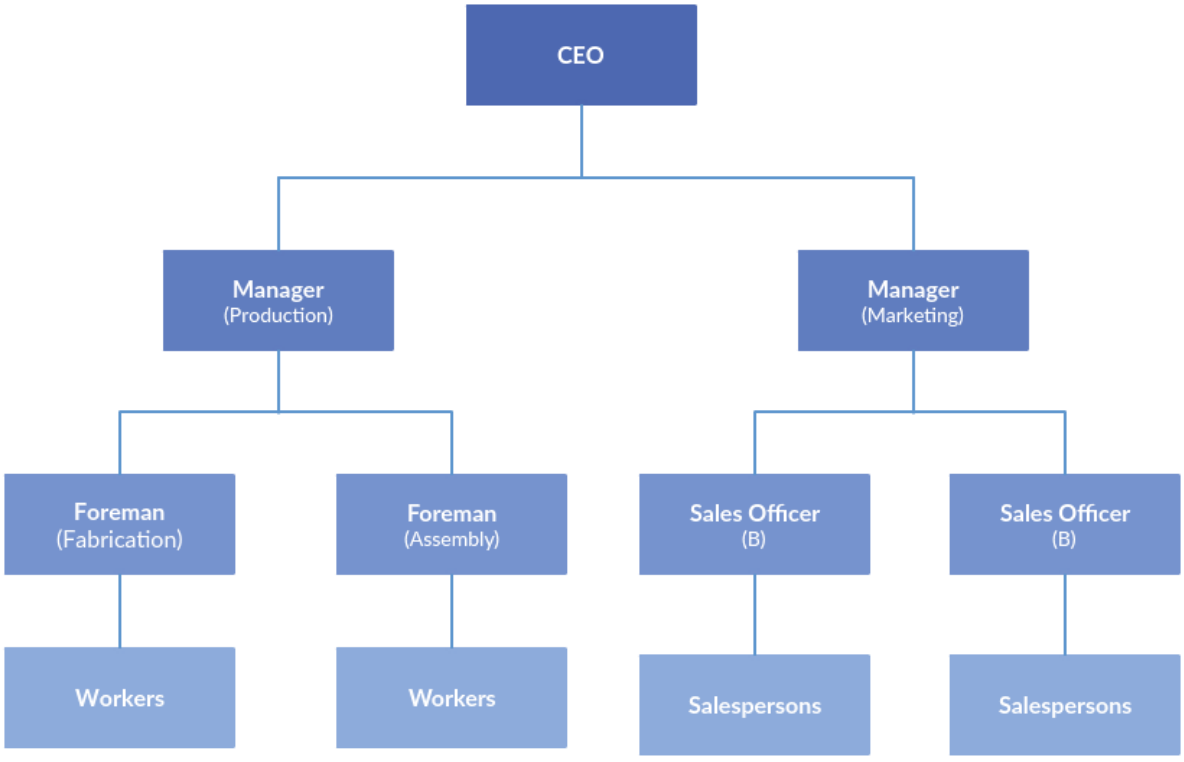
Lecture 15: Trees

Lecturer: Dr. Andrew Hines Scribes: Nontyatyambo Dazana, Arnaud Bellanger

I. Intro

In opposition to all the other data structure we have studied until now trees are the first non-linear data structure, the data is organized hierarchically.

A good example of a tree is the organization of a company. The root of the tree is unique example the CEO. The CEO is in charge of division managers that are themselves in charge of lower manager or other staff member. This imply that each member of the organisation has only one supervisor with the exception of the CEO (the root).



A Tree example: Organisational Structure

Another example is the class system of Java. Each class can inherit from only one class, a class can have multiple inheritors. All the class have the Object class as an ancestor and Object don't have ancestor Object is the root of the class tree of Java.

A tree that contain loop (example House of Troy in the lecture) is not considered as a tree data structure but as a graph.

II. Tree definitions

A tree is an abstract data type that stores elements hierarchically. With the exception of the top element, each element in a tree has a parent element and zero or more children elements.

Like in linked list each element of the tree can be considered as a node.

As stated above the top of the tree is different than the other nodes. It is called the **root** of the tree; the root doesn't have parent or **ancestor** element but can have children. The root is unique in a tree.

-A tree without root is empty.

-If the root has no children the tree has only one element.

III. Tree terminology

The **root** is linked to its children or **descendant** with **edges**, and subsequently each node is linked to its children with other edges. The **path** is the sequence of edges leading from a node A to a node B.

Two nodes are considered **Sibling** if and only if they share the same parent node.

A node is considered as a **leaf or external** if it has no children.

A node that has children is considered as internal.

A **subtree** of a tree can be defined as a tree within a tree, it has all the property of a tree but is root as an ancestor.

The **size of a tree** consists of the number of nodes.

The **depth** of a node in a tree is the number of links on the path from it to the root.

-The depth of the root is 0.

-The depth to a child of the root is 1.

The depth of p can also be recursively defined as follows:

• If p is the root, then the depth of p is 0.

• Otherwise, the depth of p is one plus the depth of the parent of p

The **height** of a tree is the maximum depth among its nodes.

Parent, any node except the root node has one edge upward to a node called parent.

Child is the node below a given node connected by its edge downward.

Visiting refers to checking the value of a node when control is on the node.

Path refers to sequence of nodes and edges connecting a node with a descendant.

Traversing means passing through nodes in a specific order.

Level of a node is defined as: 1 + the number of edges between the node and the root.

Keys represent values of nodes based on which a search operation is to be carried out.

IV. Tree ADT

Operations on trees, just like other data structures, include:-

Basic ADT method:

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.

More ADT method:

Is_root(n): return True is n is the root of the tree.

Parent(n): return the position of the parent of n or none is n is the root.

Num_children(n): return the number of children of n. 0 is n is a leaf.

Is_leaf(n): return true if n is a leaf of the tree.

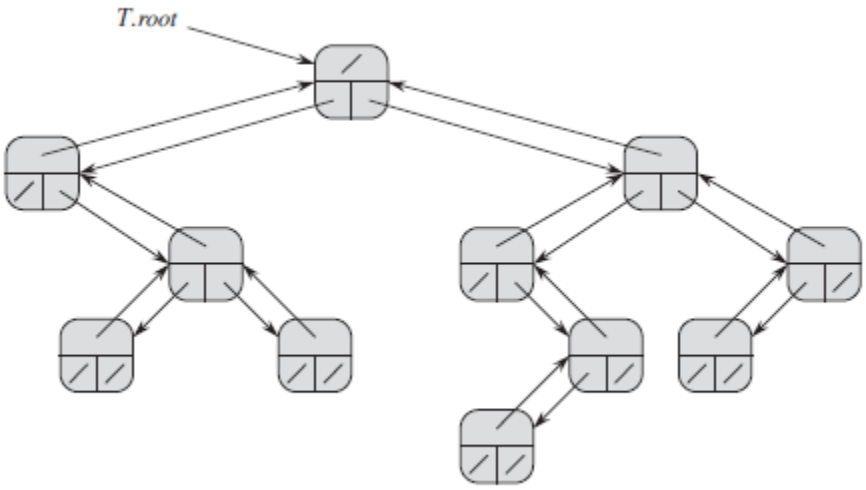
V. Tree representation

A. Linked list representation

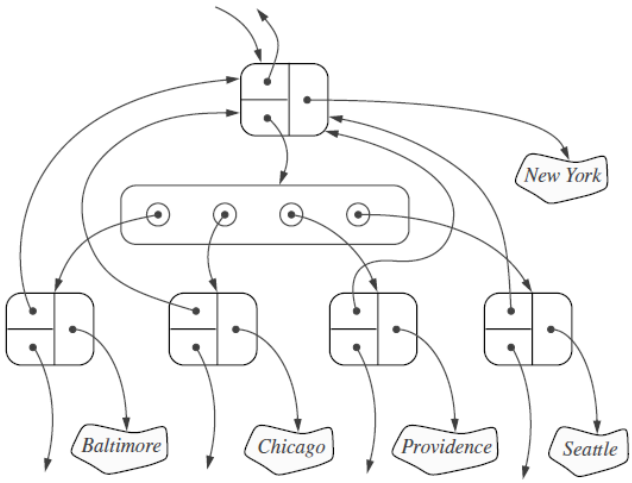
There is two kind of linked list tree representation.

1. Classic representation

Each node of a tree using the list link representation store the address of it's ancestor and the address of all its children.



If the tree is binary this if fine because each node stores its data and 3 addresses (parent, left child, right child). But if the tree is not binary (binary mean no more than 2 children by node) each node will have to store a consequent amount of information.



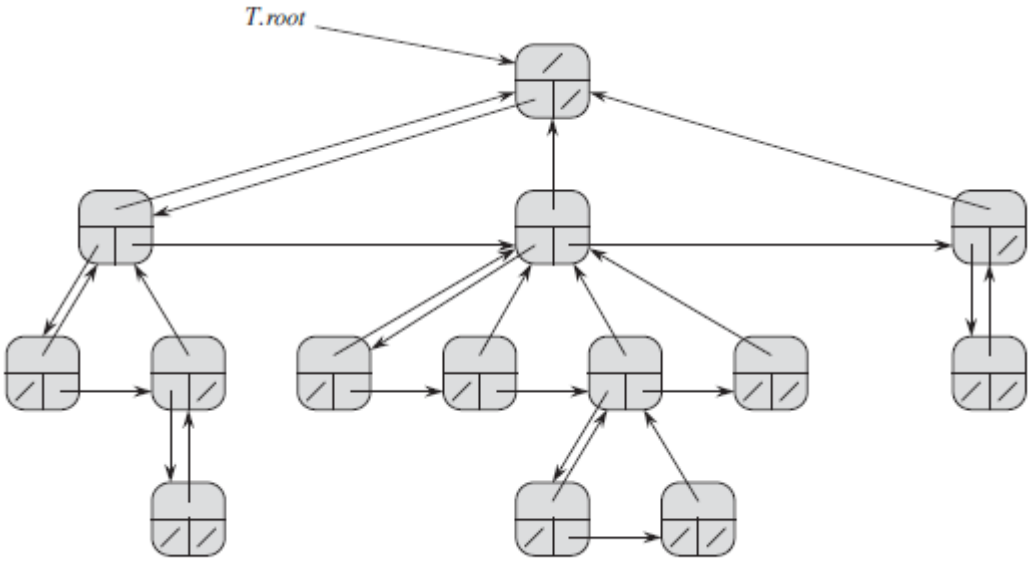
Here New York needs to store the address of Baltimore Chicago, providence and Seattle.

To solve this problem, we can use the left-child right-child representation.

2. Left-child right-child representation

In this representation each node stores its data, the address of the ancestor, the address of its left child and the address of its right sibling. If a node has more than one child, we can access directly the left child and the other children will be accessed by using the address stored in the left child.

Like represented here:



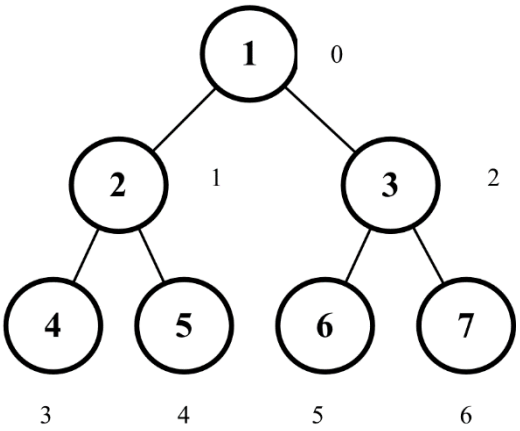
B. Array-based representation (Binary trees)

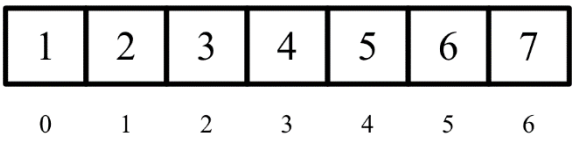
Array can also be used to represent a tree. Array-based representation can be a normal binary tree or a structured binary search tree. The difference between the two is that, for a normal binary tree there is no order on the value of the nodes whereas in binary search tree (which also inherits the properties of a binary tree), the node with value smaller than the parent node must become the left child and the node with value greater than or equal to the parent node must become the right child. The array method is using an array accessing the correct indexes to retrieve the values of the parent, left, and right children. This is done by laying out the nodes of the tree in breadth-first order in the array. In this way, the root is stored at position 0, the root's left child is stored at position 1, the root's right child at position 2, the left child of the left child of the root is stored at position 3, and so on.

We can define the position of every other node in the tree recursively:

- If root position is *i* then,
- The left child is 2**i* +1
- The right child is 2**i* +2
- If current node is *i* then its parent is *i*/2

The following diagram demonstrates the binary tree's different representations:





C. Array vs Linked list representation of a tree

Linked lists operations can be slower due to pointer manipulation and use less space if the tree is unbalanced but rotation code is simple.

Arrays on the other hand have faster operations and use less space if the tree is unbalanced but the rotation code is complex.

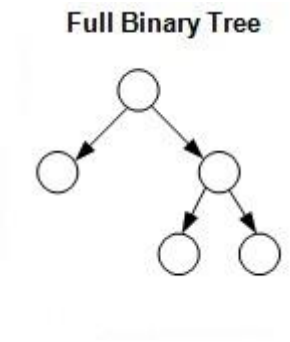
The choice of which method to use depends on what has to be done to a tree as each method can be good for one operation but bad for another.

D. The Efficiency of Array based trees (Binary trees)

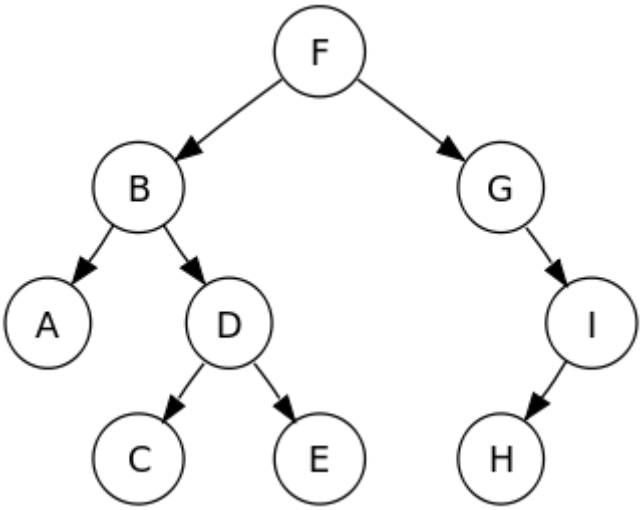
Operation	**Average case**	**Worst case**
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(log n)	O(n)

E. Binary Tree types

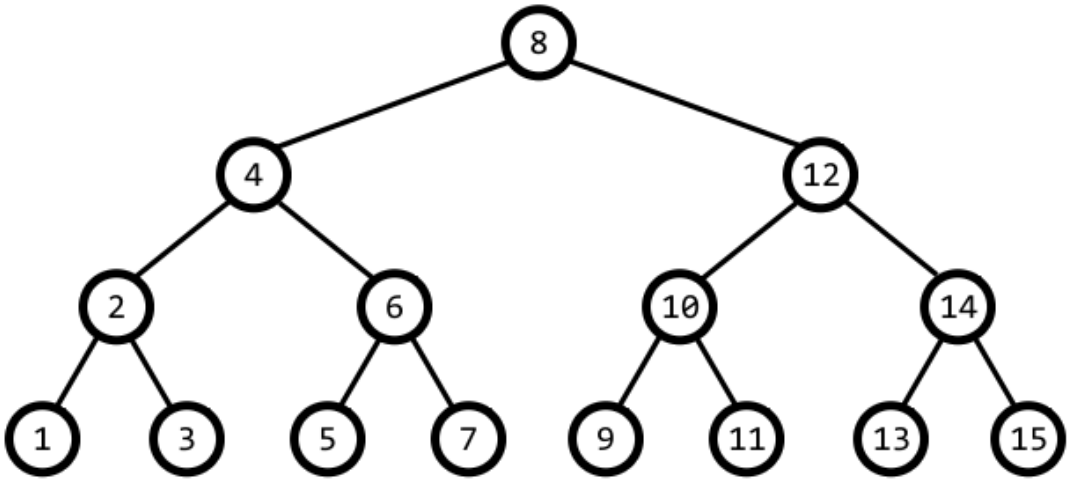
Full Binary Tree: A Binary Tree is full if every node has 0 or 2 children. We can also say a full binary tree is a binary tree in which all nodes except leaves have two children.



Complete Binary Tree: A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible.



Perfect Binary Tree: A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at the same level.



Balanced Binary Tree: A binary tree is balanced if the height of the tree is O(Log n) where n is the number of nodes. For Example, AVL tree maintains O(Log n) height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths are same and there are no

adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide O(log n) time for search, insert and delete.

VI. Traversing a tree

A. Depth first search (DFS)

A DFS will traverse the tree by always visiting a child of a node instead of visiting a sibling.

Simply put DFS will do: If the node has a child visit the child else visit a sibling.

Recursive DFS :

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

Iterative DFS:

```
Algorithm dfs:
Input:  Tree t and node n
Output:  the function explores every node from n
to_visit ← empty stack
add n to to_visit
while to_visit is not empty do
    current ← pop to_visit # get the first element
    push all children of current to to_visit
    do something on current
endfor
```

B. Breadth First Search (BFS)

The BFS will visit each node of a tree by visiting in priority the sibling of a node instead of it's children.

Recursive BFS:

```
Algorithm bfs:
Input:  queue q (originally having the root of the
tree)
Output:  explores every node of t rooted at n
if q is empty then # base case
    do something (?)
else
    current ← dequeue q
    for each child n_c of n do
        enqueue n_c
    endfor
    do something
    bfs(q)
endif
```

BFS non recursive solution:

```
Algorithm bfs:
Input:  Tree t and node n (root or not)
Output:  explores every node of t rooted at n
to_visit is a queue
enqueue n
while to_visit is not empty do
    n_current ← dequeue to_visit
    for each child n_c of n_current do
        enqueue n_c to to_visit
    endfor
    do something on n_current
endwhile
```

VII. Conclusion

Searching is one of the most important operations in computer science. Of the many search data structures that have been designed and are used in practice, search trees, more specifically balanced binary search trees, occupy a coveted place because of their broad applicability to many different sorts of problems.

A tree data structure, along with graphs, are two non-linear data structure that store data in a non-common but specific way (compared to linear structures collections such as arrays). Binary search trees come in many shapes. The shape of the tree determines the efficiency of its operations. The height of a binary search tree with n nodes can range from a minimum of Olog2(n + 1) to a maximum of n. Binary trees are a very powerful data structure and if done right, handling large amounts of sorted data becomes easier and quicker.

VIII. Bibliography

https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm last accessed on 03/04/19

https://adrianmejia.com/blog/2018/06/11/data-structures-for-beginners-trees-binary-search-tree-tutorial/ last accessed on 03/04/19

https://www.sqa.org.uk/e-learning/LinkedDS04CD/page_02.htm last accessed on 07/04/19

http://freeusermanuals.com/backend/web/manuals/1521099422ADT_Binary_Trees.pdf

last accessed on 07/04/19

http://www.bowdoin.edu/~ltoma/teaching/cs210/spring09/Slides/210-Trees.pdf last accessed on 07/04/19

Goodrich, M., Tamassia, R. and Goldwasser, M. (2013). *Data Structures & Algorithms in Python*. 2nd ed. pp.300 - 341.