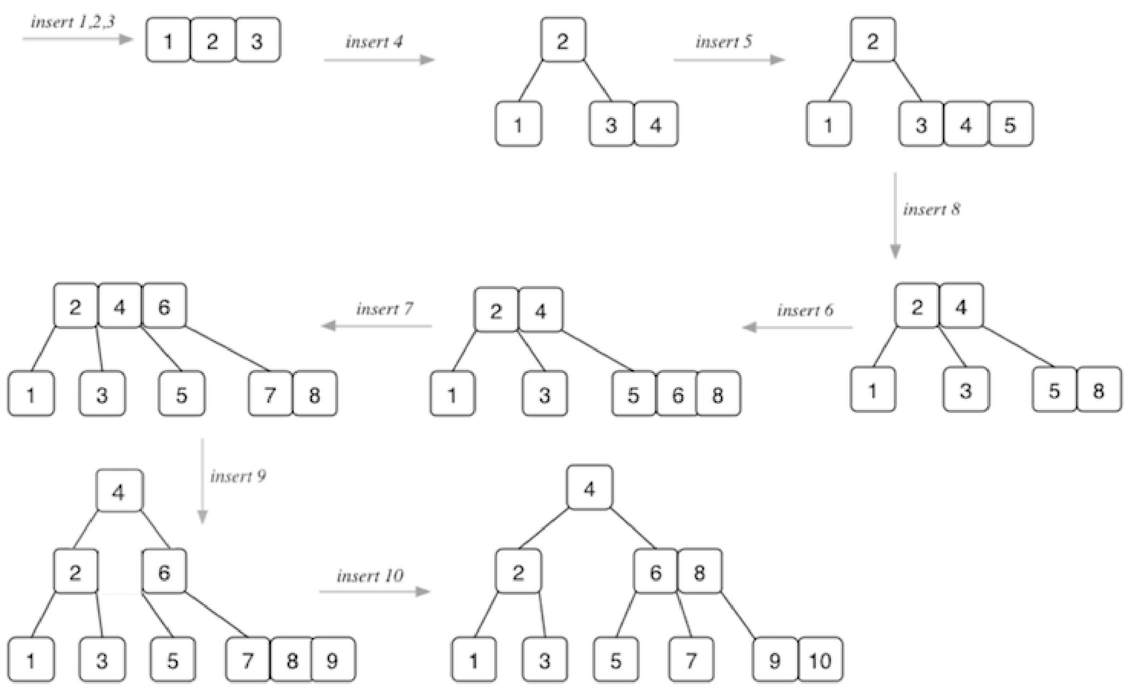
   ```
   1 2 3 4 5 8 6 7 9 10
   ```

   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
   ```

   Answer:

   The following diagram shows how the tree grows:



   Search costs (for tree after insertion of 10):

   - search(1): cmp(4),cmp(2),cmp(1) ⇒ cost = 3
   - search(7): cmp(4),cmp(6),cmp(8),cmp(7) ⇒ cost = 4
   - search(9): cmp(4),cmp(6),cmp(8),cmp(9) ⇒ cost = 4
   - search(13): cmp(4),cmp(6),cmp(8),cmp(9),cmp(10) ⇒ cost = 5

5. (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)
      end if
      if left(tree) and right(tree) are RED then
         split 4-node
      end if
      recursive insert cases
   ```

```
   re-arrange links/colours after insert
   return modified tree

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

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

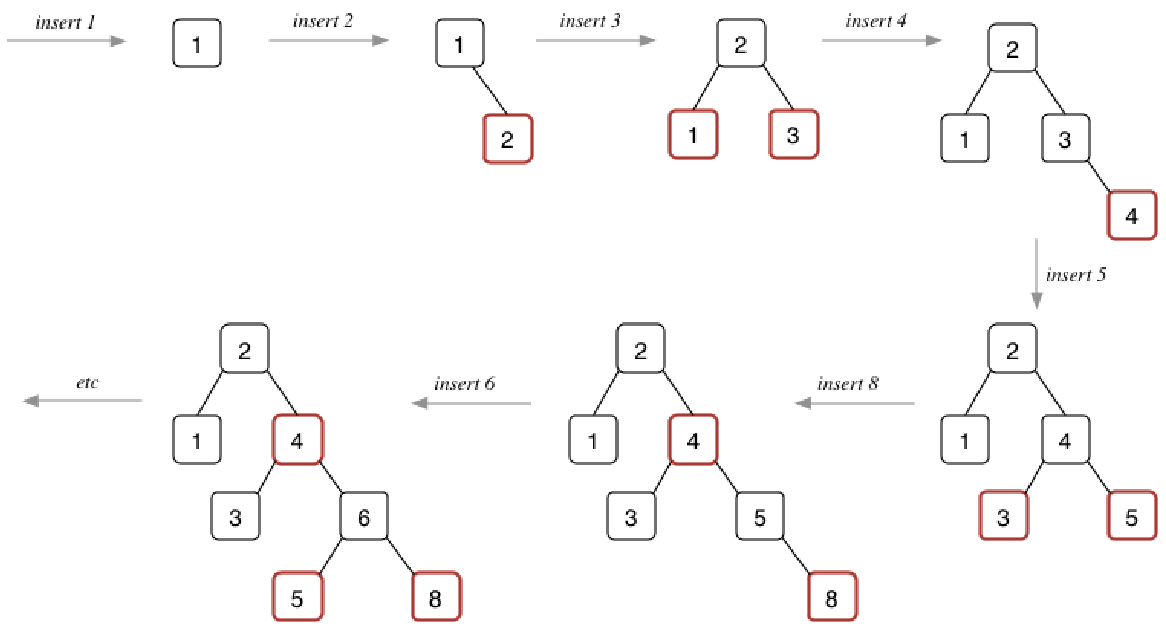
```
Tree TreeInsert(Tree t, Item it) { ... }
```

We have created a script that can automatically test your program. To run this test you can execute the dryrun *program that corresponds to the problem set and week. It expects to find a file named* RBTree.c *in the current directory. You can use dryrun as follows:*

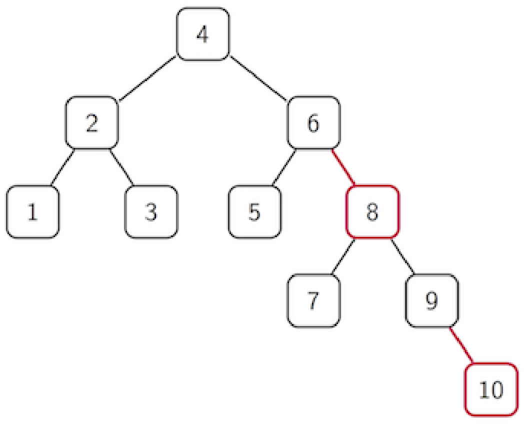
```
prompt$ ~cs9024/bin/dryrun prob10
```

Answer:

a. The following diagrams shows how the tree grows:



Search costs: The number of comparisons needed to search for a node is 1 + the level of the node in the final red-black tree:



- search(1): cost = 3
- search(7): cost = 4
- search(9): cost = 4
- search(13): cost = 5

Note that the resulting red-black tree corresponds to the 2-3-4 tree in exercise 4, with identical search costs.

b.
```
Tree insertRB(Tree t, Item it, bool inRight) {
   if (t == NULL)
      return newNode(it);
   if (it == data(t))
      return t;
   if ( isRed(left(t)) && isRed(right(t)) ) {
      colour(t) = RED;
      colour(left(t)) = BLACK;
      colour(right(t)) = BLACK;
   }
   if (it < data(t)) {
      left(t) = insertRB(left(t), it, false);
      if (isRed(t) && isRed(left(t)) && inRight) {
         t = rotateRight(t);
      }
      if ( isRed(left(t)) && isRed(left(left(t))) ) {
         t = rotateRight(t);
         colour(t) = BLACK;
         colour(right(t)) = RED;
      }
   } else {
      right(t) = insertRB(right(t), it, true);
      if ( isRed(t) && isRed(right(t)) && !inRight ) {
         t = rotateLeft(t);
      }
      if ( isRed(right(t)) && isRed(right(right(t))) ) {
         t = rotateLeft(t);
         colour(t) = BLACK;
         colour(left(t)) = RED;
      }
   }
   return t;
}

Tree TreeInsert(Tree t, Item it) {
   t = insertRB(t, it, false);
   colour(t) = BLACK;
   return t;
}
```

6. **Challenge Exercise**

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

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

   to properly delete an element from an AVL tree (unlike lazy deletion from Exercise 3) while maintaining balance.

   Answer:

   The following algorithms implements standard BST deletion and then repairs any imbalanes in the same ways as AVL insertion does.

```
Tree AVLrepair(Tree t) {
   int hL  = TreeHeight(left(t));
   int hLL = TreeHeight(left(left(t)));
   int hLR = TreeHeight(right(left(t)));
   int hR  = TreeHeight(right(t));
   int hRL = TreeHeight(left(right(t)));
   int hRR = TreeHeight(right(right(t)));
   if (hL-hR > 1) {
      if (hLR-hLL > 1)
         left(t)=rotateLeft(left(t));      // left-right case
      t = rotateRight(t);
   } else if (hR-hL > 1) {
      if (hRL-hRR > 1)
         right(t)=rotateRight(right(t));   // right-left case
      t = rotateLeft(t);
   }
   return t;
}

Tree deleteAVL(Tree t, Item it) {
   if (t != NULL) {
      if (it < data(t)) {
         left(t) = TreeDelete(left(t), it);
         t = AVLrepair(t);
      } else if (it > data(t)) {
         right(t) = TreeDelete(right(t), it);
         t = AVLrepair(t);
      } else {
         Tree new;
         if (left(t) == NULL && right(t) == NULL)
            new = NULL;
         else if (left(t) == NULL)    // if only right subtree, make it the new root
            new = right(t);
         else if (right(t) == NULL)    // if only left subtree, make it the new root
            new = left(t);
         else {                        // left(t) != NULL and right(t) != NULL
            new = joinTrees(left(t), right(t));
            t = AVLrepair(new);
         }
         free(t);
         t = new;
      }
   }
   return t;
}
```