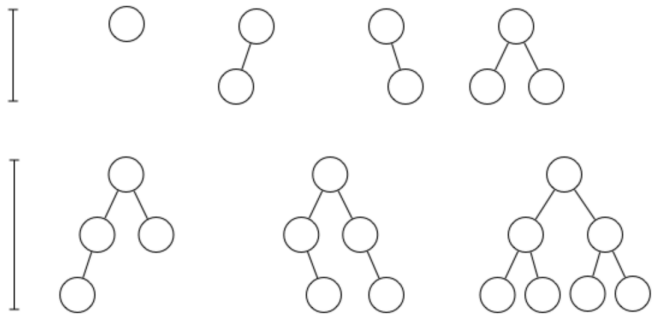
Week 7 Problem Set
Binary Search Trees

1. (Tree properties)

   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:



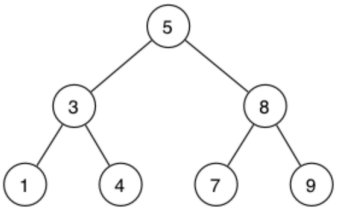
Answer:

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 $n = 2^0 + 2^1 + ... + 2^h = 2^{h+1} - 1$ 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 $h = \lceil \log_2(n+1) \rceil - 1$.

2. (Tree traversal)

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



Infix Order	1 3 4 5 7 8 9
Prefix Order	5 3 1 4 8 7 9
Postfix Order	1 4 3 7 9 8 5
Level Order	5 3 8 1 4 7 9

   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:

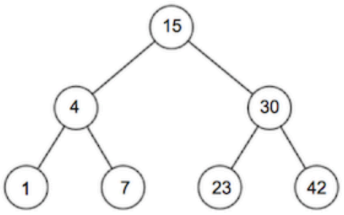
```
TreeTraversal(tree,style):
|  Input tree, style of traversal
|
|  if tree is not empty then
|  |  if style="NLR" then
|  |     visit(data(tree))
|  |  end if
|  |  TreeTraversal(left(tree),style)
|  |  if style="LNR" then
|  |     visit(data(tree))
|  |  end if
|  |  TreeTraversal(right(tree),style)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

3. (Standard insertion)

   Show the BST that results from inserting (at leaf) the following values into an empty tree in the order given:

```
15 4 7 30 42 23 1
```

Answer:



4. (Insertion at root)

Consider an initially empty BST and the sequence of values

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

a. Show the tree resulting from inserting these values "at leaf". What is its height?

b. Show the tree resulting from inserting these values "at root". What is its height?

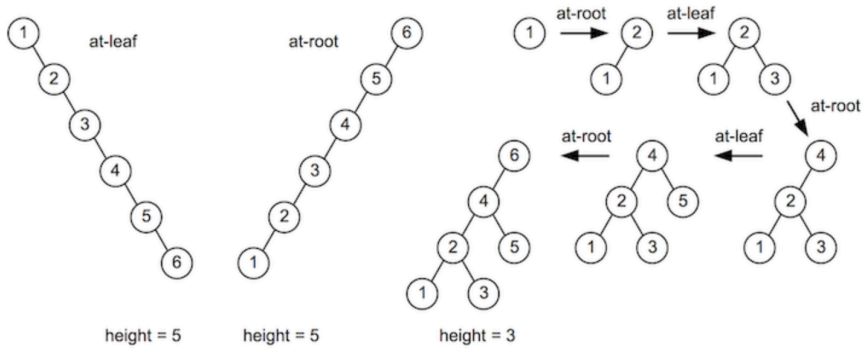
c. Show the tree resulting from alternating between at-leaf-insertion and at-root-insertion. What is its height?

Answer:

a. At-leaf-insertion results in a fully degenerate, "right-deep" of height 5.

b. At-root insertion results in a fully degenerate, "left-deep" tree of height 5.

c. 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 $\lfloor \frac{n}{2} \rfloor$.



5. **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 8). Both the attractiveness of the visualisation and the simplicity of the code will be judged.

Answer:

Solution courtesy of student Anderson Lin:

```
void printSpace(int v) {
   int sum = 0;
   for (int j = 0; j < v; j++) {
      sum += pow(2, j);
   }
   for (int i = 0; i < sum; i++) {
      printf(" ");
   }
}

void showTree(Tree t) {
   int h = TreeHeight(t);
   queue q = newQueue();
   QueueEnqueue(q, t);
   for (int i = 0; i <= h; i++) {     // in every line
      printSpace(h - i);              // print space in front of all the element in the same line

      for (int j = 0; j < pow(2, i); j++) {
         t = QueueDequeue(q);
         if (t != NULL) {
            if (j) {
               printSpace(h - i + 1); // print space between elements
            }
            printf("%d", t->data);
            QueueEnqueue(q, t->left);
            QueueEnqueue(q, t->right);
         } else {                    // if there is no subtree,
            if (j) {                 // we still pretend to have a full balanced binary subtree
               printSpace(h - i + 1);
             }
             printf(" ");            // we print a space to take the place of the missing subtree
             QueueEnqueue(q, NULL);
             QueueEnqueue(q, NULL);
         }
      }
      printf("\n");
   }
}
```

This function uses a standard queue where elements are pointers (`queue.h`, `queue.c`).