# Lab Tutorial for Week 5: Tree Data Structure

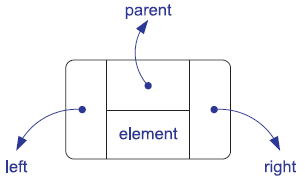
## Contents

1. Tutorial w5a: Linked Binary Tree
2. Tutorial w5b: Tree Traversal

## Tutorial w5a: Linked Binary Tree

### This practice presents an implementation of a binary tree as a linked structure, called LinkedBinaryTree. For simplicity, let us assume that each node has either zero or two children.

### A node in a linked data structure for representing a binary tree looks like below:



typedef string Elem;

struct Node {

Elem elt; // element value

Node\* par; // parent

Node\* left; // left child

Node\* right; // right child

};

### To track the position of a node in the tree, we create a class Position whose data member consists of a pointer v to a node of the tree. You can imagine the Position object is like current pointer in a linked list which tracks the current position of a node. Access to the node’s element is provided by overloading the dereferencing operator (“\*”).

class Position {

private:

Node\* v; // pointer to the node

public:

// constructor

Position(Node\*)

// get element

Elem& operator\*();

// get left child

Position left() const;

// get right child

Position right() const;

// get parent

Position parent() const;

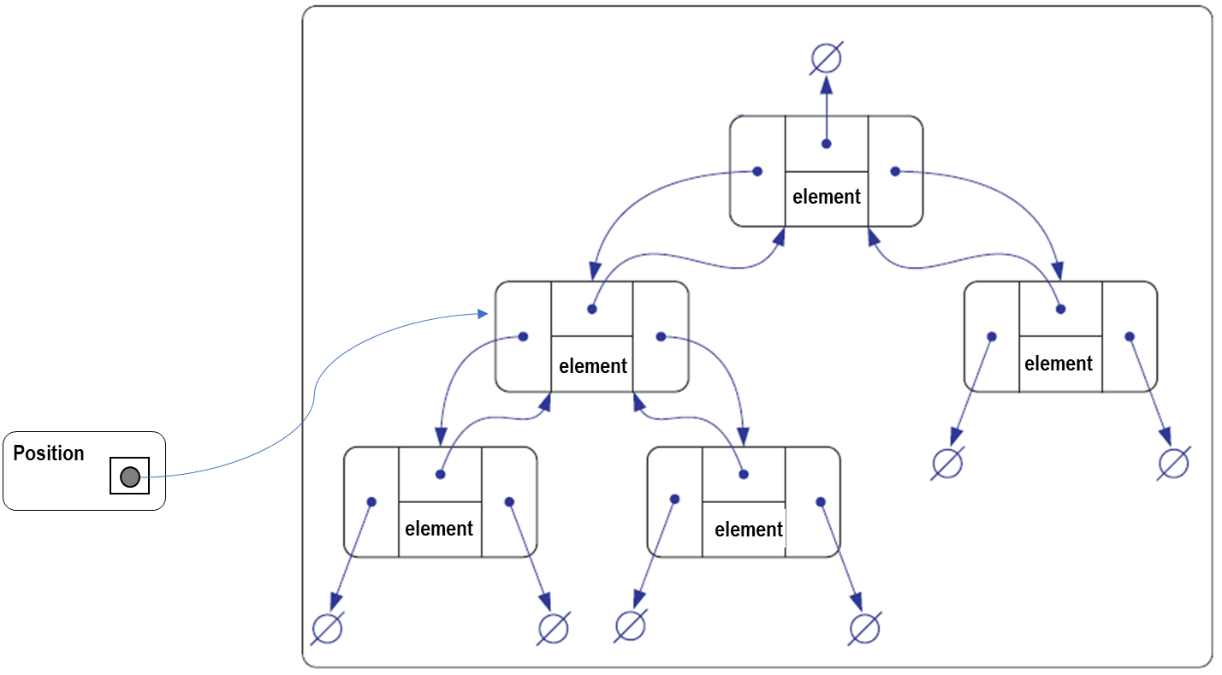
// root of the tree?

bool isRoot() const;

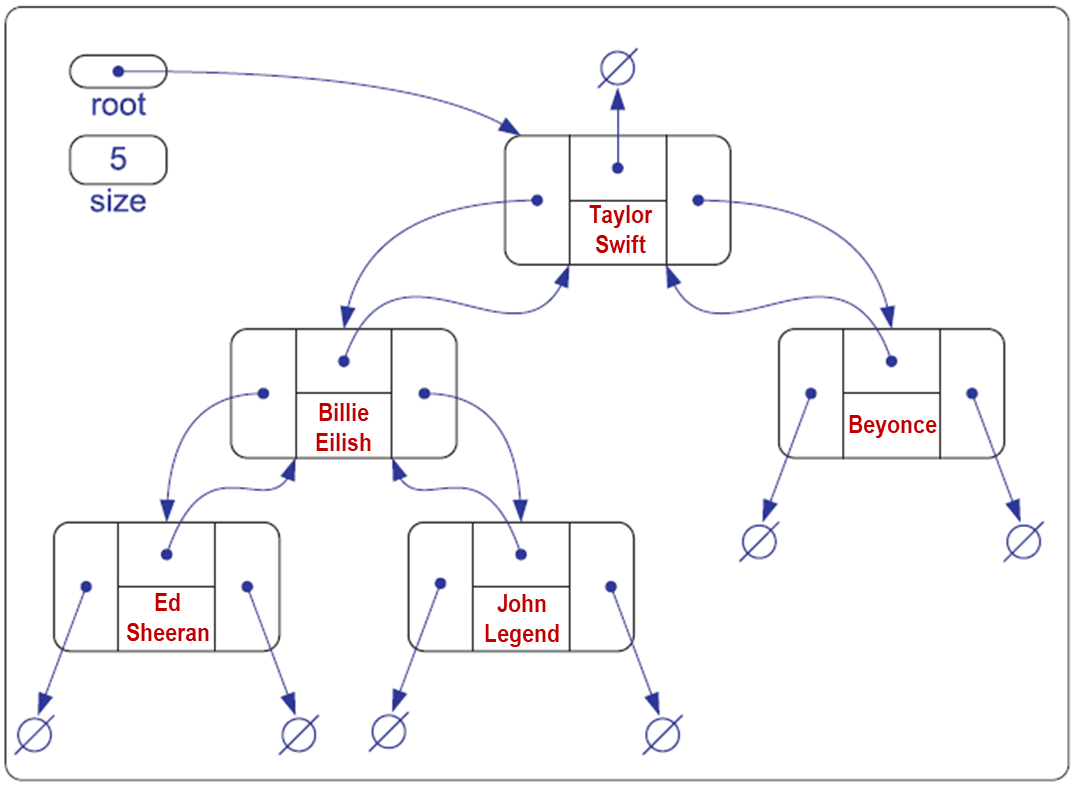
// an external node?

bool isExternal() const;

};



An example of a linked structure representation of a binary tree is shown below. The structure stores the tree’s size, that is, the number of nodes in the tree, and a pointer to the root of the tree. The rest of the structure consists of the nodes linked together appropriately. If v is the root of T, then the pointer to the parent node is NULL, and if v is an external node (**leave**), then the pointers to the children of v are NULL.



### Practice:

1. Open the starter file (LinkedBinaryTree-startercode.zip) in your IDE.
2. This file contains a linked implementation of a Binary Tree.

// a Binary Tree implemented using Linked data structure

class LinkedBinaryTree {

public:

LinkedBinaryTree(); // constructor

int size() const; // number of nodes

bool empty() const; // is tree empty?

Position getRoot() const; // get the root

PositionList positions(int) const; // list of nodes

void addRoot(); // add root to empty tree

void expandExternal(const Position& p); // expand external node

Position removeAboveExternal(const Position& p); // remove p and parent

void preorder(Node\* v, PositionList& pl) const; // preorder utility

void inorder(Node\* v, PositionList& pl) const; // inorder utility

void postorder(Node\* v, PositionList& pl) const; // postorder utility

// NOTE: housekeeping functions such as destructor, copy constructor and

// assignment operator are omitted

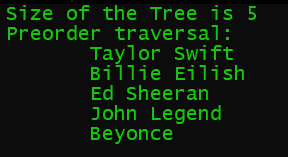
private:

Node\* root; // pointer to the root

int n; // number of nodes

};

1. Try to understand how the code works, notably the functions to build the tree given a position p. In addition to the addRoot function which creates the root node of the Tree, two functions below insert a node into and remove a node from the Tree.
   1. expandExternal(p): Transform p from an external node (**leave**) into an internal node by creating two new external nodes and making them the left and right children of p, respectively; an error condition occurs if p is an internal node.
   2. removeAboveExternal(p): Remove the external node p together with its parent q, replacing q with the sibling of p; an error condition occurs if p is an internal node or p is the root.
2. Please, compile the starter code and when you run it for the pre-order traversal, it will print the following output:



### NOTE: This program exemplifies a good separation between INTERFACE and IMPLEMENTATION. The header (.h) files represent the interface which specify WHAT the program can do, while the .cpp files is the implementation which implements HOW the program is doing in details.

### Questions:

1. If you notice, the top lines of the header files contain the conditional compilation directives, such as these:

#ifndef \_LINKED\_BINARY\_TREE

#define \_LINKED\_BINARY\_TREE

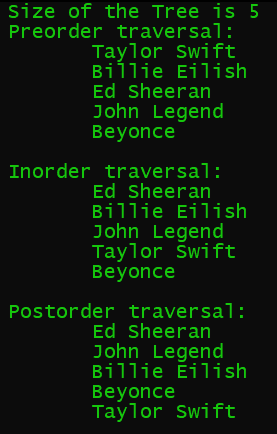
Why do we need these directives?

1. What is operator overloading? Why do you need it?

## Tutorial w5b: Implementing Tree Traversal Functions

### Practice:

1. Your task is to complete one or both functions to traverse the tree: in-order and post-order traversal. (*see the hints at the back pages of this tutorial*)
2. Save the completed source file and make sure it’s compiled.
3. Test your Tree with the following traversal:
   1. **Inorder**. What sequence of nodes you have? Please describe how you can get this sequence.
   2. **Postorder**. What sequence of nodes you have? Please describe how you can get this sequence.
4. Make sure that your program has the right output for each traversal like below:



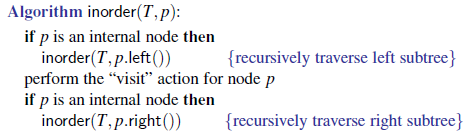
### Questions:

1. In your program, your Tree contains string data. What if someone needs to store other types of data, such as integer, floating point, or even user-defined data types (objects)? How will you revise your code to accommodate the need to store other data types?
2. In the main function (tester.cpp), you may have noticed that the code creates the Tree manually (like drawing each node one by one and assign the data also one by one). How will you do this task more automatically, for example by creating a function that prompts the users to input the data or read it from a file then do a loop to create all the nodes of the Tree while also assigning the right data in each node?
3. In this practice, you have implemented a Tree using Linked structures. Do you think it’s possible to implement a Tree using Arrays? What are the limitations?

---end of Tutorial Week 5---

# Hints for Inorder Tree Traversal

In the inorder traversal, we visit a node between the recursive traversals of its left and right subtrees. For a Binary Tree, the inorder traversal is given by the following algorithm:



# Hints for Postorder Tree Traversal

In the postorder traversal, it recursively traverses the subtrees rooted at the children of the root first, and then visits the root. For a Binary Tree, the postorder traversal is given by the following algorithm:

