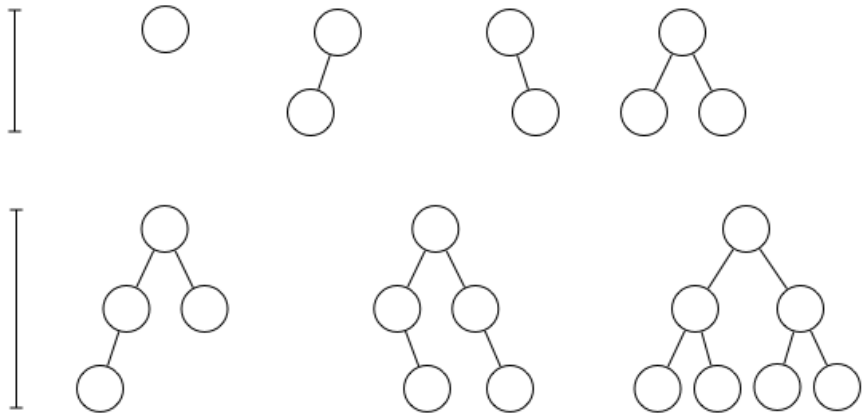
COMP9024 20T0 Data Structures and Algorithms

Week 3b Problem Set
Binary Search Trees

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:

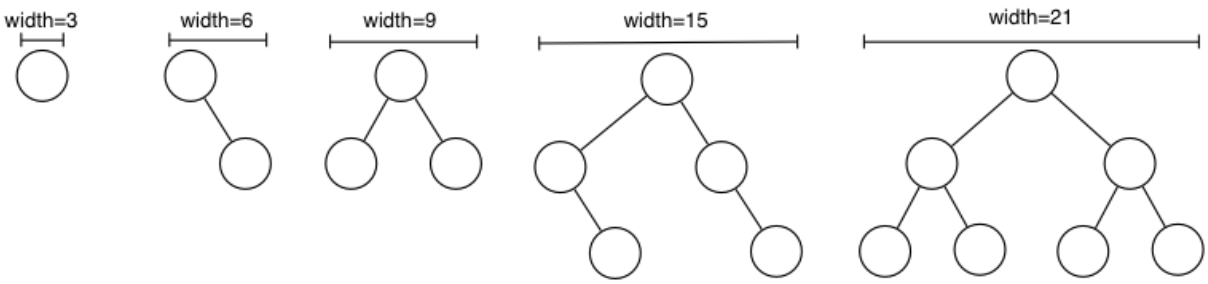


   b. In the Binary Search Tree ADT (`BSTree.h`, `BSTree.c`) from the lecture, implement the function:

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

   to compute the height of a tree.

   c. Computing the height/depth of trees is useful for estimating their search efficiency. For *drawing* trees, we're more interested in their *width*. For some simple trees the following diagrams show a useful definition of tree width if you want to keep reasonable spacing:



      i. Derive a formula for the width of a tree that generalises from the examples.

      ii. Add a new function to the `BSTree` ADT which computes the width of a tree. Use the following function interface:

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

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 program named* `BSTree.c` *with your implementation for* `TreeHeight()` *and* `TreeWidth()` *in the current directory. You can use dryrun as follows:*

```
prompt$ 9024 dryrun BSTree
```

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 $n = 2^0 + 2^1 + ... + 2^h = 2^{h+1} - 1$ 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;
   }
}
```

c. The following recursive definition of *tree width* generalises from the examples:

- An empty tree has width zero.
- A tree with just one node has width three.
- All other trees have width which is three more than the combined width of the subtrees.

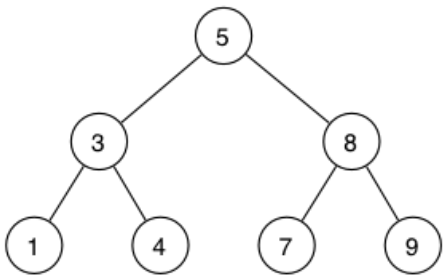
The following function computes the width:

```
#define TREEWIDTH_BASE 3

int TreeWidth(Tree t) {
   if (t == NULL)
      return 0;
   else if (left(t) == NULL && right(t) == NULL)
      return TREEWIDTH_BASE;
   else
      return TREEWIDTH_BASE + TreeWidth(left(t)) + TreeWidth(right(t));
}
```

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. (Insertion and deletion)

   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
   ```

   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: