Name: Sarvagnya H. Purohit **Registration Number**: 201070056
Date: 08-12-2021 **Branch:** Computer Engineering
Course: DSA **Course Instructor:** Dr. Mahesh Shirole

Lab Assignment 5

Aim: To perform implementation of applications of general trees and binary trees.

Theory:

Tree:
A tree is an abstract data type that stores elements hierarchically. With the exception of the top element (also called as the "root"), each element in a tree has a parent element and zero or more children elements.

Tree ADT:
The tree ADT accessor methods, allowing a user to navigate the various positions of a tree T:

- root(): Returns the position of the root of the tree (or null if empty)
- parent(p): Returns the position of the parent of position p (or null if p is the root)
- children(p): Returns an iterable collection containing the children of position p (if any)
- numChildren(p): Returns the number of children of position p

The tree ADT supports the following query methods:

- isInternal(p): Returns true if position p has at least one child
- isExternal(p): Returns true if position p does not have any children
- isRoot(p): Returns true if position p is the root of the tree

The tree ADT supports the following generic methods:

- size(): Returns the number of positions (and hence elements) that are contained in the tree
- isEmpty(): Returns true if the tree does not contain any positions (and thus no elements)
- iterator(): Returns an iterator for all elements in the tree (so that the tree itself is Iterable)
- positions(): Returns an iterable collection of all positions of the tree

Abstract Class in Java:
An abstract class may define concrete implementations for some of its methods, while leaving other abstract methods without definition. An abstract class is designed to serve as a base class, through inheritance, for one or more concrete implementations of an interface. When some of the functionality of an interface is implemented in an abstract class, less work remains to complete a concrete implementation.

__**Binary Tree**__:
A binary tree is an ordered tree with the following properties:

1. Every node has at most two children
2. Each child node is labeled as being either a left child or a right child
3. A left child precedes a right child in the order of children of a node

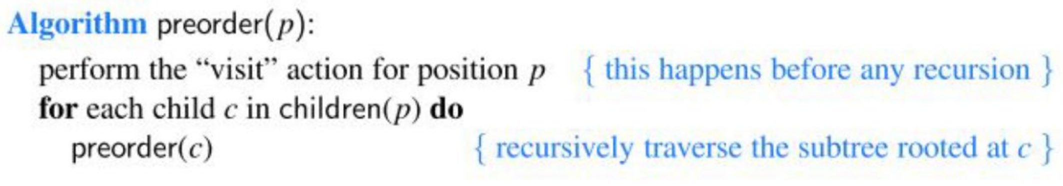
__**Implementing Binary Trees as a Linked Structure, Operations Performed**__:
__Node Structure:__ maintains references to the element stored at a position p and to the nodes associated with the children and parent of p.
If p is the root of T, then the parent field of p is null. Similarly, if p does not have a left child (respectively, right child), the associated field is null. The tree itself maintains an instance variable storing a reference to the root node (if any), and a variable, called size, that represents the overall number of nodes of T.

__Operations__: They consist of the following update methods:

- addRoot(e): Creates a root for an empty tree, storing e as the element, and returns the position of that root; an error occurs if the tree is not empty.
- addLeft(p, e): Creates a left child of position p, storing element e, and returns the position of the new node; an error occurs if p already has a left child.
- addRight(p, e): Creates a right child of position p, storing element e, and returns the position of the new node; an error occurs if p already has a right child.
- set(p, e): Replaces the element stored at position p with element e, and returns the previously stored element.
- attach(p, T1, T2): Attaches the internal structure of trees T1 and T2 as the respective left and right subtrees of leaf position p and resets T1 and T2 to empty trees; an error condition occurs if p is not a leaf.
- remove(p): Removes the node at position p, replacing it with its child (if any), and returns the element that had been stored at p; an error occurs if p has two children

__**Tree Traversals**__: A traversal of a tree T is a systematic way of accessing, or "visiting," all the positions of T.

__Preorder Traversal__: In a preorder traversal of a tree T, the root of T is visited first and then the subtrees rooted at its children are traversed recursively. If the tree is ordered, then the subtrees are traversed according to the order of the children.



```
Algorithm preorder(p):
  perform the "visit" action for position p   { this happens before any recursion }
  for each child c in children(p) do
    preorder(c)                               { recursively traverse the subtree rooted at c }
```

Post-order Traversal: The post-order traversal recursively traverses the subtrees rooted at the children of the root first, and then visits the root.

```
Algorithm postorder(p):
  for each child c in children(p) do
    postorder(c)                          { recursively traverse the subtree rooted at c }
  perform the "visit" action for position p    { this happens after any recursion }
```

Breadth-First Tree Traversal: Breadth-First Tree Traversal traverses a tree so that we visit all the positions at depth d before we visit the positions at depth d+1.
A breadth-first traversal is not recursive, since we are not traversing entire subtrees at once. We use a queue to produce a FIFO (i.e., first-in first-out) semantics for the order in which we visit nodes. The overall running time is O(n), due to the n calls to enqueue and n calls to dequeue.

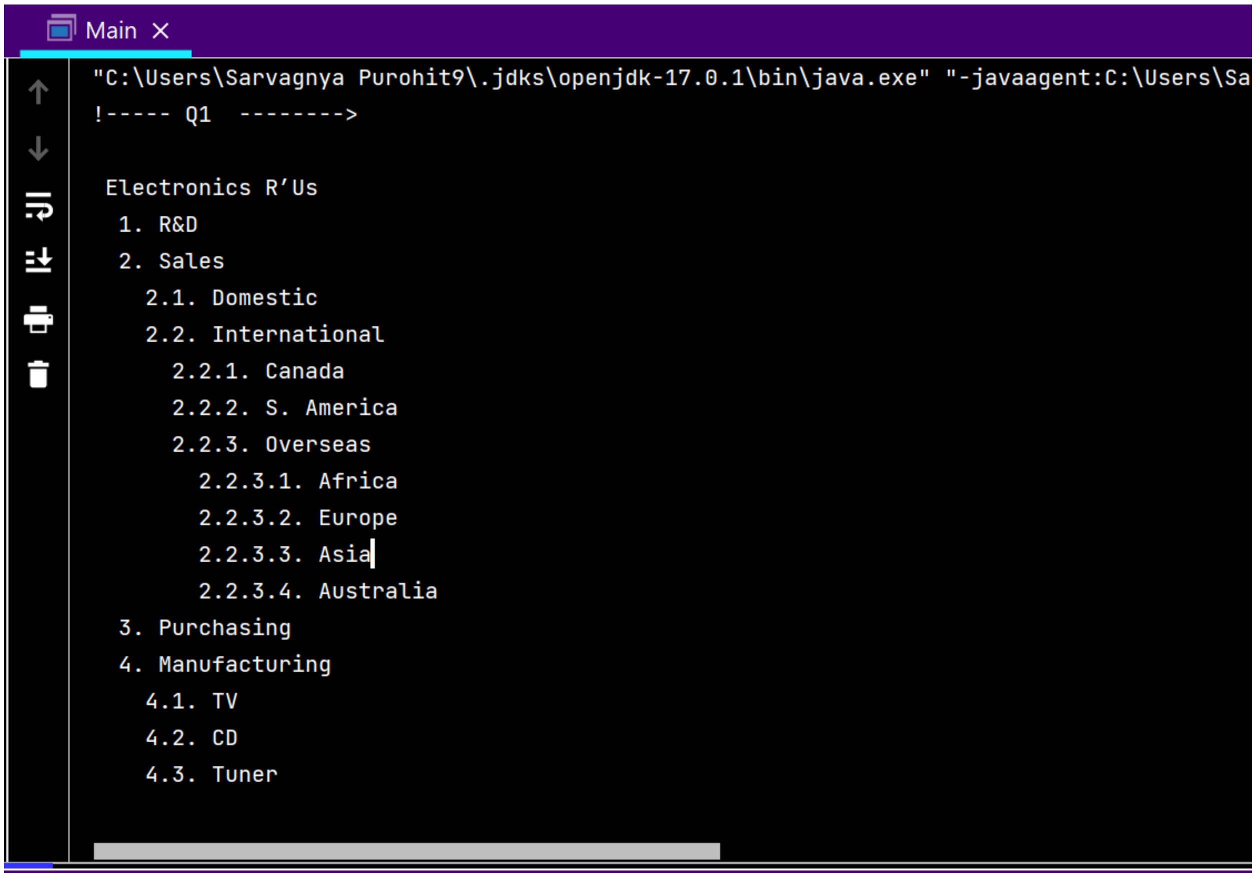
```
Algorithm breadthfirst():
  Initialize queue Q to contain root()
  while Q not empty do
    p = Q.dequeue()                       { p is the oldest entry in the queue }
    perform the "visit" action for position p
    for each child c in children(p) do
      Q.enqueue(c)    { add p's children to the end of the queue for later visits }
```

In-order Traversal: An in-order traversal visits a position between the recursive traversals of its left and right subtrees. The in-order traversal of a binary tree T can be informally viewed as visiting the nodes of T "from left to right".

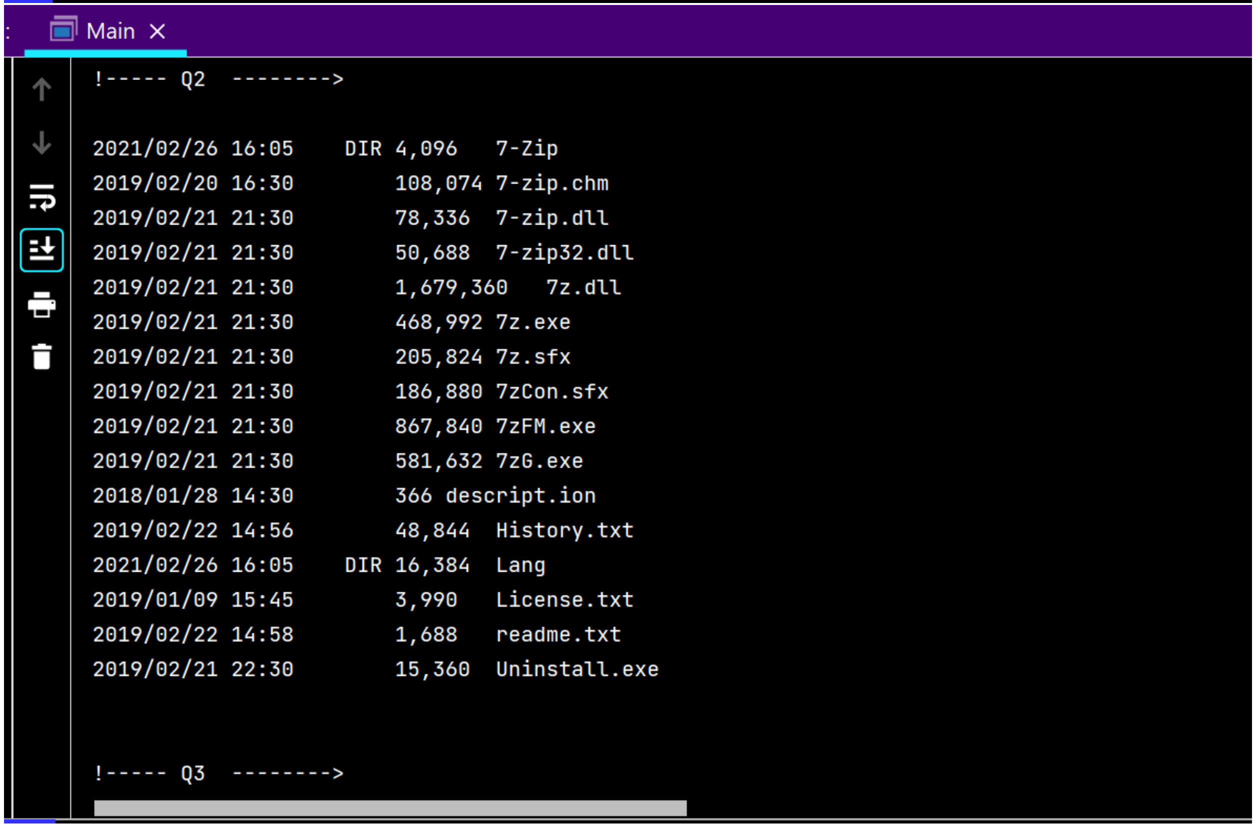
```
Algorithm inorder(p):
  if p has a left child lc then
    inorder(lc)                     { recursively traverse the left subtree of p }
  perform the "visit" action for position p
  if p has a right child rc then
    inorder(rc)                     { recursively traverse the right subtree of p }
```

Test Data and Output of the Program:

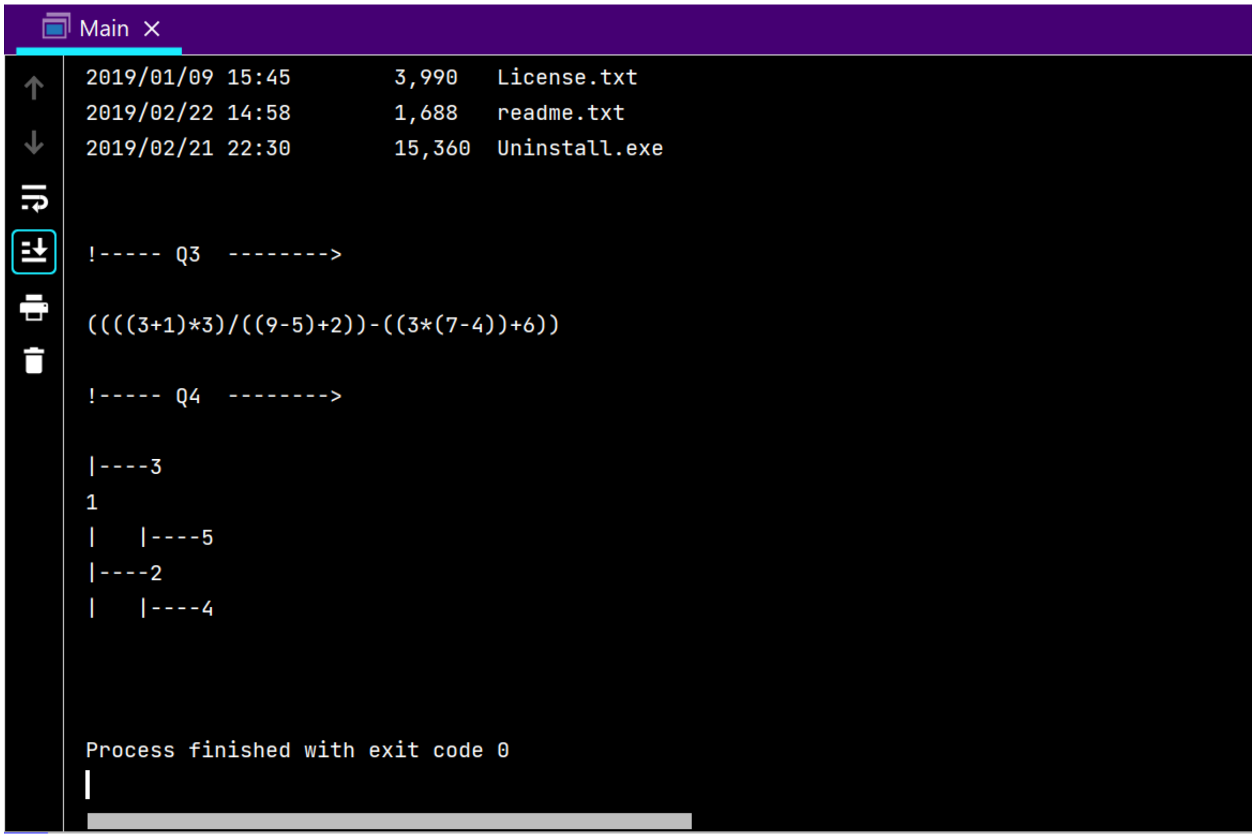
(on the next page)



```
"C:\Users\Sarvagnya Purohit9\.jdks\openjdk-17.0.1\bin\java.exe" "-javaagent:C:\Users\Sa
!----- Q1  -------->

 Electronics R'Us
  1. R&D
  2. Sales
    2.1. Domestic
    2.2. International
      2.2.1. Canada
      2.2.2. S. America
      2.2.3. Overseas
        2.2.3.1. Africa
        2.2.3.2. Europe
        2.2.3.3. Asia
        2.2.3.4. Australia
  3. Purchasing
  4. Manufacturing
    4.1. TV
    4.2. CD
    4.3. Tuner
```



```
!----- Q2  -------->

2021/02/26 16:05    DIR 4,096   7-Zip
2019/02/20 16:30        108,074 7-zip.chm
2019/02/21 21:30        78,336  7-zip.dll
2019/02/21 21:30        50,688  7-zip32.dll
2019/02/21 21:30        1,679,360   7z.dll
2019/02/21 21:30        468,992 7z.exe
2019/02/21 21:30        205,824 7z.sfx
2019/02/21 21:30        186,880 7zCon.sfx
2019/02/21 21:30        867,840 7zFM.exe
2019/02/21 21:30        581,632 7zG.exe
2018/01/28 14:30        366 descript.ion
2019/02/22 14:56        48,844  History.txt
2021/02/26 16:05    DIR 16,384  Lang
2019/01/09 15:45        3,990   License.txt
2019/02/22 14:58        1,688   readme.txt
2019/02/21 22:30        15,360  Uninstall.exe


!----- Q3  -------->
```



```
2019/01/09 15:45        3,990   License.txt
2019/02/22 14:58        1,688   readme.txt
2019/02/21 22:30        15,360  Uninstall.exe


!----- Q3  -------->

((((3+1)*3)/((9-5)+2))-((3*(7-4))+6))

!----- Q4  -------->

|----3
1
|   |----5
|----2
|   |----4



Process finished with exit code 0
```

Conclusion:

In this lab assignment, we learned to apply trees and tree traversals in various use-cases we encounter in real-world applications. They included printing indented lists (like an index of contents), computing disk space of a directory and displaying the structure of that directory, using in-order traversal to evaluate an arithmetic expression with parentheses, and graphically printing a tree on the console using ASCII characters. We learnt how trees are implemented using a linked structure involving various nodes and parent-child relationships b/w them. Each of these applications involved the use of recursion to repeat some operation recursively on any subtree of a tree. We also learnt about abstract classes in Java, which can be used as a base class for other concrete classes.