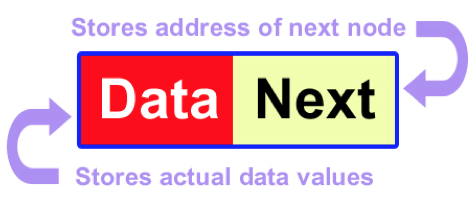
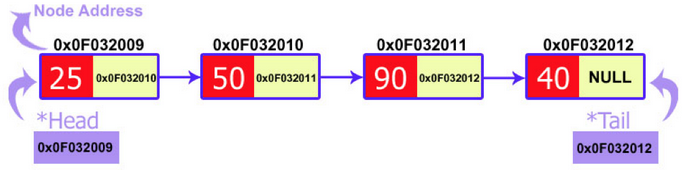
**Data Structure Notes**

**Linked-List:**

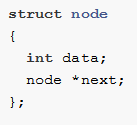
* **Introduction:**
  + A linked list is a data structure that can store an indefinite amount of items. These items are connected using pointers in a sequential manner.
  + A visual representation of a single node and the complete data structure can be seen below:



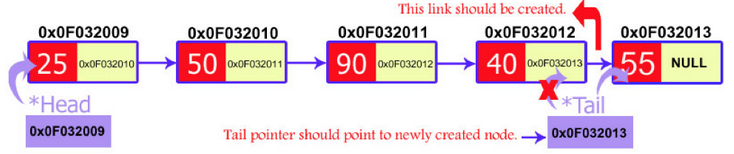


**Figure 1: Visual Representation of a Singly Linked List**

* + 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:

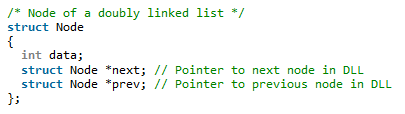


* + Linked List Operations:
    - Checking whether the list is empty.
    - Appending to a node.
    - Prepending to a node.
    - Inserting a node.
    - Deleting a node (either by index or data value).
    - Reversing a linked list.
    - Traversing a linked list.
  + Adding a node to the list: Appending 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:

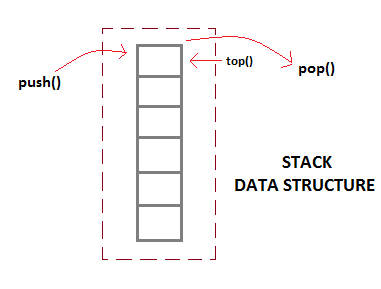


* + The operations are very similar to the singly linked list but we have more flexibility. Since we are not restricted to only iterating through the list by starting at the head of the list. We can start at the tail of the list and iterate to the head.
* **References:**

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 with a bounded (predefined) capacity. 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).



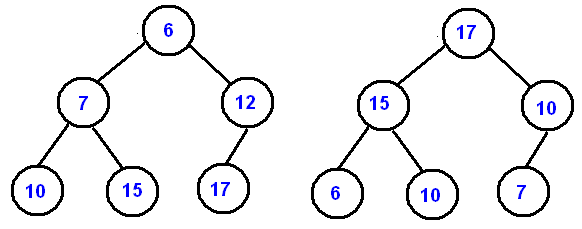
**Figure 3: Visual Representation of Stack Data Structure**

* + 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>

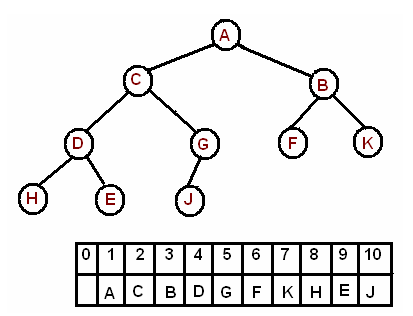
**Binary (MIN/MAX) Heaps:**

* **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.



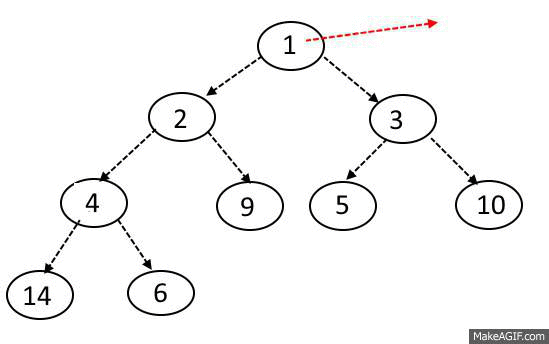
**Figure 4: Example of a Min Binary Heap (left) and a Max Binary Heap (right)**

* **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**

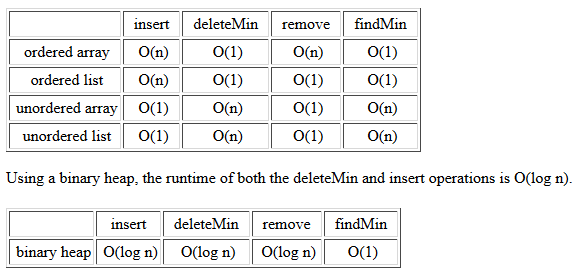
* + Index from 1:
    - Arr[i/2]: returns the **parent** node of index i.
    - Arr[(2\*i)]: returns the **left** child node of index i.
    - Arr[(2\*i) + 1]: returns the **right** child node of index i.
  + Index from 0:
    - Arr[(i – 1)/2]: returns the **parent** node of index i.
    - Arr[(2\*i) + 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.

**[](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/)