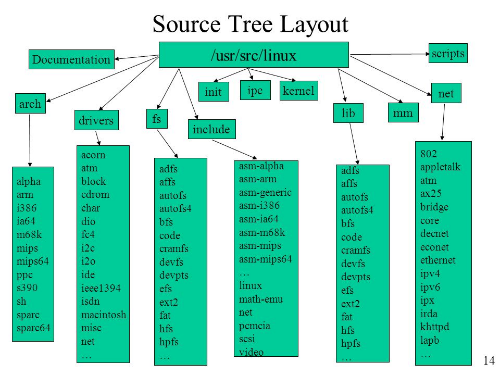
**Data Structures Search Trees**

**1.0 Basics of Search Trees**

* Search Trees: are trees used to search for data within a tree like structure. The most basic type of a tree is the binary search trees. In cases when the run time is not satisfactory, we can look at other types of trees (binary self-balancing trees) like Red-Black Search Tree or AVL.
* Trees are some of the most used data structures in computer science. They link information together in a relational way. You can use them to sort, to store information, to make inferences, and so much more. They are even used in databases to help index (find) the data in the database.
* A tree data structure can be defined recursively as a collection of nodes, where each node is a data structure consisting of a value and a list of references to nodes. Recursively meaning when a thing is defined by the means of itself
* Trees are known as a hierarchical data structure. Instead of pointing from one piece of data to the next, a tree has many different paths that you can take from each node. This creates a system of parents, and children.
* A tree consists of a root node and various sub-nodes branching from the root node. The sub-nodes may also have many other sub-nodes and so on. The root node is on top, and all the other nodes branch out of that node. This is like Linux’s root user that contains all the directories.
* A Tree is a basically a list of different data pointing out to their children and parents.
* You also have nodes called “leaves”. These are the nodes without any children, meaning they are at the end of the tree structure. Once you reach them, you cannot go any further in the tree. Leaves is plural for leaf.
* A Tree data structure allows the user to get to the data much quicker than a linked list. Because a linked list would need to go through every node prior to the node you wish to go too. However, a tree has a specific structure that it follows to get to the data node that you wish to go too. Exactly like the root user situation in Linux kernel, which is why it is must faster search for any file in Linux



**2.0 Binary Search Trees (BST)**

* Binary Tree: A tree with two options (left & right) or a single option, or no options at all
* Search Tree: The method that this tree is being used for
* BST: A tree of up to two options (left & right) and is being used for searching. Anything to the right of the main node is larger than the main node. Anything to the left of the main node is smaller than the main node
* A binary search tree is a tree with these rules:
  + Each node can only have at **most** two children
  + All right children must be greater than
  + All left children must be less than or equal to. (You can put the equal to on the right or left)
* BST Exp1:
  + Notes in regards of Exp1
    - There are 4 layers. Therefore, for the example below the average run time will be 4 layers. Even though we have 9 nodes. Therefore, the run time of BST tends to be O(log(n)). This is because every time we perform a search, we cut the tree in half
    - The run time won’t be O(log(n)) when the tree looks like **a linked list**.
      * In cases when the run time is not satisfactory, we can look at other types of trees (binary self-balancing trees) like Red-Black Search Tree or AVL.
  + Exp1 Explained:

|  |  |
| --- | --- |
| * The 8 is the root node * The left of 8 is 3 which smaller than the root node * The right of 8 is 10 which is larger than the root node * There are 4 layers of trees. Therefore, for this specific tree the longest run time will be 4 layers. Even if though we have 9 nodes. Therefore, the run time of BST tends to be log(n) except when the tree looks like a linked list. * In cases when the run time is not satisfactory, we can look at other types of trees (binary self-balancing trees) like Red-Black Search Tree or AVL. * Node 1, Node 4, Node 7, and node 14 are leaves because they are nodes without children. Leaves is the plural of “leaf" |  |
| * This is layer 0 with the root node |  |
| * This is layer 2 |  |
| * This is layer 3 |  |
| * This is layer 4 |  |

**3.0 Binary Search Trees (BST) Run Times**

* The run time of BST is in the first table below. An example of a worst case BTS in the second table. Finally, the run time for BTS and other algorithms is presented in the third table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BST Operation | Average Run Time | Comment | Worst Run Time | Comment |
| Search (Searches for a node) | O(log(n)) | Each decision we make during the search we will cut the tree in half | O(n) | This is the case when your BST looks more like a linked list. Because searching search decision we look at a single node |
| Insert (Finds the place where the node should go) | O(log(n)) | To find where a new node would go we would be cutting everything in half | O(n) | This is the case when your BST looks more like a linked list. Because searching search decision we look at a single node. Therefore we would need to search the while tree |
| Delete  (Removal of a node) | O(log(n)) | To find where we going to delete a node we would be cutting everything in half | O(n) | This is the case when your BST looks more like a linked list. Because searching search decision we look at a single node. Therefore, we would need to search the while tree |

|  |  |
| --- | --- |
| * Draw backs of BST, during the special case where every proceeding child node is greater than the parent node. * There are 5 layers and 6 nodes. If this exact format continues. For each new node we will have a new layer * This is an issue with BST, because in this special case the run time will change from be O(log(n)) to O(n) | Flatten Binary Tree to Linked List - LeetCode |
| * This is layer 0 with the root node | Flatten Binary Tree to Linked List - LeetCode |
| * This is layer 1 | Flatten Binary Tree to Linked List - LeetCode |
| * This is layer 2 | Flatten Binary Tree to Linked List - LeetCode |
| * This is layer 3 | Flatten Binary Tree to Linked List - LeetCode |
| * This is layer 4 | Flatten Binary Tree to Linked List - LeetCode |
| * This is layer 5 | Flatten Binary Tree to Linked List - LeetCode |

|  |  |  |
| --- | --- | --- |
| Trees Run Times | | |
| Table  Description automatically generated | Graphical user interface  Description automatically generated | Table  Description automatically generated with medium confidence |
| **Note**: The **space** run time complexity of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of characteristics of the input. It is the memory required by an algorithm until it executes completely. All three algorithms here have a space run time of O(n) | | |

**3.0 Tree Traversals “go through”**

* Tree Traversal: is the act of printing a tree as a string
* There are three different ways of Tree Traversing. At every step. You must apply the order until you can not apply it as each step. At each step you must explore all of the order then collect the data. This seems easy. However, it requires practice. If you ever need watch the video again
  + 1. In order traversal: Left, Root, Right (An ascending order of nodes)
  + 2. Pre order traversal: Root, Left, Right
  + 3. Post order traversal: Left, Right, Root
  + 4. Level order( Takes the levels(Top Down)or (Bottom Up)) & goes left to right
    - This order is difficult to convert back to a Tree from a string
* Exp1 :



* + (a) Inorder (Left, Root, Right) : 4 2 5 1 3
  + (b) Preorder (Root, Left, Right) : 1 2 4 5 3
  + (c) Postorder (Left, Right, Root) : 4 5 2 3 1
  + (d). Level order( Takes the levels(Top Down)or (Bottom Up)):
    - (Top Down): 1,2,3,4,5
    - (Bottom Up) 4,5,2,3,1
* Exp2

