6. Binary Trees

Problem 6.1

1. Show the result of method *traverse* for the binary tree in Figure 6.1 where we simply print the letters in all the nodes. Write the result for all three approaches: Inorder, Preorder and Postorder.

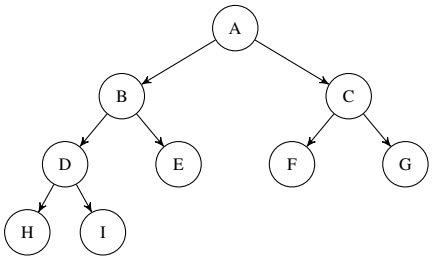


Figure 6.1: Binary Tree.

2. Implement all three methods (Inorder, Preorder, Postorder) for traversing the binary tree using recursion. The traverse method has the following signature:

```
void traverse(NodeProcessor proc, Order order);
```

where `NodeProcessor` is the interface:

```
public interface NodeProcessor<T> {
    boolean process(T data);
}
```

The `traverse` method visits the nodes of the tree according to the specified order and calls the method `process` on the nodes' data. The traversal terminates when all the nodes in the tree have been visited, or when `process` return false. The order is specified by a variable of type `Order`, which is an enumeration of tree traversal orders:

```
public enum Order {
    Preorder,
    Inorder,
    Postorder
}
```

Problem 6.2

1. As a user of the ADT Binary Tree, write the instructions necessary to transform the tree shown in Figure 6.2 into the one shown in Figure 6.3 (let the tree be called `bt`, a variable of type `BT<String>`).

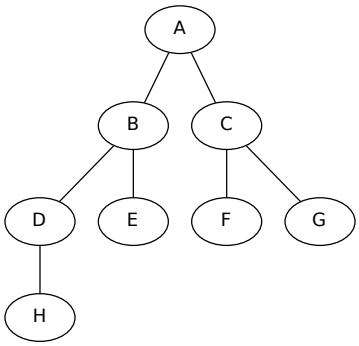


Figure 6.2: A binary tree (*H* is the left child of *D*).

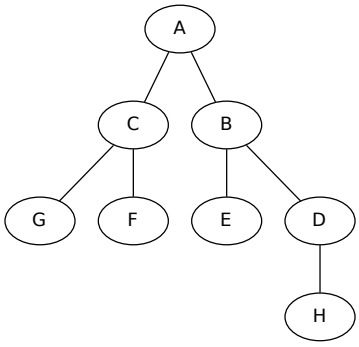


Figure 6.3: The mirror of the tree shown in Figure 6.2 (*H* is the right child of *D*).

2. Draw the tree shown in Figure 6.2 after calling the method `func` written below.

```
public class LinkedBT<T> implements BT<T>{
    ...

    public void func(){

        recFunc(root,true);
    }

    private void recFunc(BTNode<T> t, boolean flag){

        if(t==null)
            return;

        recFunc(t.left, flag);
```

```
        recFunc(t.right, !flag); // Notice the !

        if(flag){
            BTNode<T> tmp= t.left;
            t.left= t.right;
            t.right= tmp;
        }
    }
}
```

Problem 6.3

A `perfect` binary tree is a binary tree where all leaf nodes are at the same level. A `full` binary tree is a binary tree where all leaf nodes are at the same depth.

1. Given the height h of a perfect full binary tree, How can we know the number of leaf nodes l? Assume that the height of a single root node is 0.
2. How can we know the total number of nodes n in a perfect full binary tree if we knew the height is h?
3. If we know the total number of nodes in a perfect full binary tree is n, how can we know the number of non-leaf nodes?

Problem 6.4

1. Write the **iterative** method `collectInOrder`, member of the class `BT` (binary tree) that returns a list that contains all the data in the tree in the `InOrder` order.
   The method signature is: `public List<T> collectInOrder()`.

   ■ **Example 6.1** For the tree shown in Figure 6.3, the output to `collectInOrder()` is the list: $G \to C \to F \to A \to E \to B \to H \to D$. ■
2. Write the **recursive** method `mirror`, member of the class BT, that transforms the tree into its mirror (see Figure 6.3 for an example). The signature of the method is: `public void mirror()`. This method calls the private recursive method `recMirror`.
3. Write the **recursive** method `find`, a private member method of the class `BT` (binary tree) that takes as input a node t and data e and returns true if e exists in the subtree rooted at t, false otherwise. The method signature is: `private boolean find(BTNode<T> t, T e)`.

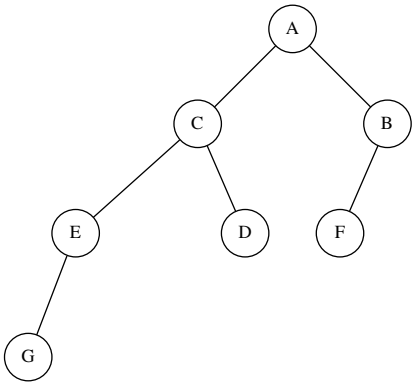


Figure 6.4: A binary tree.

■ **Example 6.2** In the tree shown in Figure 6.4, the call to `find("E")` with the node

containing data *C* as parameter returns true, whereas the call with the node containing data *B* as parameter returns false. ■

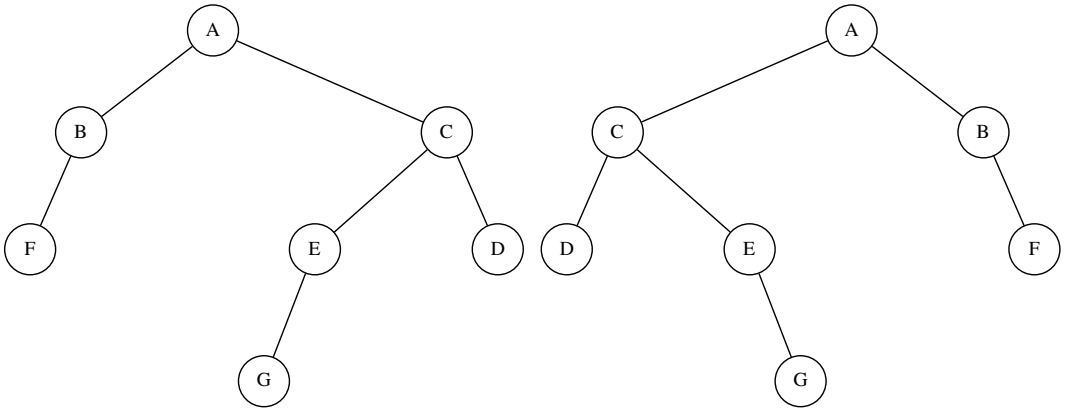
4. Write a **recursive** method `sizeBalanced`, member of the class `BT` (Binary Tree), that returns true if the tree is empty, or, at every node, the absolute value of the difference between the number of nodes in the two subtrees of the node is at most 1. The method signature is: `public int sizeBalanced()` (this method calls the private recursive method `recSizeBalanced`).

   ■ **Example 6.3** The binary tree shown shown in Figure 6.4 is not size balanced. The size balance at `E` is -1, at `C` is -1, at `B` is -1, but at `A`, the size balance is 2-4=-2. ■

Problem 6.5

1. Write the **recursive** method `isMirror`, member of the class `LinkedBT` (Binary Tree), that takes as input a binary tree and returns true if the two trees are the mirror image of each other. The method signature is `public boolean isMirror(BT<T> bt)` (this method must call the private recursive method `recIsMirror`). **Important**: Non-recursive solutions are not accepted.

   ■ **Example 6.4** The two trees shown below are mirror images of each other.



   ■

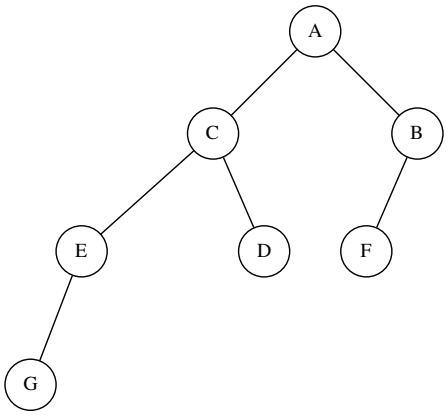
2. Write the method `boolean equal(BTNode<T> t1, BTNode<T> t2)`, member of `LinkedBT`, which returns true if the sub-trees `t1` and `t2` are equal (have the same data).
3. Write the **recursive** method `twoChildren`, member of the class BT (Binary Tree), which returns the number of nodes with two children. **Do not use any auxiliary data structures and do not call any `BT` methods**. The method signature is `public int twoChildren()`. This method must call the private recursive method `recTwoChildren`. **Important**: Non-recursive solutions are not accepted.

Problem 6.6

1. Write the **recursive** method `atLevel(int l)`, member of the class BT (Binary Tree), which returns the number of nodes at level l. We consider the **root** of the tree to be at **level 1**. Assume also that $l \geq 1$. The method signature is `public int atLevel(int l)`. This method must call the private recursive method `recAtLevel`. Do not use any auxiliary data

structures and do not call any `BT` methods.

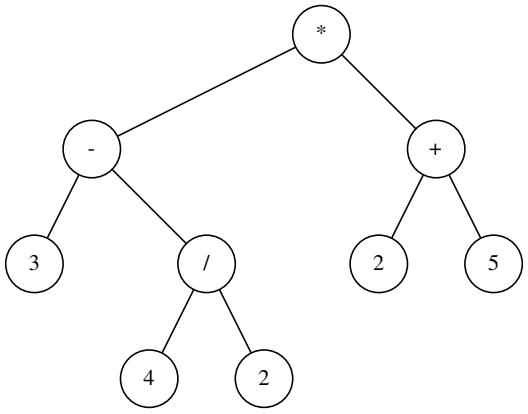
■ **Example 6.5** The call `atLevel(2)` on the binary tree shown below returns 2, `atLevel(3)` returns 2, `atLevel(1)` returns 1, `atLevel(6)` returns 0. ■



2. Write the method `nt l` **public void** `cut` member of `LinkedBT` that cuts all nodes at a depth larger than `l` (root is at depth 0).

Problem 6.7

An arithmetic expression can be represented as a binary tree as shown in the figure below.



Write the **recursive** method **public double** `eval(BT<Token> expr)` that evaluates the expression *expr*. The class `Token` represents a token of the expression and is described as follows.

```
public enum TokenType {
    Operand,
    Operation
}
```

```
public class Token {
    public TokenType getType(){} // Returns the type of the token
    public double getVal(){} // Returns the value of the token if it
        is of type Operand
    public double apply(double op1, double op2){} // If the token is
        of type Operation, this method applies it to the operands
        passed as parameters and returns the result of the operation.
}
```

Assume that the tree *expr* is not empty.

Problem 6.8

A binary tree can be implemented using an array implementation as follows. The root of the tree is stored at position 1 of the array (position 0 is left unused), and for each node stored at position i, its left child is stored at position $2i$, and its right child is stored at position $2i+1$. Implement the interface `BT` using this array representation by writing the class `ArrayBT`.

Problem 6.9

1. Write the method `private BTNode<T> build(T e, BTNode<T> l, BTNode<T> r)` which return a sub-tree with `e` as the root, and `l` and `r` as the left and right sub-trees of the root.
2. Consider the class `LinkedBT` where the data is comparable (`T extends Comparable<T>`). Consider a sub-tree `t` that satisfies the following condition: at each node, the elements in its left sub-tree are smaller than the data at the node, and all the elements in the right sub-tree are larger than the data at the node (this is basically the BST condition). Use the method `build` above to write the method `private BTNode<T> insert(T e, BTNode<T> t)` that inserts `e` into `t` and returns the new sub-tree while conserving the order of the data.

Problem 6.10

Write the method `public boolean isPathTree()`, member of the BT class, which returns true if the BT is a path tree, and false otherwise. A BT is a path tree if it does not have any node that has two children.

Problem 6.11

1. Write the method `public boolean isFullTree()`, member of the BT class, which returns true if the BT is a full tree, and false otherwise. A BT is a full tree if every node other than the leaves has two children.
2. Write the method `public boolean isCompleteTree()`, member of the BT class, which returns true if the BT is a complete tree, and false otherwise. A BT is complete if all levels are full, except may be the last one which must be full to the left.

Problem 6.12

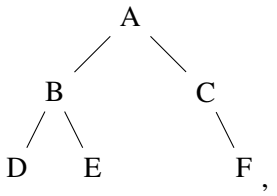
1. Write the method `private int f(BTNode<T> t, T e, int k)`, member of `BT`, which returns the number of nodes in the level `k` of the subtree `t` having data equal to `e`. The root of the subtree (that is `t`) is at level 0.
2. Repeat the question as user.

Problem 6.13

We can represent the path from the root to any node in a non empty binary tree using a string containing the letters 'L' and 'R'. The letter 'L' indicates going left, whereas 'R' indicates going right.

■ **Example 6.6** In this tree, the empty string corresponds to the root 'A', "LL" corresponds to

’D’, ”LR” corresponds to ’E’ and ”RR” corresponds to ’F’.



,

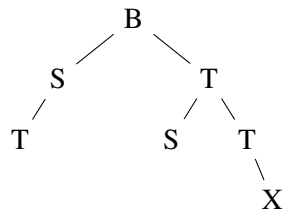
■

1. Write the method `private static void concat(char c, List<String> l)`, member of `BT` which concatenates the character `c` at the start of every string in `l`. If the list is empty, nothing happens.
2. Write the method `public T get(String path)`, member of `BT`, which returns the data of the node indicated by `path`. Assume that the tree is not empty and that `path` is valid.
3. Write the method `public List<String> leafPaths()`, member of `BT`, which returns the paths to all leaf nodes as strings. Use the method `private static void concat(char c, List<String> l)` above. If the tree is empty, empty list is returned. Note that in line 16 below, we are calling the method `concatLists(l1, l2)` which returns the concatenation of the two lists `l1` and `l2`.

Problem 6.14

A player wants to escape a maze where at most two options are available at each step: go straight or turn. This maze can be represented by a binary tree where the data is a character that can take four values: ’B’ for begin (only at the root), ’S’ for go straight, ’T’ for turn, or ’X’ meaning this is an exit. Exits are located at leaf nodes only, but not all leaf nodes are exits, they could be dead ends.

■ **Example 6.7** In this maze, you see that you reach the exit by moving: `T,T,X` staring from the root.



■

This is the class `MNode`:

```
class MNode {
    public char data;
    public MNode left, right;
    ...
}
```

1. Write the method `private boolean follow(MNode t, String path)`, which tests if the path indicated by `path` and starting from `t` is valid. A path is valid if its directions are available (not necessarily leading to an exit).

   ■ **Example 6.8** In the maze shown above, the paths: "TT", "TTX", "ST" are valid, whereas the paths "SS" and "TST" are not valid. ■

2. Write the method `private boolean escape(MNode t)`, which searches preorder for an exit starting at `t` and returns true if it finds one.

Problem 6.15

1. Write the method `private boolean f(BTNode<T> t, T e, int k)`, member of `BT`, which return true if `e` appears in `t` at a depth that is equal or greater than `k` (assume that `t` is at depth 0).
2. Repeat the same question as above, but this time as a user.

Problem 6.16

1. Write the method `public static <T> void prune(BT<T> bt, int l)`, user of `BT`, which removes all nodes that are at a depth larger than `l` (the root is at depth 0).
2. Write the method `List<T> pathToCurrent(BT<T> bt)`, which returns the path from the root to current.
3. ★ Write the method `List<List<T>> pathsToLeaves(BT<T> bt)`, which returns all paths from the root to all leaf nodes.

Problem 6.17

1. Write the method `public int depth()`, member of `LinkedBT`, which returns the depth of current (root is at depth 0).
2. Write the method `public boolean atDepth(T e, int l)`, which returns true if the data `e` is located at depth `l` (starting from 0 at the root). Notice that tree can contain duplicates, and it is enough that one occurrence satisfies the condition for the method to return true.
3. Modify the previous method by changing the condition from equality with `l` to $\leq$ and $\geq$ respectively.
4. Write the method `private boolean atSameDepth(BTNode<T> p, BTNode<T> q)`, which returns true if and only if `p` and `q` are at the same depth.
5. Write the method `private List<T> path(BTNode<T> p, BTNode<T> q)`, member of `LinkedBT`, which returns the path from `p` to `q`.
6. Write the method `public boolean isBal()`, member of the BT class, which returns true if the BT is a balanced, and false otherwise. A BT is balanced if for each node, the absolute difference in height of its two subtrees is at most 1. Assume you have a method called `private int height(BTNode<T> p)` that returns the height the sub-tree `p`. The method `isBal()` makes a call to the recursive method `private boolean isBalRec(BTNode<T> p)`.
7. Write the method `public boolean isOrd()` which is a member of the BT (binary tree) class. It returns true if for every node N the left child is smaller than N, and N is smaller than the right child. The method `public boolean isOrd()` calls the **recursive** method `private boolean isOrdRec(BTNode<T> p)`. Assume that the generic type `T` extends the interface `Comparable` and use the method `compareTo()` for comparison.

Problem 6.18

★ Write a **non-recursive** method `public static printPreorder(BT<String> bt)`, that prints all the content of a binary tree pre-order.

Problem 6.19

Write an implementation of `BT` where the node has a pointer to parent in addition to left and right. All methods must be $O(1)$.