COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

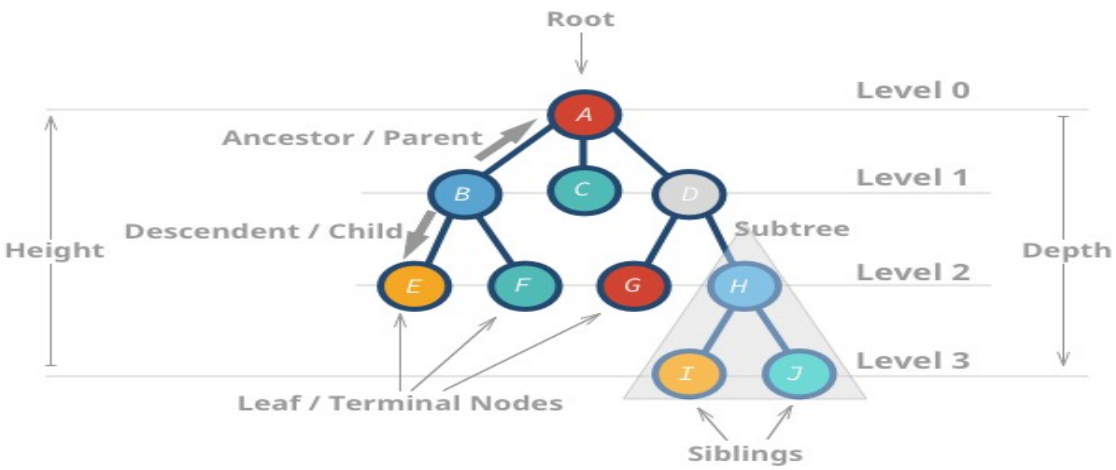
Lecture 15: Tree

Lecturer: Dr. Andrew Hines *Scribes: Enxi Cui & Xuejiao Ge*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

15.1 Outline

This session introduces a new data structure called **Tree**. First it shows the differences of **linear and non-linear** data structures. Then there are several examples from daily life to help students to understand what is/is not a tree structure. After generalizing the definition of tree as an abstract data type(ADT), it shows two ways to represent a tree: **Array-based and Linked List-based** and two methods to searching/traversing a tree(**DFS and BFS**). Additionally, it explains another important term called **Binary Tree**, which is frequently mentioned in interviews.



15.1.1 Linear vs. Non-Linear data structures

Linear data structures:

1-element(objects) are sequential and ordered in a way so that:

2-there is only one first element and has only one next element

3-there is only one last element and has only one previous element, while

4-all other elements have a next and a previous element

(Codeandwork.github.io, n.d.)

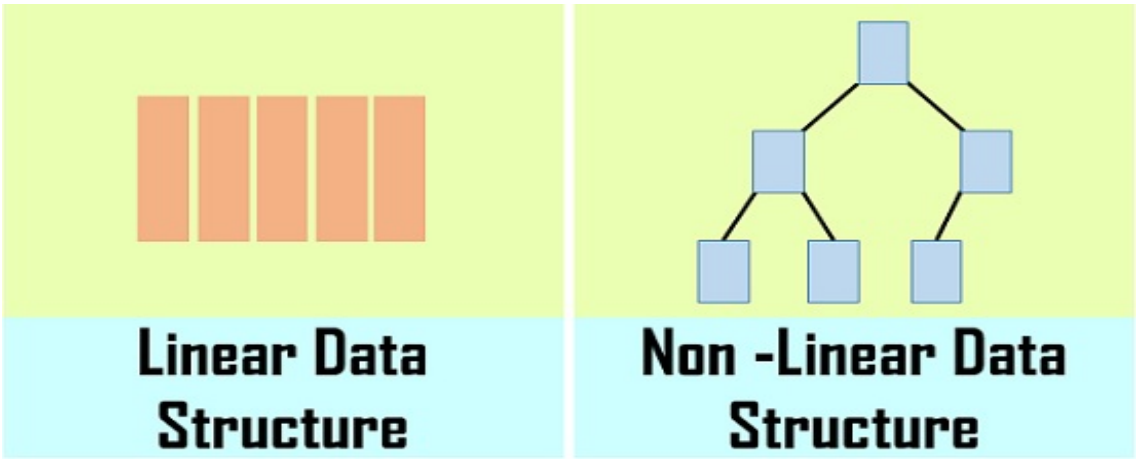
Popular linear data structures:

-Arrays

-Stack

-Queue

-Lists(Abhiraj, 2016)



A **non-linear data structure** is a data item is connected to several other data items in the data structure. In other words, a data element in a non-linear data structure can be connected to multiple elements to reflect the special relationship among them. Elements in a non-linear data structure can not be traversed all by once.(Nash, 2009)

Tree is a typical example of a non-linear data structure. Tree has one root node that acts as a starting point and links to other nodes. Graphs, File system, Data base system, languages (programming) have a hierarchical structure = notion of Abstract Syntax Tree are all non-linear data structures.

15.2 Terminologies of Trees

15.2.1 Defition of tree

-An abstract data type that stores elements hierarchically.

-Each element has a **parent** element and zero or more **child**(children) element(s).

-The top element is called the **root** of the tree, branches from the root node.

-A tree, T, is a set of **nodes** storing elements and the nodes have a **parent-child** relationship that fulfill the properties following:

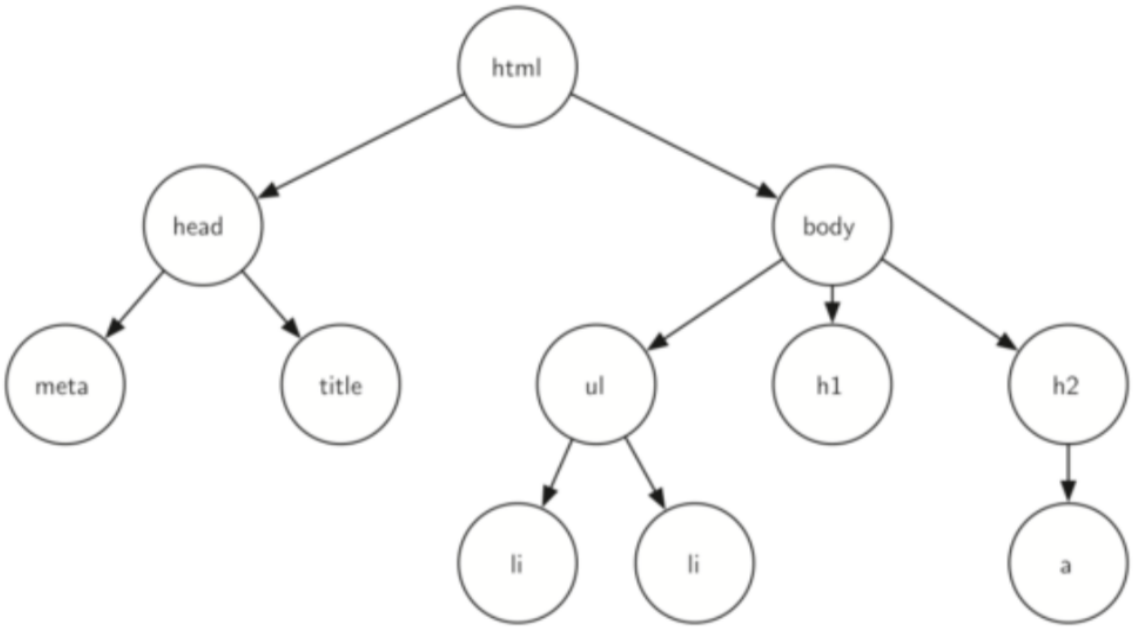
—-If T is not empty, it has a special node, called the **root** of T, that has no parent.

—-Each node v of different from the root has a unique **parent** node w; every node with parent w is a child of w.

15.2.2 Tree Examples

15.2.2.1 HTML tags

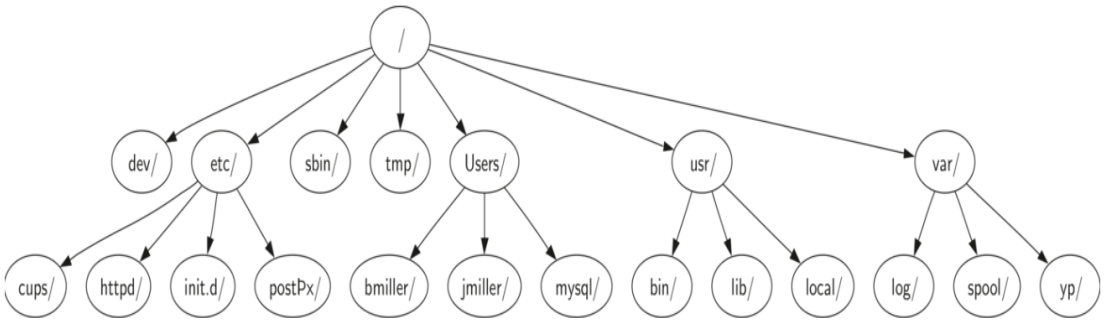
The tree of the html tags shows a good hierarchy structure, with each layer of the tree corresponding to a nested layer within the tag. The first one in the file is $\langle html \rangle$ and the last one is $\langle \backslash html \rangle$. All the other tags in the page are in pairs.(Miller and Ranum, 2013)




```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
    <li>List item one</li>
    <li>List item two</li>
</ul>
<h2><a href="http://www.cs.luther.edu">Luther CS </a><h2>
</body>
</html>
```

15.2.2.2 File System Trees

In the file system tree, we can go from the root directory to any other directories. Another important property of the tree is the ability to move subtree to a different location in the tree without affecting the lower level of the hierarchy. For example, we could take the entire subtree from the root directory and reconnect it. This will transform the subtree path without affecting any of its sub-directories



15.2.2.3 Node Relations

Siblings: two nodes children of the same parent.

If there is a meaningful linear order among the children of each node: the first, second, third etc. node. Such an order is usually visualised by arranging siblings left to right, according to their order.—-a.k.a, **tree is ordered**.

External: a node with no children also called a leaf.

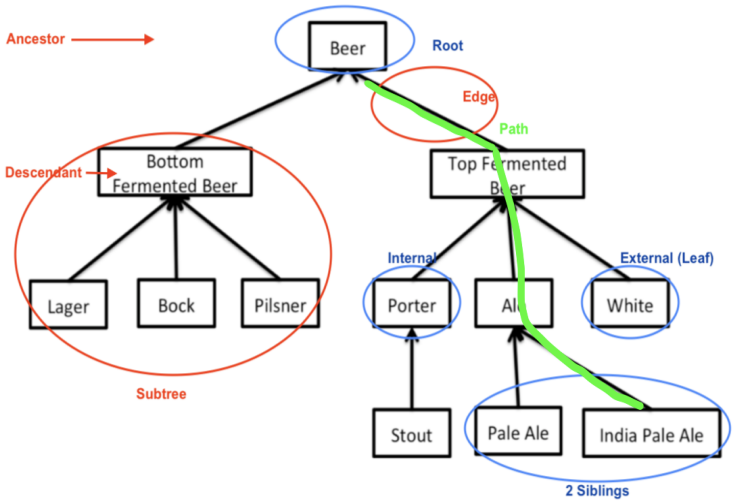
Internal: a node with at least one child.

Ancestor: a node u is an ancestor of node v if u=v or u is an ancestor of the parent of v.

Descendant: a node v is a descendant of a node u if u is an ancestor of v. **Subtree:** the subtree of a tree T rooted at a node v is the tree consisting of all the descendants of v in T (including v).

Edge: an edge of tree T is a pair of nodes (u,v) such that u is the parent of v, or vice versa.

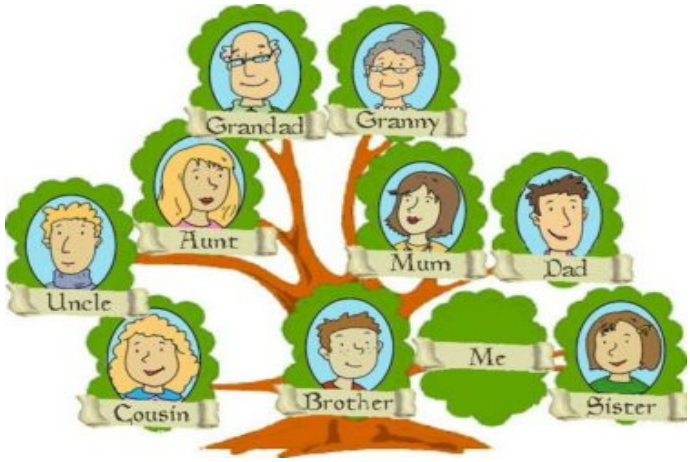
Path: The path of t is a sequence of nodes in which any two adjacent nodes form an edge. There is only one path between any node and the root.



15.2.3 Not a Tree – Examples

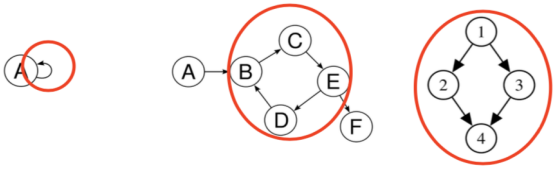
15.2.3.1 Family Tree

While a family tree is a familiar tree, it is actually a directed acyclic graph (DAC) ADT as relatives can mate but cannot be there own ancestor.



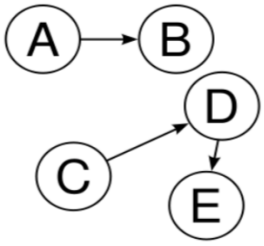
15.2.3.2 Cycle

Trees do not include directed or undirected cycles.



15.2.3.3 Non-connected

Trees do not have non-connected parts.



(But Forest as trees sometimes, i.e. a group of trees.)

15.2.4 Depth & Height of Trees

Depth: How far away a node is from the root. The root of a tree has depth 0.

Height: The depth of the lowest leaf/external node. In other words, the length of the longest path from the root to a leaf. An empty tree has height 0.

Algorithm 1: Computing Height

```
1  Algorithm height;
   Input  : my tree a tree
   Output: number of hops on longest path to a leaf
2  if my tree is empty then
3      return 0
4  else
5      maxheight ←0
6      for each child c of my tree do
7          maxheight ← max (height(c), max height)
8      end for
9      return 1 + max height
10 endif
```

15.2.5 The Tree ADT Methods

`create_empty_tree()`: creates an empty tree.

`create_tree(n)`: creates a one node tree whose root is Node n.

`add_child(tree)`: add a subtree to the current tree.

`is_empty()`: determines whether a tree is empty.

`get_root()`: retrieves the Node that forms the root of a tree.

`remove_child(tree)`: detaches a subtree from the current tree.

15.3 Tree representation

Trees can be represented in two way:

1) Array Representation (Sequential Representation).

–Array of arrays

–Matrix

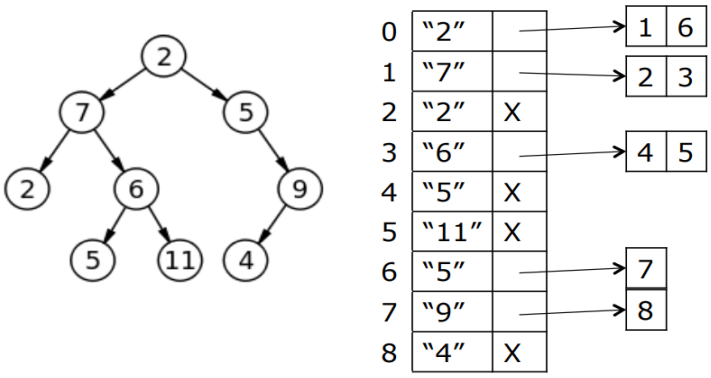
2) Dynamic Node Representation (Linked Representation).

(Katiyar, 2018)

15.3.1 Array-based Representation of Trees(Array of arrays)

–Each representing the children of a node.

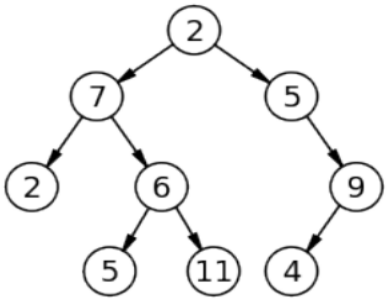
–Array indices 0-8 are the nodes (top to bottom from left). Array contains node value and an array of child node indices. e.g. Root node has child nodes stored at indices 1 and 6.



15.3.2 Array-based Representation of Trees(A matrix)

–Can be very sparse if one node has lot of children.

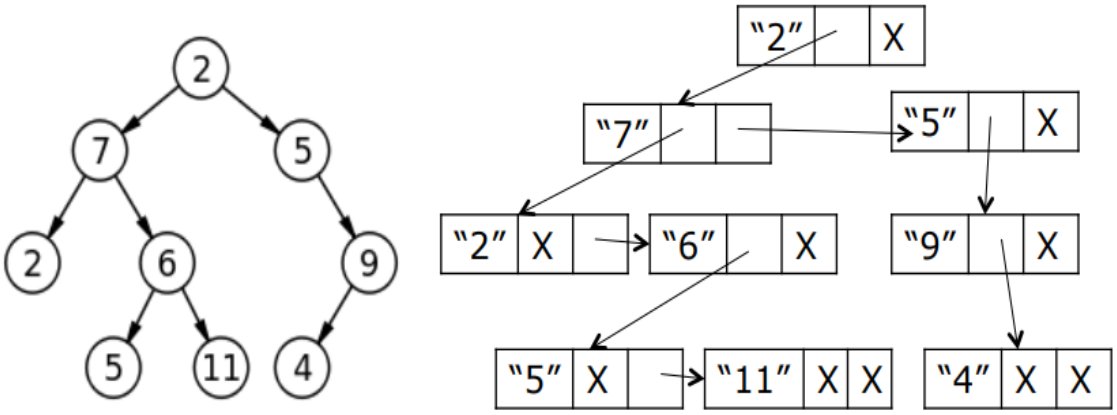
–Evolution likely to be easier to manage.



	label	0	1	2	3	4	5	6	7	8
0	"2"		1					1		
1	"7"			1	1					
2	"2"									
3	"6"					1	1			
4	"5"									
5	"11"									
6	"5"								1	
7	"9"									1
8	"4"									

15.3.3 Linked List-based Representation of Trees

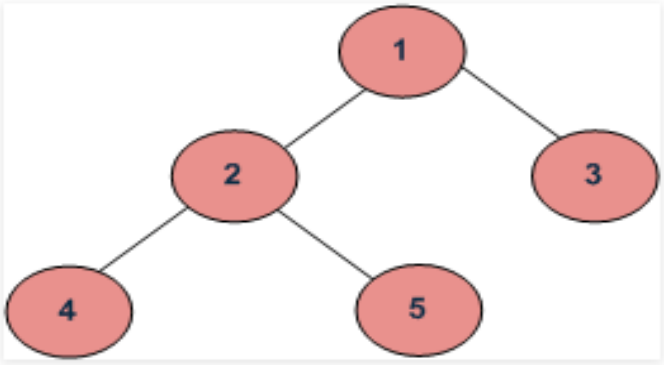
–Using a doubly linked list



15.4 Tree Traversing

Different from linear ADT which one have just a single logical way to traverse the path, trees can be traversed in different ways.

15.4.0.1 Depth First Traversals



For depth-first search, it requires a stack data structure, which follows a Last In, First Out (LIFO) rule.

(Betances, 2018)

(a) Inorder search (Left-Root-Right) order: 4 2 5 1 3

(b) Preorder search (Root-Left-Right) order: 1 2 4 5 3 – most commonly used Depth First Search(DFS)

(c) Postorder search (Left-Right-Root) order: 4 5 2 3 1

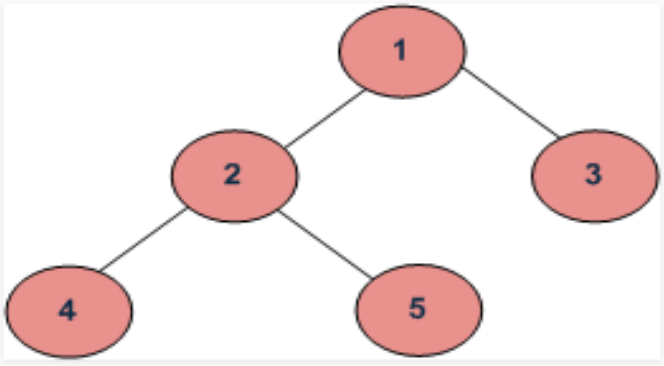
Algorithm 2: Depth First Search (recursive)

```
1 Algorithm dfs:;
  Input  : Tree t and node n
  Output: the function explores every node from n
2 if n is a leaf then # base case
3       do something
4 else
5 for each child n_c of n do dfs(n_c)
6           do something
7       endfor
8 endif
```

Algorithm 3: Depth First Search (non-recursive)

```
1 Algorithm dfs:;
  Input  : the function explores every node from n
  Output: average of the 4 numbers
2 to_visit ← empty stack
3 add n to to visit
4 while to visit is not empty do
5       current ← pop to visit # get the first element
6       push all children of current to to visit
7       do something on current
8 endfor
```

15.4.0.2 Breadth First Traversals



Breadth First Traversals use the queue data structure to be able to visit every node from left to right in every level. A queue is a linear data structure that follows the First In, First Out (FIFO) rule.(Betances, 2018)

Breadth First Search order: 1 2 3 4 5

Algorithm 4: Breadth First Search (non-recursive)

```
1  Algorithm bfs:;
   Input  : Tree t and node n (root or not)
   Output: explores every node of t rooted at n
2  to visit is a queue
3  enqueue n
4  while to visit is not empty do
5      n_current ← dequeueto_visit
6      for each child nc of n current do
7          enqueue n_c to to visit
8      endfor
9      do something on n current
10 endwhile
```

Algorithm 5: Breadth First Search (recursive)

```
1  Algorithm bfs:;
   Input  : queue q (originally having the root of the tree)
   Output: explores every node of t rooted at n
2  if q is empty then # base case
3      do something (?)
4  else
5      current ← dequeueq
6      for each child n_c of n do
7          enqueue n_c
8      endfor
9      do something
10     bfs(q)
11 endif
```

15.5 Binary Tree

15.5.1 Definition

A binary tree is a tree in which each node refers to zero, one, or two dependent nodes.(Wentworth et al., 2012)

Because every node in a binary tree can have only 2 children, we just call them the left and right child.

15.5.2 Building a binary tree

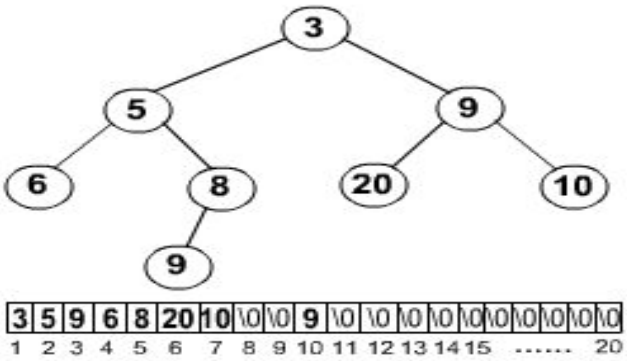
```
class Tree:
    def __init__(self, cargo, left=None, right=None):
        self.cargo = cargo
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.cargo)
```

```
tree = Tree(1, Tree(2), Tree(3))
```

15.5.3 Array-based Binary Tree

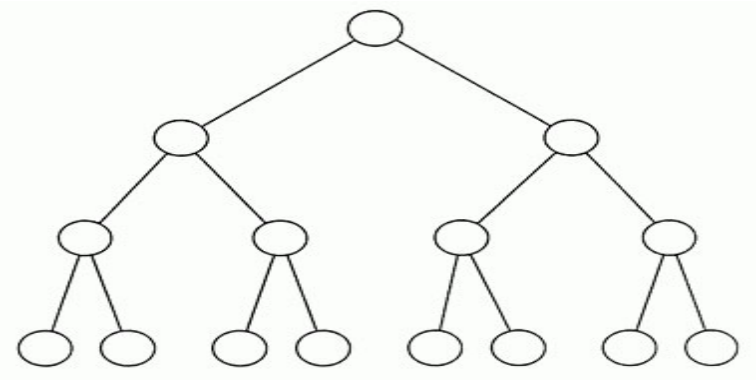
We can use a single array to represent a binary tree.

All the nodes are numbered / indexed according to a scheme giving 0 to root. Then all the nodes are numbered from left to right level by level from top to bottom. Empty nodes are also numbered. Then each node having an index i is put into the array as its ith element.(IIT GUWAHATI, 2009)

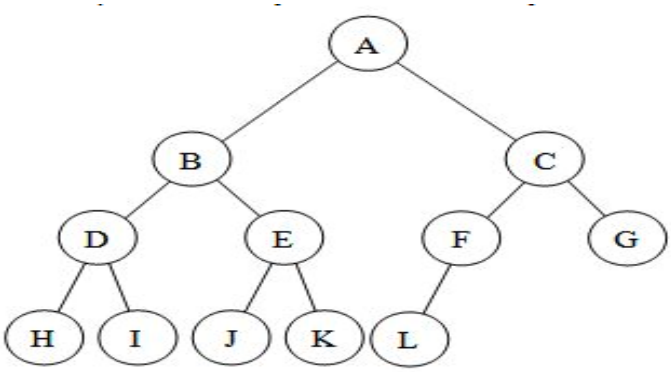


15.5.4 Full Binary Tree and Complete Binary Tree

Full Binary Tree

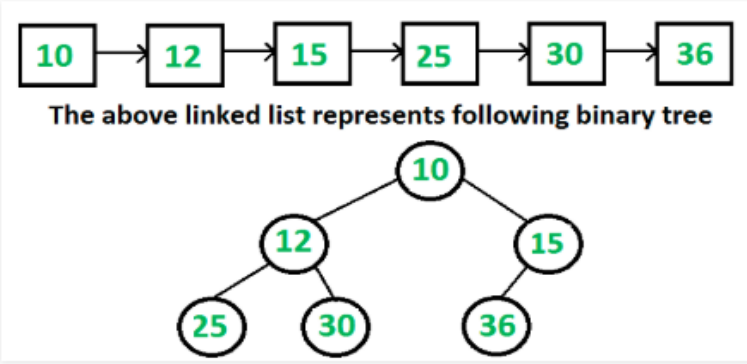


Complete Binary Tree



15.5.5 Linked List-based Binary Tree

We could also get a complete binary tree from a linked list representation.(Enaganti, 2013)



15.5.6 Binary Search Tree(BST)

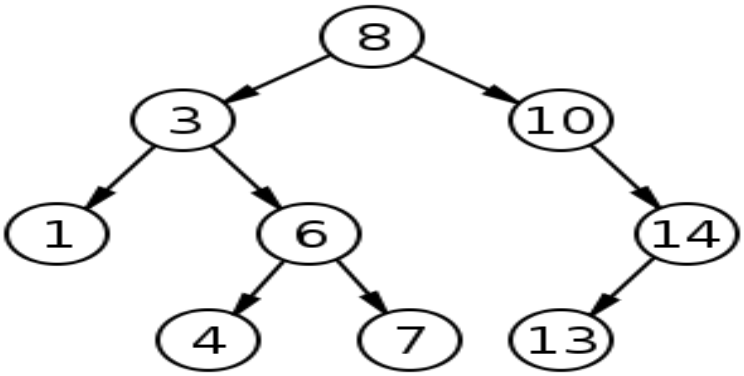
For a binary tree to beome a binary search tree, the data of all the nodes in the left side(sub-tree) of the root node should be the data of the root. The data of all the nodes in the right side(sub-tree) of the root node should be the data of the root.(HackerEarth, 2019)

1. The left node's value is always smaller than its parent's value.

2. The right node's value is always greater than its parent's value.

3. A BST is considered to be balanced if all levels of the tree is fully filled with the exception of the last level. On the last level, the tree is filled from left side to right side.

4. A Perfect BST is one in which it is both full and complete (all child nodes are on the same level and each node has a left and a right child node).(Turney, 2018)



15.6 Reference

1.Abhiraj Smit. (2016). Overview of Data Structures — Set 1 (Linear Data Structures) - GeeksforGeeks.[online] Available at:https://www.geeksforgeeks.org/overview-of-data-structures-set-1-linear-data-structures[Accessed 5 Apr. 2019].

2.Codeandwork.github.io. (n.d.). Linear data structures. [online] Available at: https://codeandwork.github.io

/[Accessed 5 Apr. 2019]courses/java/linearDataStructures.html [Accessed 8 Apr. 2019].

3.Miller, B. and Ranum, D. (2013). 6.2. Examples of Trees Problem Solving with Algorithms and Data Structures.[online] Interactivepython.org.Available at: http://interactivepython.org/runestone/static/pythonds/Trees/ExamplesofTrees.html [Accessed 8 Apr. 2019].

4.Datastructuresnotes.blogspot.com. (2009). What is a non-linear datastructure?. [online] Available at: http://datastructuresnotes.blogspot.com/2009/03/what-is-non-linear-datastructure.html [Accessed 7 Apr. 2019].

5.Katiyar, S. (2018). Binary Tree (Array implementation) - GeeksforGeeks. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/binary-tree-array-implementation/ [Accessed 5 Apr. 2019].

6.Tech Differences. (2018). Difference between Linear and Non-linear Data Structure (with Comparison Chart) - Tech differences. [online] Available at: https://techdifferences.com/difference-between-linear-and-non-linear-data-structure.html [Accessed 8 Apr. 2019].

7.Betances, M. (2018). Data Structures: Traversing Trees. [online] Medium. Available at: https://medium.com

/quick-code/data-structures-traversing-trees-9473f6d9f4ef [Accessed 8 Apr. 2019].

8.Wentworth, P., Elkner, J., Downey, A. and Meyers, C. (2012). 27. Trees How to Think Like a Computer Scientist: Learning with Python 3. [online] Openbookproject.net. Available at: http://openbookproject.net

/thinkcs/python/english3e/trees.html [Accessed 6 Apr. 2019].

9.IIT GUWAHATI (2009). NPTEL :: Computer Science and Engineering - Data Structures and Program Methodology. [online] Available at: https://nptel.ac.in/courses/106103069/51 [Accessed 5 Apr. 2019].

10.Enaganti, R. (2013). Construct Complete Binary Tree from its Linked List Representation - GeeksforGeeks. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/given-linked-list-representation-of-complete-tree-convert-it-to-linked-representation/ [Accessed 8 Apr. 2019].

11.HackerEarth. (2019). Binary Search Tree Tutorials and Notes — Data Structures — HackerEarth. [online] Available at: https://www.hackerearth.com/zh/practice/data-structures/trees/binary-search-tree/tutorial/ [Accessed 7 Apr. 2019].

12.Turney, K. (2018). Data Structures 101: Binary Search Tree. [online] freeCodeCamp.org. Available at: https://medium.freecodecamp.org/data-structures-101-binary-search-tree-398267b6bff0 [Accessed 5 Apr. 2019].