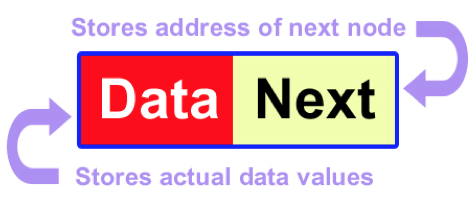
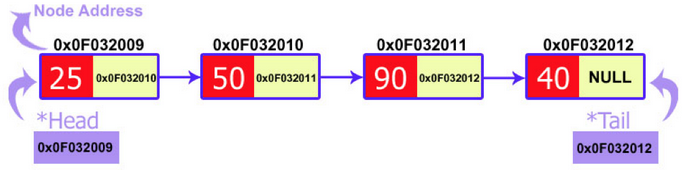
**Data Structure Notes**

**Linked-List:**

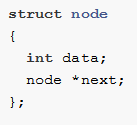
* **Introduction:**
  + A linked list is a data structure that can store an indefinite amount of items. These items are connected using pointers in a sequential manner.
  + A visual representation of a single node and the complete data structure can be seen below:



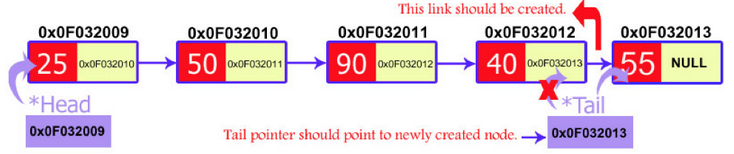


**Figure 1: Visual Representation of a Singly Linked List**

* + Each element in a linked list is called a node. A node contains some data and a single pointer or multiple pointers. In a singly linked list the node only contains one pointer, which points to the memory address of the next node in the linked list. For a doubly linked list the node contains a pointer to the next node in the list and the previous node in the list.
  + There are multiple types of linked lists:
    - Singly Linked List
    - Circular Linked List
    - Doubly Linked List
* **Linked Lists vs. Arrays:**
  + The size of arrays is fixed, and the memory is pre allocated.
  + Inserting a new element into an array when it is full, requires that a new array of a larger size is created and we insert each element of the old array into the new array. This process is costly in both time and space for arrays.
  + Advantages of Linked Lists over arrays:
    - Dynamic Size
    - Ease of Insertion/Deletion
  + Drawbacks:
    - We have to access each element sequentially with linked lists, which takes O(n) in the worst case, whereas with arrays it is constant time look up.
    - Linked list requires more space than arrays. The reason being that each node of the Linked List stores not just the data but also the memory address of the next node in the list.
* **Singly Linked List Implementation:**
  + The main difference between singly linked lists and doubly linked lists is the structure of the node. The node for a singly linked list will point only to the next memory address, whereas the doubly linked list will point to the next and previous memory address of the node.
  + The structure of the node for a singly linked list can be seen below:

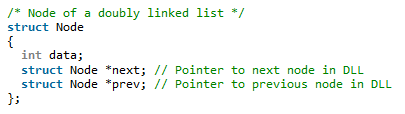


* + Linked List Operations:
    - Checking whether the list is empty.
    - Appending to a node.
    - Prepending to a node.
    - Inserting a node.
    - Deleting a node (either by index or data value).
    - Reversing a linked list.
    - Traversing a linked list.
  + Adding a node to the list: Appending a new node to the end of a linked list has 2 steps:
    - First we need to link the address of the new node, to the next\_ member of the current tail node.
    - The second step is to then update the tail so that it is the new node is now the tail of the linked list.



**Figure 2: Visual Representation of appending a new node to the end of the list**

* + Insertion: Inserting a new node into a linked list has three specific cases: inserting at the start, inserting at the end (which we over in the previous section), and inserting at a particular position in the list.
    - Insertion at the start:
      * We first need to set the next\_ field of the new node to the head of the linked list.
      * Then set the head of the link list equal to the new node.
    - Insertion at a specified position:
      * We pass the address of the new node in the next\_ field of the previous node.
      * We pass the address of the current node in the next\_ field of the new node.
  + Deletion: Similar to insertion, deletion has three specific use cases: deletion at the start, at the end, and at a specified position.
    - Deletion at the start:
      * Create a temporary node and pass it the address of the head node.
      * Set the second node equal to the head.
      * Delete the temporary node.
    - Deletion at the end:
      * Iterate to the second to last node
      * Set a temp node equal to the tail and delete it.
      * Set the tail equal to the second to last node.
      * Set the new tail’s next pointer to null
    - Deletion at a specific point: We use two pointers to iterate through the linked list.
      * As we iterate through the list we have the current node and previous node.
      * Once we come to the specified position, we set previous node’s next field equal to the current node’s next field and then delete the current node.
* **Doubly Linked List Implementation:**
  + The structure of the node for a doubly linked list can be seen below:

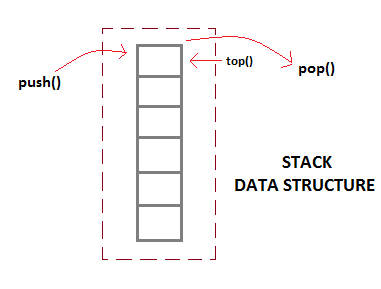


* + The operations are very similar to the singly linked list but we have more flexibility. Since we are not restricted to only iterating through the list by starting at the head of the list. We can start at the tail of the list and iterate to the head.
* **References:**

1. <https://www.codementor.io/codementorteam/a-comprehensive-guide-to-implementation-of-singly-linked-list-using-c_plus_plus-ondlm5azr>
2. <https://www.geeksforgeeks.org/data-structures/linked-list/>
3. <https://www.cs.cmu.edu/~ab/15-123S09/lectures/Lecture%2010%20-%20%20Linked%20List%20Operations.pdf>
4. <https://www.programiz.com/dsa/linked-list-operations>

**Stack:**

* **Introduction:**
  + A Stack is an abstract data type. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, that element goes to the top of the stack. The only element that can be removed is the element that is at the top of the stack. (word for word from reference 2).



**Figure 3: Visual Representation of Stack Data Structure**

* + The order in which the collection of items is stored is said to be LIFO (Last In First Out) or FILO (First In Last Out).
  + There are a few **basic operations**, which are performed by the stack data structure:
    - Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
    - Pop: Removes an item from the stack, in reverse order. Meaning the last item pushed onto the stack is the first item to come off when pop is called. If the stack is empty and we try to pop, this is said to be an Underflow condition.
    - Peek or Top: Returns the top element of the stack, without removing it from the stack.
    - isEmpty: Checks if the stack is empty.
* **Implementation of Stack Data Structure:**
  + Two ways to implement a stack:
    - Array – The are quick but limited in size.
    - Linked List – requires overhead to allocate, link, unlink, and deallocate, but is not limited in size.
* **Applications:**
  + Parsing
  + Expression Conversion (Infix to Postfix, Postfix to Prefix etc)
  + Balancing of symbols
  + Redo-undo features at many places like editors, photoshop
  + Forward and backward feature in web browsers
  + Used in many algorithms like: Tower of Hanoi, tree traversals, stock span problem, histogram problem.
* **References:**

1. <https://www.geeksforgeeks.org/stack-data-structure-introduction-program/>
2. <https://www.studytonight.com/data-structures/stack-data-structure>

**Queue:**

* **Introduction:**
  + Similar to a stack, a queue is a linear data structure. The main difference between a queue and a stack is how the elements are stored. A stack object is defined as a LIFO (Last In First Out) data structure, whereas a Queue object is a **FIFO** (First In First Out) data structure.
  + There are two main operations performed on queues:
    - Enqueue – the process of adding an element into a queue.
    - Dequeue – the process of removing an element from a queue.
* **Operations:**
  + There are four basic operations performed on queues:
    - Enqueue – Add an element into a queue.
    - Dequeue – Remove an element from a queue.
    - Front – Returns the value at the front of the queue.
    - Rear – Returns the value at the end of the queue.
* **Implementations:**
  + The underlying implementations of queues can vary, as long as the operations of the queue stay the same. The most common implementations of queues are arrays, linked lists and stacks.
    - Array Implementation: In order to implement a queue using an array implementation, two indices must be known, the front and the rear. We enqueue an element at the rear and dequeue an element at the front. This highlights the FIFO structure. However, arrays are fixed size structures, which causes problems when we reach the end of the array. The solution to this problem is a **circular queue (5)**.
    - Linked List Implementation: Very similar to the array implementation. However, we don’t have to worry about the circular queue issue, because we are no longer dealing with a fixed size structure.
    - Stack Implementation: Another implementation of queues is with multiple stacks. In order to perform a Enqueue we only need one stack that we can push data onto. For a Dequeue we need two stacks. For more specific implementation details look at reference (6).

* **Applications:**
  + Multiple user requests of a single shared resources. (printer, CPU task scheduler, Disk Scheduling)
  + IO Buffers, pipes, file IO, etc.
  + Breadth-First Searches
* **References:**

1) <https://www.geeksforgeeks.org/queue-data-structure/#intro>

2) <https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html>

3) <https://www.studytonight.com/data-structures/queue-data-structure>

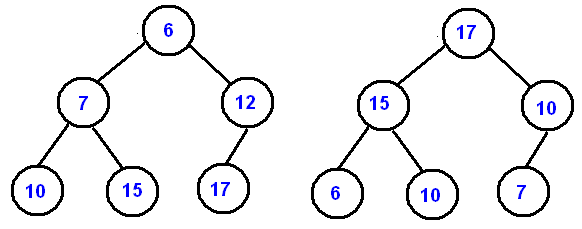
4) <https://introcs.cs.princeton.edu/java/43stack/>

5) <https://www.studytonight.com/data-structures/circular-queue>

6) <https://www.studytonight.com/data-structures/queue-using-stack>

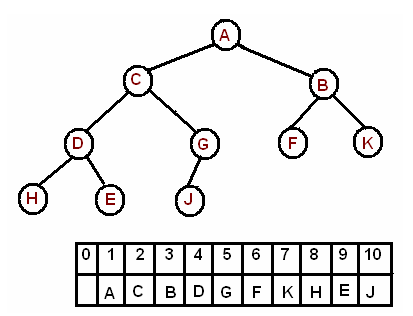
**Binary (MIN/MAX) Heaps:**

* **Introduction:**
  + A binary heap is a complete binary tree which satisfies the heap ordering property. The ordering can be one of two types:
    - The *min-heap* property: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
    - The *max-heap* property: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.



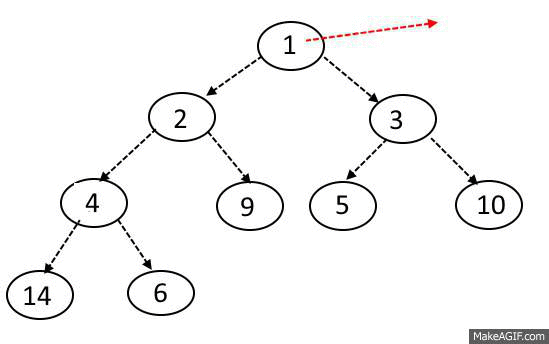
**Figure 4: Example of a Min Binary Heap (left) and a Max Binary Heap (right)**

* **Array Implementation:**
  + We can uniquely represent a complete binary tree with an array. The level order traversal of the tree is stored in the array.
  + Some implementations include leaving the first element of the array open and storing the root element as the second item. Others have the root value as the first element of the array. Either way we can visualize the storing of a binary tree in the examples below.



**Figure 5: Level-Order Binary Tree Array Storing**

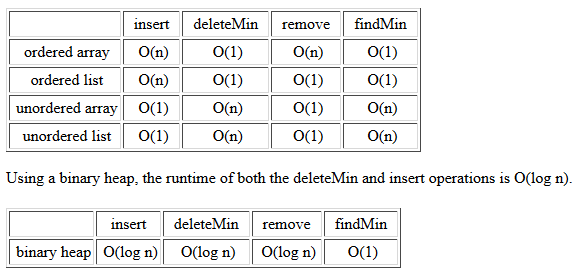
* + Index from 1:
    - Arr[i/2]: returns the **parent** node of index i.
    - Arr[(2\*i)]: returns the **left** child node of index i.
    - Arr[(2\*i) + 1]: returns the **right** child node of index i.
  + Index from 0:
    - Arr[(i – 1)/2]: returns the **parent** node of index i.
    - Arr[(2\*i) + 1]: returns the **left** child node of index i.
    - Arr[(2\*i) + 2]: returns the **right** child node of index i.
* **Insert:**
  + When we insert a new value into the heap, we put that value at the end of the array. Then we compare that value with its parent, if it is larger (for max heap) or smaller (for min heap) than the parent’s value we swap the nodes. If not, we leave it.
* **Extract Minimum:**
  + To Extract/Delete the minimum value from the heap we swap the min with the last element of the array and discard the last element. We have to make sure that we restore the property of the heap, which is known as heapify. We do this with the sink-down method (5).
    - If the parent node is larger than either of the child nodes. Compare the child nodes, which every of the two nodes are smaller swap with the parent node.
    - Proceed down the tree until the parent node is smaller than the child node.

**[](file:///C:\Users\pmollica\AppData\Roaming\Microsoft\Word\Delete-OR-Extract-Min-from-Heap.gif)**

**GIF 1: Press ctr + click to see the animation.**

* **Delete A Node:**
  + Find the index for the element to be deleted.
  + Take out the last element from the last level from the heap and replace the index with this element.
  + Perform the Sink-Dow method.
* **Comparative Time and Space Complexity:**





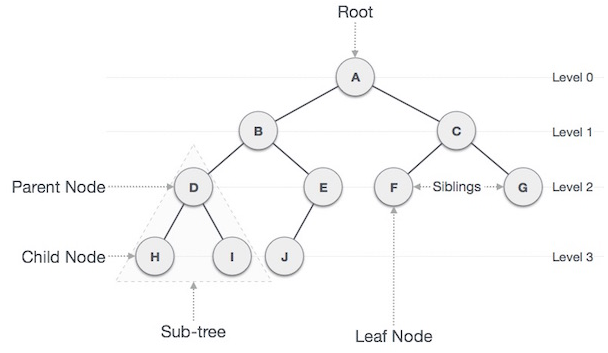
**Figure 6: Time Complexity Comparisons**

* **Applications:**
  + Priority Queue
  + Heap Sort
  + Graph Algorithms: The priority queues are especially used in Graph Algorithms like Dijkstra’s Shortest Pat and Prim’s Minimum Spanning Tree.
  + K’th Largest Element in an array
  + Sort an almost sorted array.
  + Merge K sorted arrays.
* **References:**

1. [**https://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html**](https://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html)
2. [**https://www.geeksforgeeks.org/binary-heap/**](https://www.geeksforgeeks.org/binary-heap/)
3. [**https://www.youtube.com/watch?v=WCm3TqScBM8**](https://www.youtube.com/watch?v=WCm3TqScBM8)
4. [**https://algorithms.tutorialhorizon.com/binary-min-max-heap/**](https://algorithms.tutorialhorizon.com/binary-min-max-heap/)

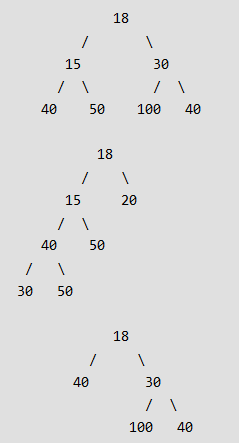
**Binary Trees:**

* **Introduction:**
  + A tree is a hierarchical or a non-linear data structure. Every tree contains nodes, these nodes store data and have pointers to other nodes called leaves. We have a topmost node called the root node. From the root node we work are way down the tree with child nodes. We can visualize the tree data structure below:



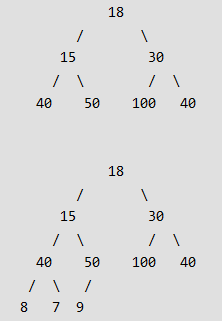
**Figure 7: Visualization of a tree data structure. (6)**

* + Some important terminology associated with tree structures are the following:
    - *Root*: The topmost node of a tree.
    - *Parent*: Any node that contains child nodes, and isn’t the root node.
    - *Child*: A node that is connected below another node.
    - *Leaf*: A node that does not have a child node.
    - *Edge*: The connection between two nodes.
    - *Height*: The height of a tree is the length of the tree, is the number of nodes from the root to the bottommost leaf – 1. E.g. the height of the tree in figure 7 is 3.
    - *Depth*: This is associated with a node. So the depth of a specific node is the length from that node to the root. E.g. in figure 7. The depth of node C is 1 and the depth of node H is 3.
    - *Subtree*: A nice property of trees is that each child node has its own sub tree, or a subset of the whole tree.
  + There are many types of trees but one of the most common trees are Binary Trees, which will be the focus of this section. A binary tree is defined by the node, where every node is limited to at most 2 child nodes (typically named left and right).
* **Properties:**
  + Maximum number of nodes per level is equal to 2L, where L is the level. E.g. the Level of the root node is 0, 20 = 1.
  + Maximum number of nodes in a binary tree of height h is equal to 2h – 1.
  + In a binary tree, the number of leaf nodes is always one more than nodes with two children.
    - L = T + 1
      * L = Number of leaf nodes
      * T = Number of internal nodes with two children.
    - Proof can be shown in reference 2.
* **Types of Binary Trees:**
  + *Full Binary Tree*: A Binary tree is said to be full if every node in the tree has either 0 or 2 child nodes. Examples of full binary trees are seen below:



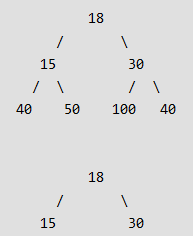
**Figure 8: Examples of Full Binary Trees.**

* + *Complete Binary Tree*: A binary tree is said to be complete if all levels of the tree are completely filled, with the possible exception of the last level. Examples of complete binary trees can be seen below:



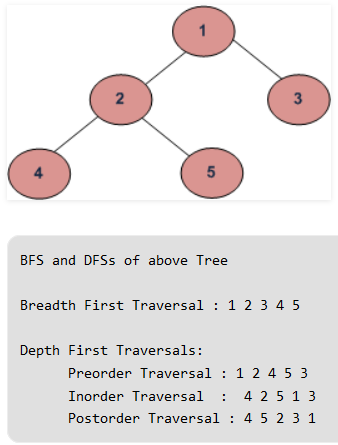
**Figure 9: Examples of Complete Binary Tree**

* + *Perfect Binary Tree*: A Binary tree is said to be a perfect when all internal nodes have 2 child nodes and all the leaves are at the same level. Examples of a perfect binary tree can be seen below:



**Figure 10: Examples of Perfect Binary Tree**

* + *Balanced Binary Tree*: A binary tree is said to be balanced if the height of the tree is O(Log *n*), where *n* is the number of nodes. Different types of trees, such as AVL and Red-Black trees maintain this property differently, which will be discussed in a later section.
  + *A Degenerate (or Pathological) Tree*: A tree where every internal node has only one child node. In terms of performance this tree is equivalent to a linked list.
* **Operations:** Below is a list of basic operations on binary trees.
  + Insert – Insert a node or an entire sub tree to a node.
  + Search – Search for specific items in a tree.
  + Traversal – There are four main methods of Tree traversals
    - Breadth First Traversal (or in order traversal)
    - Depth First Traversal:
      * Preorder:
      * Inorder:
      * Postorder:
* **Implementation:**
  + Tree Traversal: There are two categories of tree travers Breadth First Traversal or Depth First Traversal: (We can visualize the different traversal methods in figure 11)
    - Breadth First Traversal: This is also called level order traversal, where we start from the root node and traversal down level by level.
    - Depth First Traversal: In this category of traversal we work our way down to the bottom leafs first and traverse back up to the top. There are three different types of Depth First Traversals:
      * Inorder (Left-Root-Right)
      * Preorder (Root-Left-Right)
      * Postorder (Left-Right-Root)



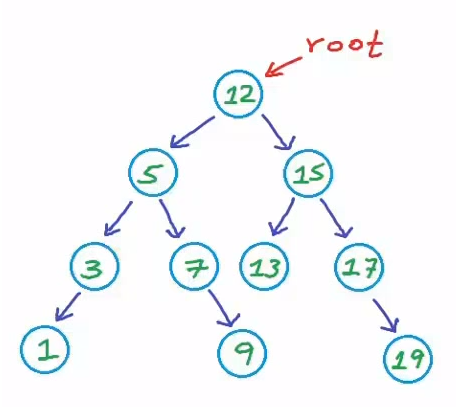
**Figure 11: Different Traversal Methods (7)**

* **Applications:** 
  + More efficient searching of data.
  + To represent hierarchical data.
* **References:**

1. <https://www.geeksforgeeks.org/binary-tree-set-1-introduction/>
2. <https://www.geeksforgeeks.org/binary-tree-set-2-properties/>
3. <https://leetcode.com/explore/learn/card/data-structure-tree/>
4. <https://medium.com/the-renaissance-developer/learning-tree-data-structure-27c6bb363051>
5. <https://en.wikipedia.org/wiki/Tree_(data_structure)>
6. <https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm>
7. <https://www.geeksforgeeks.org/bfs-vs-dfs-binary-tree/>
8. <https://en.wikipedia.org/wiki/Tree_traversal>

**Binary Search Trees:**

* **Introduction:**
  + Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value. There are only two child nodes per parent node in a binary search tree.
  + The properties of a binary search tree (BST) are the following:
    - The left sub tree will only contain key values less than the current node’s key value.
    - The right sub tree will only contain key values greater than the current node’s key value.
    - All subtrees are themselves binary search trees.
  + An Example of a BST can be seen in figure 12.



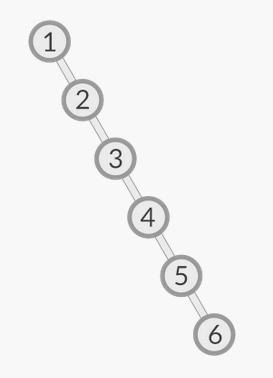
**Figure 12: Binary Search Tree Example**

* **Implementation:** (The implementations can typically be done with or without recursion)
  + Insertion: (Time Complexity: O(h), where h = height of tree)
    - Start from the root node:
    - Compare the value of the current node to the value of the root node.
    - If the value is less than or equal to the value of the root node, proceed to the left child node.
    - If the value is greater than the value of the root node, proceed to the right child node.
    - Repeat this comparison until you reach a null node. Which is where you will insert the new value.
  + Search: (Time Complexity: O(h), where h = height of tree)
    - Start from the root node and compare the key value with the value you are searching for. If they are equal return the root node.
    - If the value you are searching for is not equal to the value of the root node, compare that value to the root value.
    - If the value is less than move to the left child and node and compare the values.
    - If the value is greater than move to the right child node and compare the values.
    - Continue these comparisons until you find the value you are searching for.
    - If the value is not in the tree return a null pointer.
  + Delete: This is slightly more complex than the other operations, because there are multiple cases that need to be handled: (Time Complexity: O(h), where h = height of tree)
    - A node with no children (leaf node):
      * Delete node.
    - A node with just one child:
      * In this case we need to keep track of the parent node.
      * If the parent’s left child is the node we are looking to delete, we set the parents left child to the node’s child. Then delete the node.
      * If the parent’s right child is the node we are looking to delete, we set the parent’s right child to the node’s child. Then delete the node.
    - A node with two children:
      * In order to delete a node with two child nodes, we must first find the node that we will replace it with.
      * Finding the replacement node by finding the in order successor of the node. Which will be the left most child node after the right child node of the node we are trying to delete.
      * After we find the in order successor we replace that node’s data with the data of the node we are trying to delete.
      * If the in order successor has a right child node, we must attach that node to the left child of the parent node.
      * We then delete the in order successor node.
* **References:**

1. <https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm>
2. <https://medium.com/the-renaissance-developer/learning-tree-data-structure-27c6bb363051>
3. <https://www.geeksforgeeks.org/inorder-successor-in-binary-search-tree/>
4. <https://helloacm.com/how-to-delete-a-node-from-a-binary-search-tree/>

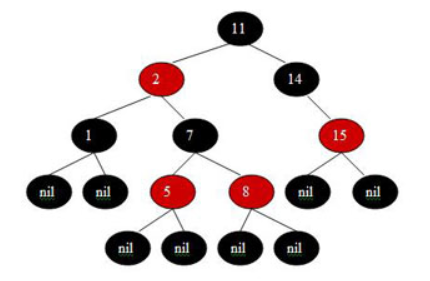
**Red-Black Tree**:

* **Introduction:**
  + One problem that can occur with binary search trees is that they can become completely unbalanced (figure 13), which can cause most of the tree’s operations to execute in O(n) in the worst case, where n is the number of nodes.



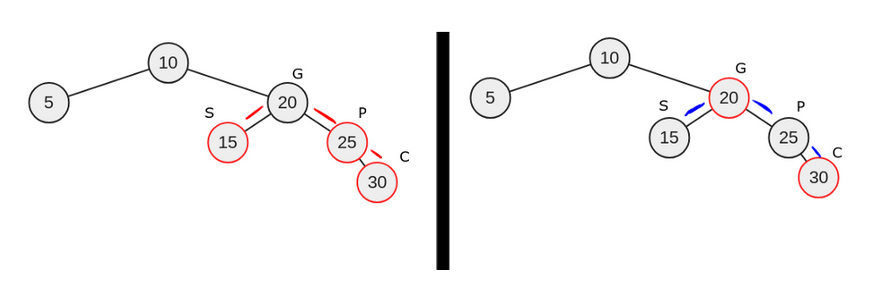
**Figure 13: Unbalanced Binary Search Tree**

* + The majority of operations performed on a Binary Trees are dependent on the height of the tree. This is why it is important to keep the tree’s height as small as possible, and when the height is as small is possible we say the tree is a balanced tree.
  + A balanced tree is a usefully property, because it reduces the worst case time complexity, of most BST operations, to O(log2 n).
  + A self-balancing tree is a tree that will rebalance itself after specific operations. An example of a self-balancing tree is a Red-Black Tree (shown in figure 14).
  + A Red-Black tree aims to keep the self-balancing property by coloring the nodes of the tree either red or black, while also preserving a specific sets of properties (described in the next section).



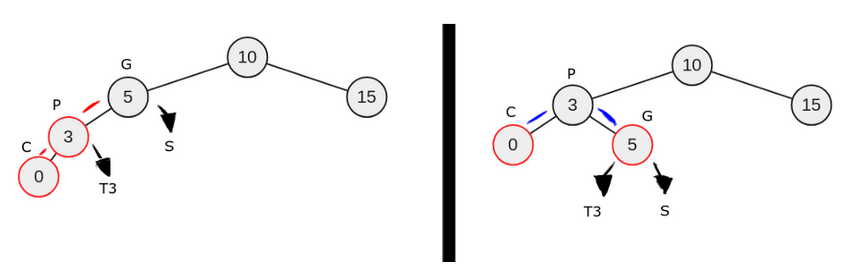
**Figure 14: Example of a red black tree**

* **Properties:**
  + A Red-Black is a binary search tree that contains the following properties.
    - Every node is colored with either red or black.
    - The root node is black.
    - All leaf (nil) nodes are black nodes.
    - Both children of a red node must be a black node.
    - Every path from a node n to a descendent leaf has the same number of black nodes (not counting node n). We call this number the black height of n, which is denoted bh(n).
  + The incoming (inserted) node is always red.
    - A double violation will occur if a child and parent node are both red.
      * We can resolve the violations with a recoloring of the node or a restructuring of the tree (rotation).
  + The properties of a Red-Black tree guarantee that the height of the tree will be O(log2 n).
* **Recoloring/Rotating (Insertion):** Recoloring and Rotating are methods that are used in order to rebalance a Red-Black tree.
  + Recoloring: is an operation that changes the color of a node in order to maintain the properties.
    - We recolor when the parent, the parent’s sibling, and the child are all red. The solution is to recolor the grandparent red, the parent and the parent’s sibling black. (Figure 15)



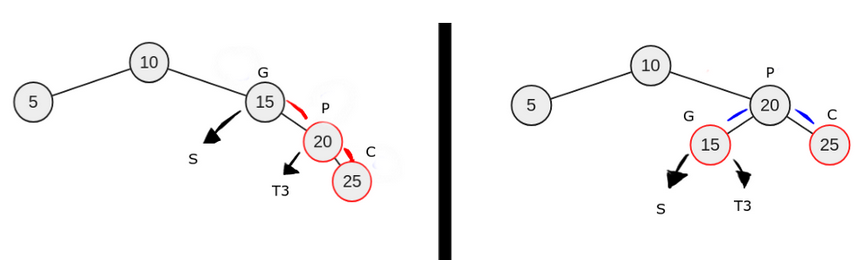
**Figure 15: Example of Recoloring**

* + A Rotation: is a binary operation that involves swapping and modifying the pointers of the grandparent, parent, and child nodes. Specifically, there are four types of rotation, a left left rotation, right right rotation, a left right rotation, and a right left rotation.
    - Left Left Rotation: A left left rotation occurs when we have a double red violation on the left parent and the left child, which can be seen below in figure 16.
      * We resolve the violation by recoloring P to black, G to red and having the right node of P then point to g. Also t3 needs to be reattach from the right pointer of P to the left pointer of G.



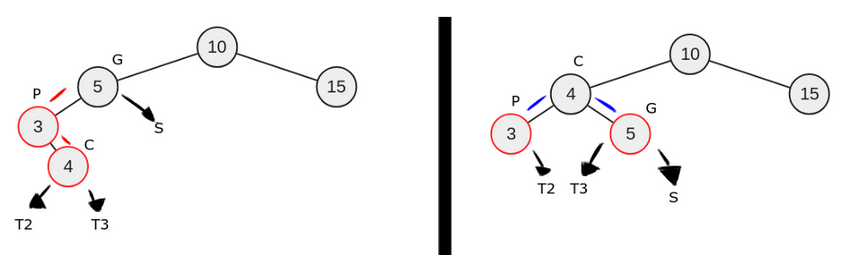
**Figure 16: Example of Left Left Rotation**

* + - Right Right Rotation: A right right rotation occurs when we have a double red violation of the right parent (relative to the grandparent) and right child both, which can be seen in figure 17.
      * We resolve the violation by recoloring P to black, G to red, and have the left pointer of P point to G. Also t3 need to be reattached to the right pointer of G.



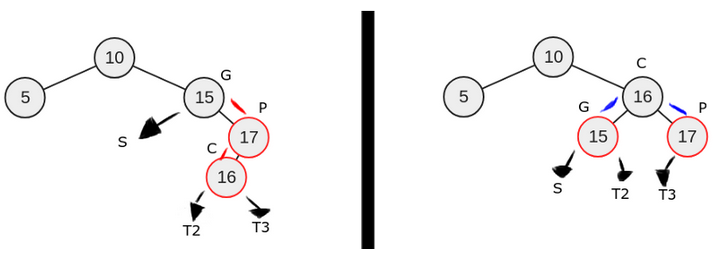
**Figure 17: Example of a right right rotation**

* + - Left Right Rotation: A left right rotation occurs when a double red violation occurs with the left parent (relative to the grandparent) and the right child (Example can be seen in figure 18). This rotation is sometimes referred to as a double rotation, because we can first rotate the parent with the child and then perform a left left rotation.
      * Left Right violation:
        + Grandparent (G) node is black
        + G’s left pointer points to the parent (P), which is red.
        + The sibling (S) of P is black or null
        + P’s right pointer points to the child (C), which is red
      * Violation Resolution:
        + Swap C and P but also maintain the BST order, so that C’s left pointer points to P.
        + Now we just perform a left left rotation.



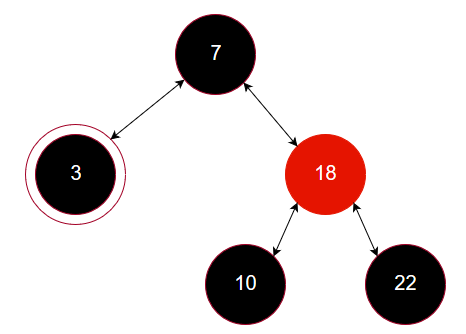
**Figure 18: Example of Left Right Rotation**

* + - Right Left Rotation: A right left rotation occurs when the grandparent’s right pointer points to the parent, which is colored red, and the parent’s left pointer points to the child, which is colored red (Example can be seen in figure 19). This rotation is also referred to as a double rotation.
      * Right Left Violation:
        + Grandparent (G) node is black
        + The right pointer of G points to the parent (P) node, which is red.
        + The sibling to the parent is black.
        + The left pointer of P points to the child (C) node, which is red.
      * Violation Resolution:
        + Swap C and P but also maintain the BST order, so that C’s right pointer points to P.
        + Perform a right right rotation.

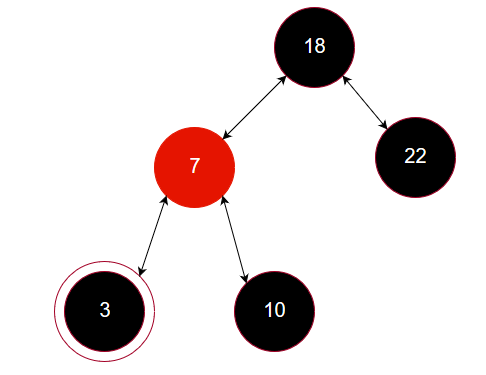


**Figure 19: Example of Right Left Rotation**

* **Operations:**
  + Search:
    - Same search operation described in the Binary Search Tree section.
  + Insertion:
    - Every node that is inserted is colored red unless it is the root node, which is black.
    - Insert according the node like you typically would with a binary search tree
    - Check for violation if a violation occurs, resolve the violation by either recoloring and/or restructuring, as shown in the previous section.
  + Removal: This operation is slightly more complex because we introduce the notion of a double black node, which is a black node that is deleted and replaced by a child node that is also black. Below we will work through the different cases of deletion. **Note:** We will denote the in order predecessor as *x*.
    - Case 1: *x* (the inorder predecessor) is red
      * Proceed as if it was a normal BST, replace the data in *x* with the node you were originally trying to remove and then remove x.
    - Case 2: *x* is black with a red child
      * Connect parent of *x* with child of *x*.
      * Delete *x*
    - Case 3: *x* is black and has a black child.
      * Replace node with its child.
      * This makes *x* a double black node, which needs to be transformed into a normal black node. This is done by looking at 6 different cases.
      * Case 3-1 (Terminal): If the root node is double black
        + Change color to black and you are done.
      * Case 3-2: *x* is double black, has a black parent, and a red sibling (seen below in figure 20).
        + If the red sibling is the right child node of the parent, we perform a left rotate from the parent. If the red sibling is the left child, we perform a right rotate from the parent.
        + The original parent (node 7 in figure 20) gets recolored red.
        + The original sibling (node 18 in figure 20) gets recolored black.

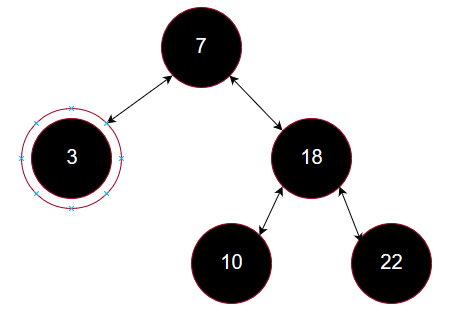


**Figure 20: Example of Delete Case 3-2**

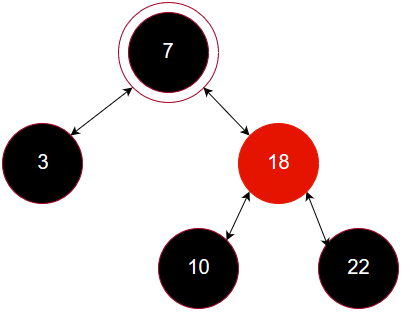


**Figure 21: Delete Case 3-2 after fix**

* + - * Case 3-3: *x* is double black, has a black parent, a black sibling, and the sibling’s children are both black (Seen below in figure 22)
        + We push the double black up to the parent.
        + Change the color of the sibling to red.
        + The new structure can be seen below in figure 23.

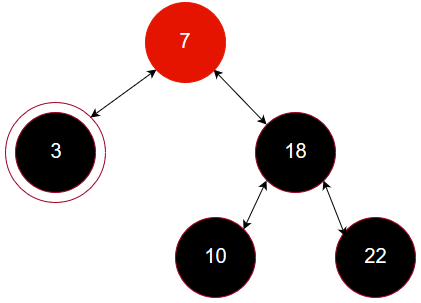


**Figure 22: Example of Delete Case 3-3**

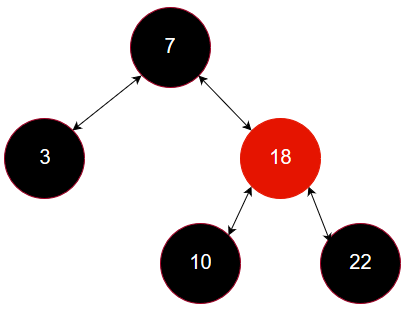


**Figure 23: Delete Case 3-3 after fix**

* + - * Case 3-4 (Terminal): *x* is double black, has a red parent (seen in figure 24), and both sibling’s children are black
        + We set the parent to black.
        + We set the sibling to red.
        + The new structure can be seen below in figure 25

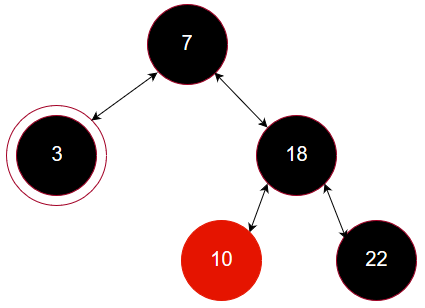


**Figure 24: Example of Delete Case 3-4**

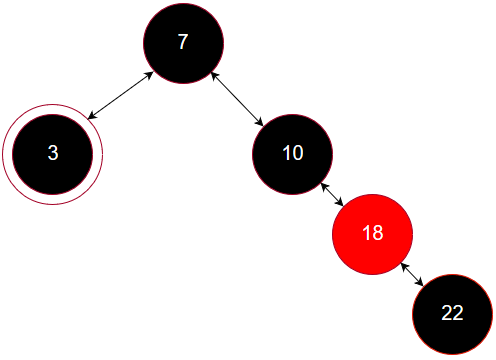


**Figure 25: Delete Case 3-4 after fix**

* + - * Case 3-5: *x* is double black, the sibling is black and the sibling’s inner child is red. (Seen below in figure 26).
        + If the Sibling node is the parent’s right child, we perform a right rotation around the sibling node. If the sibling node is the parent’s left child, we perform a left rotation around the sibling node.
        + We need to change the sibling’s color to red and the sibling’s inner child’s color to black after the rotation.
        + The new structure can be seen below in figure 27.

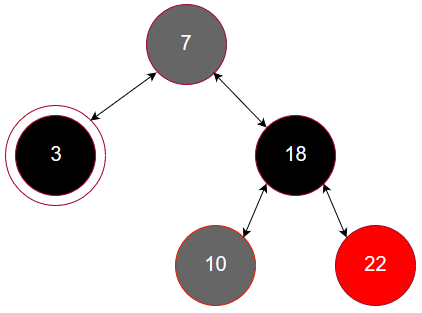


**Figure 26: Example of Delete Case 3-5**

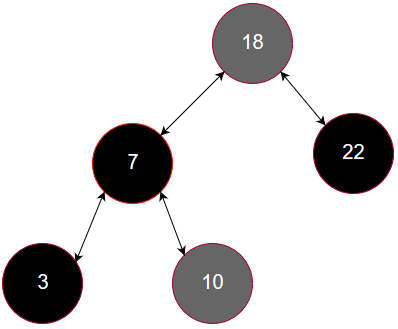


**Figure 27: Delete Case 3-5 after fix**

* + - * Case 3-6 (Terminal): *x* is a double black, *x*’s sibling is black and *x*’s sibling’s out child node is red (Seen below in figure 28). Note we don’t care about the parent node’s color or sibling’s inner child’s color, (colored gray in figure 28).
        + If *x*’s sibling is the right child of the parent, we perform a left rotation from the parent. If *x*’s sibling is the left child of the parent, we perform a right rotation from the parent.
        + After the rotation we recolor the double black to black, the parent’s node to black and the sibling’s outer child to black.
        + The new structure can be seen below in figure 29.

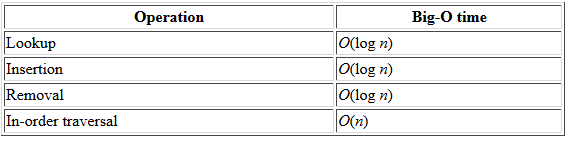


**Figure 28: Example of Delete Case 3-6**



**Figure 29: Delete Case 3-6 after fix**

* **Time Complexities:** Time complexities of each Red-Black tree operation can be seen below in figure 30.



**Figure 30: Table of time complexities for each operation of a Red-Black Tree.**

* **Other Self-Balancing Trees:**
  + *AVL Tree* – The first designed self-balancing tree
  + *AA-Tree* – A slightly modified version of a Red-Black Tree, where the left child can only be a black node. This is to simplify the delete operation by reducing the number of cases that need to be handled.
  + *Splay Tree* – The main purpose of a splay tree is to keep the recently accessed or searched for nodes at the root. The reasoning for this is that in typically applications 80% of the access are on 20% of the items. Note that this is more like self-adjusting tree than a self-balancing tree, we can still have operations run in O(n) in the worst case but on average runs in O(log2n).
  + *B-Tree* – (<https://www.geeksforgeeks.org/b-tree-set-1-introduction-2/>)
* **References:**

1. <https://appliedgo.net/balancedtree/>
2. <https://towardsdatascience.com/red-black-binary-tree-maintaining-balance-e342f5aa6f5>
3. <https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/>
4. <https://www.geeksforgeeks.org/red-black-tree-set-2-insert/>
5. <https://www.geeksforgeeks.org/red-black-tree-set-3-delete-2/>
6. <https://www.cpp.edu/~ftang/courses/CS241/notes/self%20balance%20bst.htm>
7. <https://www.topcoder.com/community/data-science/data-science-tutorials/an-introduction-to-binary-search-and-red-black-trees/>
8. <https://www.d.umn.edu/~gshute/ds/binary-tree/red-black-tree.xhtml>
9. <http://www.cs.toronto.edu/~wgeorge/csc265/2013/09/26/tutorial-3-red-black-tree-deletion.html>
10. <http://software.ucv.ro/~mburicea/lab8ASD.pdf>
11. <https://www.youtube.com/user/AMGaweda/videos>
12. <https://www.youtube.com/watch?v=CTvfzU_uNKE&t=1s>
13. <https://www.youtube.com/watch?v=aA-nLw28eUw>