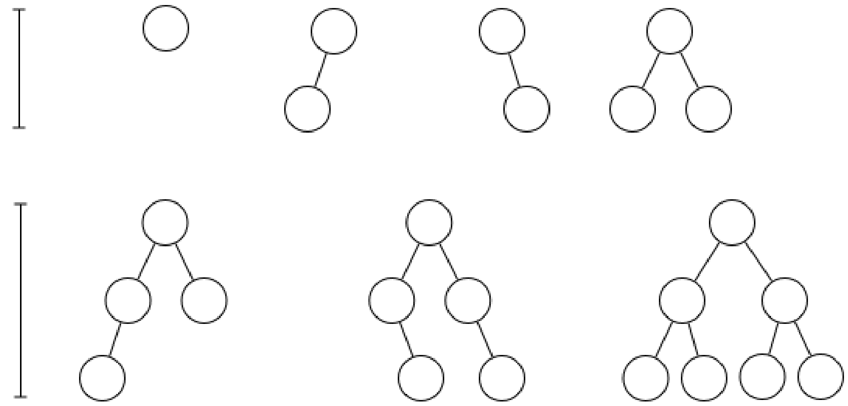
Week 09 Problem Set

Binary Search Trees, Rebalancing

1. (Tree properties)

   a. Derive a formula for the minimum height of a binary search tree (BST) containing *n* nodes. Recall that the height is defined as the number of edges on a longest path from the root to a leaf. You might find it useful to start by considering the characteristics of a tree which has minimum height. The following diagram may help:



   b. In the Binary Search Tree ADT (`BSTree.h`, `BSTree.c`) from the lecture, implement the function:

```
int TreeHeight(Tree t) { ... }
```

   to compute the height of a tree.

Answer:

   a. A minimum height tree must be balanced. In a balanced tree, the height of the two subtrees differs by at most one. In a *perfectly* balanced tree, all leaves are at the same level. The single-node tree, and the two trees on the right in the diagram above are perfectly balanced trees. A perfectly balanced tree of height h has $n = 2^0 + 2^1 + \ldots + 2^h = 2^{h+1} - 1$ nodes. A perfectly balanced tree, therefore, satisfies $h = \log_2(n+1) - 1$.

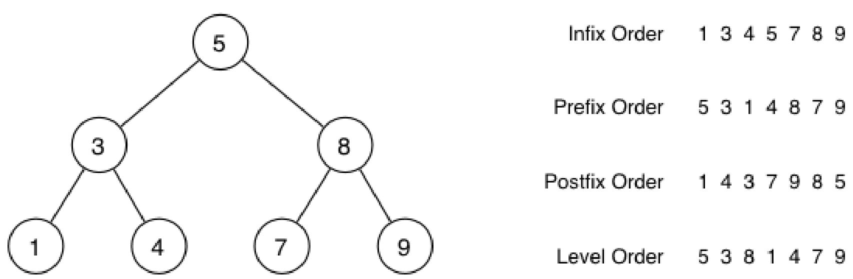
   By inspection of the trees that are not perfectly balanced above, it is clear that as soon as an extra node is added to a perfectly balanced tree, the height will increase by 1. To maintain this height, all subsequent nodes must be added at the same level. The height will thus remain constant until we reach a new perfectly balanced state. It follows that for a tree with *n* nodes, the minimum height is $h = \lceil \log_2(n+1) \rceil - 1$.

   b. The following code uses the obvious recursive strategy: an empty tree is defined to be of height -1; a tree with a root node has height one plus the height of the highest subtree.

```
int TreeHeight(Tree t) {
   if (t == NULL) {
      return -1;
   } else {
      int lheight = 1 + TreeHeight(left(t));
      int rheight = 1 + TreeHeight(right(t));
      if (lheight > rheight)
         return lheight;
      else
         return rheight;
   }
}
```

2. (Tree traversal)

Consider the following tree and its nodes displayed in different output orderings:



   a. What kind of trees have the property that their infix output is the same as their prefix output? Are there any kinds of trees for which all four output orders will be the same?

   b. Design a recursive algorithm for prefix-, infix-, and postfix-order traversal of a binary search tree. Use pseudocode, and define a single function `TreeTraversal(tree,style)`, where `style` can be any of `"NLR"`, `"LNR"` or `"LRN"`.

Answer:

a. One obvious class of trees with this property is "right-deep" trees. Such trees have no left sub-trees on any node, e.g. ones that are built by inserting keys in ascending order. Essentially, they are linked-lists.

   Empty trees and trees with just one node have all output orders the same.

b. A generic traversal algorithm:

```
TreeTraversal(tree,style):
|  Input tree, style of traversal
|
|  if tree is not empty then
|  |  if style="NLR" then
|  |     visit(data(tree))
|  |  end if
|  |  TreeTraversal(left(tree),style)
|  |  if style="LNR" then
|  |     visit(data(tree))
|  |  end if
|  |  TreeTraversal(right(tree),style)
|  |  if style="LRN" then
|  |     visit(data(tree))
|  |  end if
|  end if
```

3. (Insertion and deletion)

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

a. Show the BST that results from inserting the following values into an empty tree in the order given:

```
6 2 4 10 12 8 1
```

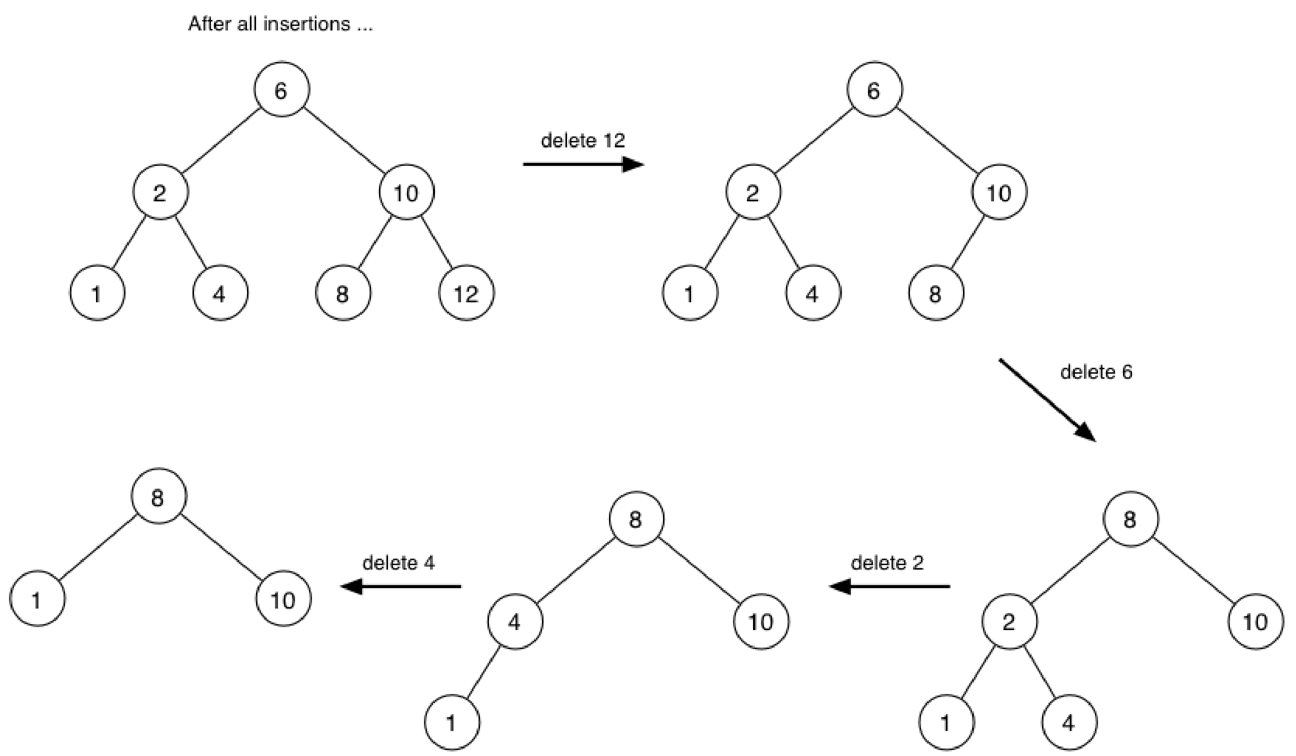
   Assume "at leaf" insertion.

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

```
TreeDelete(t,12);
TreeDelete(t,6);
TreeDelete(t,2);
TreeDelete(t,4);
```

   Assume that deletion is handled by joining the two subtrees of the deleted node if it has two child nodes. Show the tree after each delete operation.

Answer:



4. (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 the Binary Search Tree ADT (`BSTree.h`, `BSTree.c`) from the lecture by an implementation of the function:

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

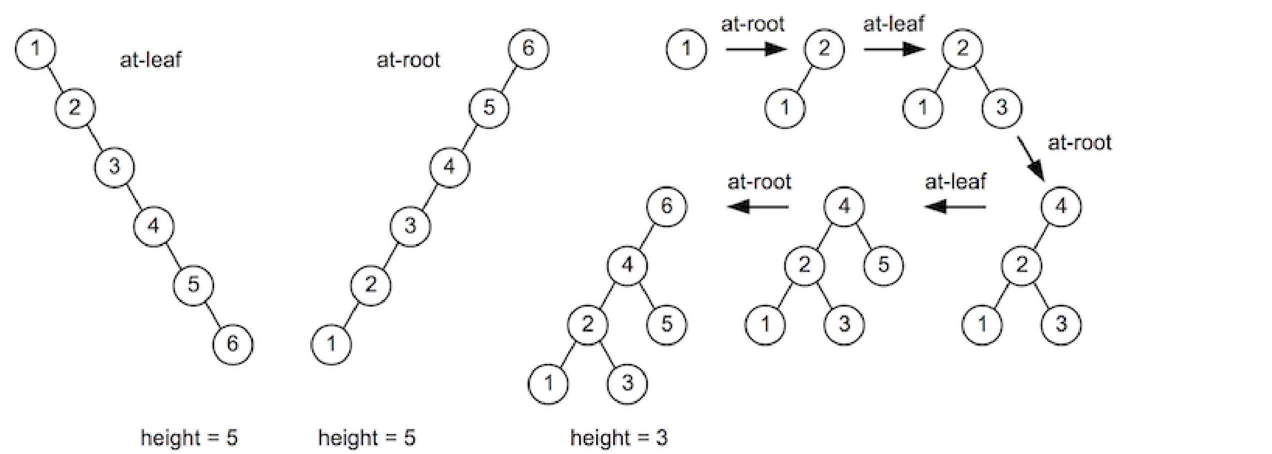
We have created a script that can automatically test your program. To run this test you can execute the `dryrun` *program that corresponds to the problem set and week. It expects to find a file named* `BSTree.c` *in the current directory. You can use dryrun as follows:*

```
prompt$ ~cs9024/bin/dryrun prob09
```

Note: The `dryrun` *script expects your program to include your implementation for Exercise 1.b.*

Answer:

a. At-leaf-insertion results in a "right-deep" tree while at-root insertion results in a "left-deep" tree. Both are fully degenerate trees of height 5. Alternating between the two styles of insertion results in a tree of height 3. Generally, if *n* ordered values are inserted into a BST in this way, then the resulting tree will be of height $\left\lfloor \frac{n}{2} \right\rfloor$.

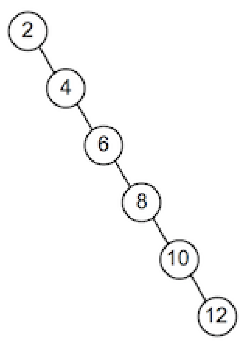


b.

```
Tree insertAtRoot(Tree t, Item it) {
   if (t == NULL) {
      t = newNode(it);
   } else if (it < data(t)) {
      left(t) = insertAtRoot(left(t), it);
      t = rotateRight(t);
   } else if (it > data(t)) {
      right(t) = insertAtRoot(right(t), it);
      t = rotateLeft(t);
   }
   return t;
}
```

5. (Rebalancing)

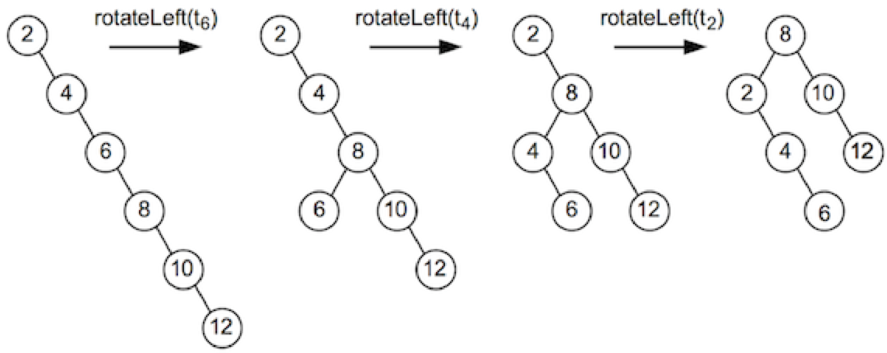
Trace the execution of `rebalance(t)` on the following tree. Show the tree after each rotate operation.



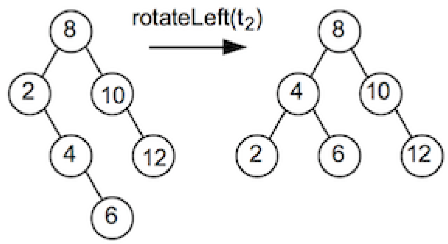
Answer:

In the answer below, any (sub-)tree $\texttt{t}_\texttt{n}$ is identified by its root node $\texttt{n}$, e.g. $\texttt{t}_2$ for the original tree.

Rebalancing begins by calling $\texttt{partition(t}_2\texttt{,3)}$ since the original tree has 6 nodes. The call to $\texttt{partition(t}_2\texttt{,3)}$ leads to a series of recursive calls: $\texttt{partition(t}_4\texttt{,2)}$, which calls $\texttt{partition(t}_6\texttt{,1)}$, which in turn calls $\texttt{partition(t}_8\texttt{,0)}$. The last call simply returns $\texttt{t}_8$, and then the following rotations are performed to complete each recursive call:



Next, the new left subtree `t`$_2$ gets balanced via `partition(t`$_2$`,1)`, since this subtree has 3 nodes. Calling `partition(t`$_2$`,1)` leads to the recursive call `partition(t`$_4$`,0)`. The latter returns `t`$_4$, and then the following rotation is performed to complete the rebalancing of subtree `t`$_2$:



The left and right subtrees of `t`$_4$ have fewer than 3 nodes, hence will not be rebalanced further. Rebalancing continues with the right subtree `t`$_{10}$. Since this tree also has fewer than 3 nodes, rebalancing is finished.

6. **Challenge Exercise**

The function `showTree()` from the lecture displays a given BST sideways. A more attractive output would be to print a tree properly from the root down to the leaves. Design and implement a new function `showTree(Tree t)` for the Binary Search Tree ADT (`BSTree.h`, `BSTree.c`) to achieve this.

Please email me your solution. The best solution will be added to our BST ADT implementation and used in the next lecture (week 10). Both the attractiveness of the visualisation and the simplicity of the code will be judged.

Answer:

Solution courtesy of Daniel Hocking:

```
void showTreeU(Tree t, int depth, int target, int height) {
   int i;
   if (t != NULL) {
      // When target depth has been found, print the value with spaces
      if (depth == target) {
         // The number of spaces is related to the depth, deeper means less spaces as there are more nodes
         int spaces = (int)pow((double)2, (double)(height - depth));
         for (i = 0; i < spaces; i++)
            printf("  ");
         printf("%d", data(t));
         for (i = 0; i < spaces; i++)
            printf("  ");
         return;
      }
      // Recurse until correct depth found or tree is null
      showTreeU(left(t), depth + 1, target, height);
      showTreeU(right(t), depth + 1, target, height);
   } else {
      // Adds extra spaces when the tree is unbalanced
      int spaces = (int)pow((double)2, (double)(height - depth)) * 2;
      for (i = 0; i < spaces; i++)
         printf("  ");
   }
}

void showTree(Tree t) {
   // Find the height of the tree first, required for spacing
   int height = TreeHeight(t);
   int target = 0;
   // Print the tree out one row at a time, target is the row to print this time
   while (target <= height) {
      // Call recursive function
      showTreeU(t, 0, target, height);
      // Add some spacing between rows
      printf("\n\n");
      target++;
   }
}
```

Special mention to Jian Gao for his solution that uses a queue.