

5. Recursion and Binary Trees

Problem 5.1


1. Write the method `reverseArray` that reverses an array of n elements. The method signature is `public void reverseArray(int[] A, int lo, int hi)`, where `lo` and `hi` are the lowest and highest index in the array, respectively.

■ **Example 5.1** Calling `reverseArray({2,4,5,12,5},0,4)`, will result in 5, 12, 5, 4, 2. ■

2. Write the recursive method `checkDuplicateElement` that checks if an element k , passed as parameter, exists at least twice in an array A of length n .
3. Write the method `isItSortedEven` that checks if the elements in the array's even indices are sorted. The array length is n . The method should return true if they are sorted, false otherwise. **Do not use any additional data structure.** The signature of the method is `public static boolean isItSortedEven(int[] a, int n)`

Problem 5.2

1. Write a recursive method, member of the class `LinkedList` that reverses the content of the list (**Do not use any auxiliary data structure and do not call any other method when writing this method**).

 Recursive member functions are private in general, since their parameters may depend on the internal representation of the data structure. Consequently, such methods are initially called from a non-recursive public member method. For example:

```
public class LinkedList<T> {  
  
    ...  
    private Node<T> recReverse(Node<T> p){ // The recursive method  
        ...  
    }  
  
    public void reverse(){// Non-recursive public method  
        head = recSwap(head); // Call to the recursive method  
    }  
}
```

```
}
}
```

Write the **recursive** method `void remove(List<T> l, T e)` that deletes **all the occurrences** of the element e from the list l starting from the current element and keeping all other elements in their order.

■ **Example 5.2** If $l : 2 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 2$, and current pointing to 3, then after calling `remove(l, 2)`, l becomes $2 \rightarrow 1 \rightarrow 3 \rightarrow 4$. ■

2. Write a recursive method `merge`, member of the class `LinkedList` that takes as input a list l_2 and merges it with the current list into one list. The current list and the input list must not be modified. The merge operation creates a new list by taking the first element from current list, then the first element from l_2 then the second element from current list then the second element from l_2 and so on. The method signature is `public LinkedList<T> merge(LinkedList<T> l2)`.

■ **Example 5.3** If the list l_1 contains: $A \rightarrow B \rightarrow C$, and l_2 contains: $D \rightarrow E \rightarrow F \rightarrow G \rightarrow H$, then the result of `l1.merge(l2)` is: $A \rightarrow D \rightarrow B \rightarrow E \rightarrow C \rightarrow F \rightarrow G \rightarrow H$. ■

3. Write a recursive method `merge(q1, q2)` that merges the queues q_1 and q_2 into a new queue. After the call, q_1 and q_2 become empty (**Do not use any loops**). The method signature is: `public <T> Queue<T> merge(Queue<T> q1, Queue<T> q2)`.
4. Rewrite the method `merge` so that the two queues q_1 and q_2 do not change after the call (**Do not use any loops**).

Problem 5.3

1. Write the **recursive** method `void removeEle(Stack<T> st, T e)` that deletes **all the occurrences** of the element e from the stack st keeping all other elements in their order.
2. Write the **recursive** method `int stackSum(Stack<Integer> st)` that sums all the elements in the stack and returns the total result. The stack must not be changed at the end of method.
3. Write a **recursive** method that checks if an array A of size n is sorted in increasing order.
4. Write the **recursive** method `isPalindrome`, member of the class `DoubleLinkedList` that checks if the list is a palindrome. The method signature is `public boolean isPalindrome()`.

Problem 5.4

The parts of this problem are related.

1. We add to the interface `List` the method `T car()`, which returns the first element of the list, and the method `List<T> cdr()`, which returns the rest of the elements as a new list. The method `car` cannot be called on an empty list. On the other hand, if the list is empty, the method `cdr` returns an empty list.

■ **Example 5.4** If $l : A, B, C, D$, then `l.car()` returns the element A , and `l.cdr()` returns the list B, C, D . ■

Implement these two methods as members of the class `LinkedList`.

2. Write the method `public static <T> List<T> list(T e)`, which returns a list containing the single element e .
3. Write the method `public static <T> List<T> concat(List<T> l1, List<T> l2)`, which returns the concatenation of the lists l_1 and l_2 . The two input lists must not change.

■ **Example 5.5** If $l_1 : A, B$ and $l_2 : C, D$, then `concat(l1, l2)` returns the list A, B, C, D . ■

4. Using the methods: `car`, `cdr`, `list` and `concat`, write the following **recursive methods**¹:

- The method `public static <T> void print(List<T> l)`, which prints the list `l`.
- The method `public static <T> List<T> inverse(List<T> l)`, which returns the inverse of the list `l`. The list `l` must not change.
- The method `public static <T> List<T> remove(List<T> l, T e)`, which returns the list obtained by removing all occurrences of `e` from `l`. The list `l` must not change.
- The method `public static <T> List<T> replace(List<T> l, T e1, T e2)`, which returns a list that is a copy of `l` except that all occurrences of `e1` are replaced by `e2`.
- ★★ Given a list `l` that has no duplicate elements, write the method named `public static <T> List<List<T>> subsets(List<T> l)` which returns all the subsets of `l`.

Problem 5.5

In this problem, do not use any auxiliary data structures (in particular, do not use a stack).

- Write a recursive method, `eval`, to evaluate a postfix expression. The expression is represented as a String and contains the following operators: `+`, `-`, `*` and `/`. For simplicity, assume that all the numbers are single digit and unsigned, for instance 5, or 6 but not 23, 124 or -4. An example of an input is: `"873-*4+23-*58-"`.

Programming hint: in order to transform a single character located at position i in a string `exp` to its numerical value, you may use:

```
val = Character.getNumericValue(exp.charAt(i));
```

- Write a recursive method, `infix`, to transform a postfix expression into an infix one. Use the same assumptions as in the previous question. For simplicity, put all operation between parentheses. For instance, the postfix expression `"23+"` is transformed to `"(2+3)"`, and `"873-*4+23-*58-"` is transformed to `"(((8*(7-3))+4)*(2-3))+(5-8)"`.

```
public class Postfix {

    // Private recursive method.
    private static double recEval(...) {
        ...
    }

    // Public non-recursive
    public static double eval(String exp) {
        ...
    }

    // Private recursive method.
    private static String recInfix(...) {
        ...
    }

    // Public non-recursive
    public static String infix(String exp) {
        ...
    }
}
```

¹The names of the methods `car` and `cdr` are those of two primitive operations in the Lisp language which were named this way for historical reasons (see https://en.wikipedia.org/wiki/CAR_and_CDR). Most programs in Lisp consist in clever use of these two primitives to recursively solve problems.

Problem 5.6

1. Show the result of method *traverse* for the binary tree in Figure 5.1 where we simply print the letters in all the nodes. Write the result for all three approaches: Inorder, Preorder and Postorder.

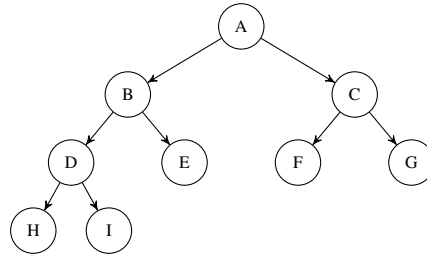


Figure 5.1: Binary Tree.

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 5.7

1. As a user of the ADT Binary Tree, write the instructions necessary to transform the tree shown in Figure 5.2 into the one shown in Figure 5.3 (let the tree be called `bt`, a variable of type `BT<String>`).

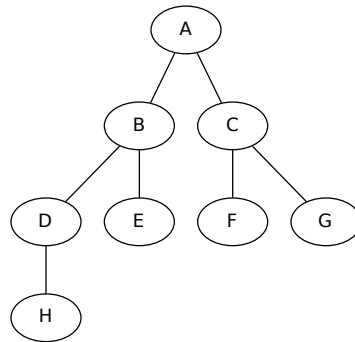


Figure 5.2: A binary tree (*H* is the left child of *D*).

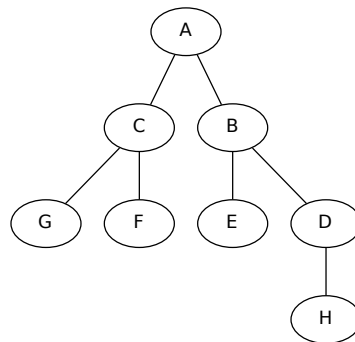


Figure 5.3: The mirror of the tree shown in Figure 5.2 (*H* is the right child of *D*).

2. Draw the tree shown in Figure 5.2 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 5.8

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 5.9

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 5.6** For the tree shown in Figure 5.3, the output to `collectInOrder()` is the list: $G \rightarrow C \rightarrow F \rightarrow A \rightarrow E \rightarrow B \rightarrow H \rightarrow D$. ■

2. Write the **recursive** method `mirror`, member of the class `BT`, that transforms the tree into its mirror (see Figure 5.3 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)`.

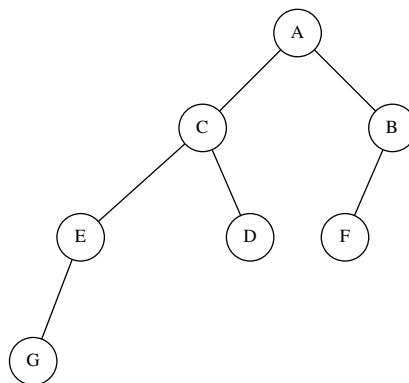


Figure 5.4: A binary tree.

■ **Example 5.7** In the tree shown in Figure 5.4, 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

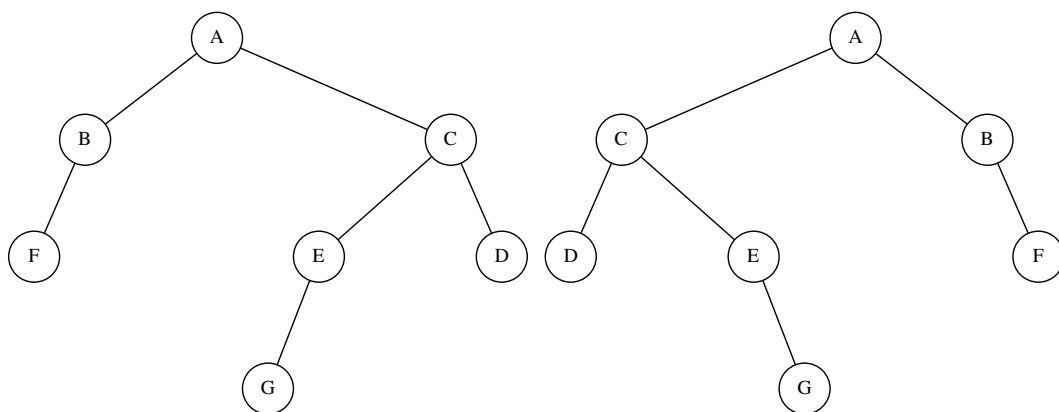
`recSizeBalanced`).

■ **Example 5.8** The binary tree shown in Figure 5.4 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 5.10

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 5.9** The two trees shown below are mirror images of each other.

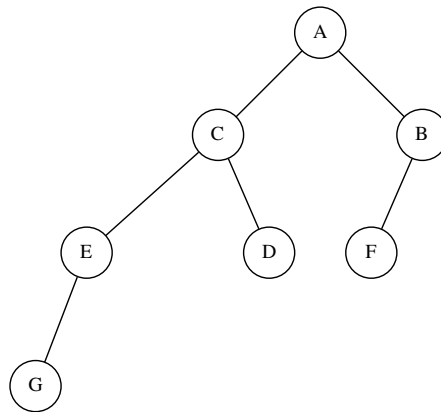


2. Write the method `boolean equal(BTNode<T> t1, BTNode<T> t2)`, member of `LinkedBT`, which returns true if the sub-trees `t1` and `t2` are equal (have the same data).
3. Write the **recursive** method `twoChildren`, member of the class `BT` (Binary Tree), which returns the number of nodes with two 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 5.11

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.

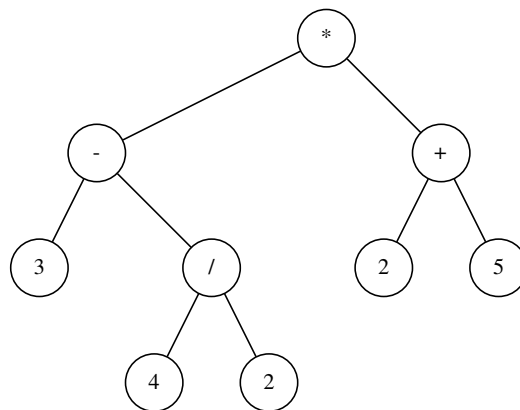
■ **Example 5.10** The call `atLevel(2)` on the binary tree shown below returns 2, `atLevel(3)` returns 2, `atLevel(1)` returns 1, `atLevel(6)` returns 0. ■



2. Write the method `public void cut(int depth)` in the class `LinkedBT` that cuts all nodes at a depth larger than `depth` (root is at depth 0).

Problem 5.12

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 5.13

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 5.14

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 5.15

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 5.16

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.

Problem 5.17

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 5.18

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`.

Problem 5.19

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

Problem 5.20

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