* Algorithms:

1. A finite set of rules that give a sequence of operations for solving a specific type of problem – Knuth
2. Algorithms are software programs, which manipulate the data.
3. No pointers: 1. Easier to write and understand, b. more secure and less errors.

* Following are important categories of algorithm:

1. Sort
2. Insert/Update/Delete
3. Search

* Characteristics of an Algorithm:

1. Unambiguous
2. Well defined input – 0 or more inputs
3. Well defined output – 1 or more outputs
4. Finiteness – Must terminate after finite steps
5. Feasibility – Should be feasible with the available resources
6. Independent – Step by step independent of any programming languages.

* Algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of the following properties: 1. Time Complexity, 2. Space Complexity.
* **Space Complexity:** Amount of memory space required by the algorithm, during the course of its execution. **(IDE)**

1. **Instruction Space:** Space for executable version
2. **Data Space:** Space required to store all the constants and variables value.
3. **Environment Space:** Space to store environment information.

* **Time Complexity:** The amount of time needed by the program to run to completion (big O notation).

1. Time is estimated by counting the number of elementary functions performed by algorithm
2. Varies based upon input data.
3. Always calculate the worst-case time complexity.

* Big O notation is a way of measuring the efficiency of an algorithm and how well it scales based on the size of the dataset.

1. **O(1) / Constant Time:** Algorithm will always takes same amount of time irrelevant of data set size.
2. 1 item takes 1 second, 100 items takes 1 second.
3. **O(log n) / Logarithmic complexity:** Time taken increases with the size of the dataset, but not proportionately.
4. 10 items takes 2 seconds, 100 items takes 3 seconds
5. Not good as constant but still pretty good.
6. Algorithms very scalable.
7. **O(n) / Linear complexity/ Linear Time:** Time grows proportionately.
8. 1 item takes 1 second, 10 items takes 10 seconds.
9. **O(n log n):** Combination of above two. First loop O(n) and second loop is O(log n)
10. 1 item takes 2 seconds, 10 items takes 12 seconds.

* **Types of Notations for Time complexity:**

1. **Big Oh:** Fewer or same as <expression> iterations.
2. **Big Omega:** More or same as <expression> iterations.
3. **Big Theta:** Same as <expression> iterations.
4. **Little Oh:** Fewer than <expression> iterations.
5. **Little Omega:** More than <expression> iterations.

* **Asymptotic analysis:** Analysis the algorithm time/space change based on input.

Correctness of an Algorithm:

1. Step 1: Statement to be proven
2. Step 2: List all assumptions
3. Step 3: Chain of reasoning from assumptions to the statement.

* Why?

1. Time consumption
2. Hardware requirements
3. Feasible or not
4. Improvement

- Data structures define how the data’s are collected and arranged in computer memory or disk so that we can perform operations on these data in an effective way.

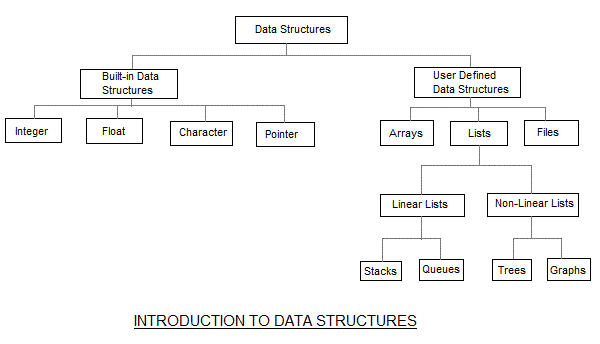
e.g. Arrays, LinkedList, Stack, Binary trees, Hash tables.

1. Real-World Data storage

2. Programmer's tools

3. Modeling

**Data Structures:**

****

|  |  |  |  |
| --- | --- | --- | --- |
| **SL. NO** | **Data Structure** | **Advantages** | **Disadvantages** |
| 1. | Array | Fast access if we know the index, Quick insertion | Slow Search, slow deletion |
| 2. | Ordered Array | Quicker search than Array | Slow deletion |
| 3. | Stack | Last-in, first-out | Slow access to other items |
| 4. | Queue | First-in, first-out | Slow access to other items |
| 5. | Linked List | Quick insertion/Quick deletion | Slow Search |
| 6. | Binary tree | Quick search, insertion, deletion | Deletion algorithm is complex. |
| 7. | Red-black tree | Quick search, insertion, deletion, tree always balanced. | Complex |
| 8. | 2-3-4 tree | Quick search, insertion, deletion, tree always balanced. Good for disk storage. | Complex |
| 9. | Hash table | Very fast access if key know, fast insertion. | Slow deletion if the key not known, inefficient memory usage. |
| 10. | Heap | Fast insertion, deletion, access to largest item. | Show access to other items. |
| 11. | Graph | Models real-world situations. | Some algorithms are slow and complex. |

Overview of Algorithm:

1. Insert a new data item.
2. Search for specified item.
3. Delete a specified item.

* Software engineering is the study of ways to create large and complex computer programs involving many programmers.
* Software engineering includes specification, design, verification, coding, testing, production and maintenance.

**Arrays:**

* Insert/Search/Delete based upon index.

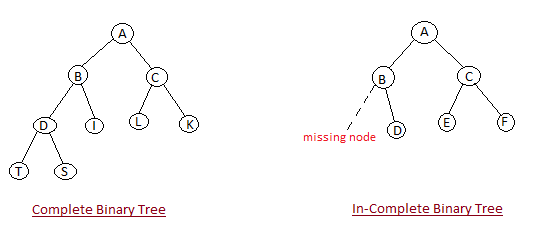
**Sorting:** It is a way of storing a data in a sorted order, the order maybe either ascending or descending.

**Types of sorting Techniques:**

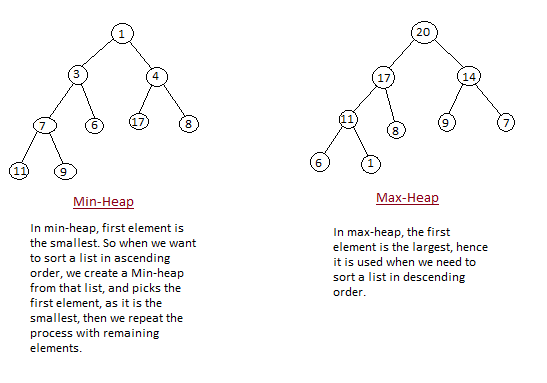
1. Bubble sort
2. Selection sort
3. Quick sort
4. Merge sort
5. Insertion sort
6. Heap sort

**Simple sorting:**

1. **Bubble sort:** The bubble sort repeatedly compares adjacent elements of an array. The first and second elements are compared and swapped if out of order. Then the second and third elements are compared and swapped if out of order. This sorting process continues until the last two elements of the array are compared and swapped if out of order. When this first pass through the array is complete, the bubble sort returns to elements one and two and starts the process all over again.
2. **Insertion sorting:**
3. **Merge Sort Algorithm:**
4. Follows Divide and Conquer.
5. Doesn’t divide two halves, divided into N sub lists, each having one element, because a list of one element is considered sorted, and then repeatedly merge.
6. It is fast and time complexity is O(n log n)
7. Very stable sort.
8. **Selection Sorting:**
9. Simplest sorting algorithm
10. The algorithms first finds the smallest element in the array and exchanges it with the element in the first position, then finds the second smallest element and exchange it with the element in the second position and continues in this way until the entire array is sorted.
11. **Heap Sort:**
12. Simplest sorting algorithm
13. The algorithms first finds the smallest element in the array and exchanges it with the element in the first position, then finds the second smallest element and exchange it with the element in the second position and continues in this way until the entire array is sorted.
14. Heap is a special tree-based data structure, that satisfies the following heap properties**:**
15. **Shape Property:** Complete binary tree, all levels of the tree are fully filled.

****

1. **Heap Property:** All nodes are either [greater than or equal to] or [less than or equal to] each of its children. If the parent nodes are greater than their children, heap is called a **Max-Heap,** and if the parent nodes are smaller than their child nodes, heap is called **Min-Heap.**

****

* First step is to create a Heap data structure (Max-Heap or Min-Heap)
* Once heap is built, we start comparison (N/2 where N nodes) and move max value to parent node.
* Delete the root node (Once we reached the first parent, take the element and put in the sorted list).
* Put the last node in the root node position.
* Repeat the step to create Max Heap
* **heapsort(), buildheap,** and **satisfyheap**()
* **buildHeap:** produces a max/min heap array from an unordered array.
* **Heap build algorithm, where i is the index…**

iParent(i) = floor((i-1) / 2)0

iLeftChild(i) = 2\*i + 1

iRightChild(i) = 2\*i + 2

1. **Quicksort:**

* Based on Divide and conquer.
* Obtain a pivot element, divide into two sub arrays.
* Sort left array lesser than or equal to pivot element.
* Sort right array greater than or equal to pivot element.
* Combine both the arrays.
* Pick the last element as pivot (Implementation)
* The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

1. **Shellsort:**

* It is based on insertion sort algorithm
* In this sorting algorithm we compare elements that are distant apart rather than adjacent.
* gap < n where n is the number of elements.
* In each pass we reduce gap until 1
* In last pass we do insertion sort.

**Pseudo-code:**

1. Use left arrow (🡨 ) instead of “=”
2. Use “=” instead of “==”
3. Use Standard mathematical numerical and Boolean expressions.

**Programming exposed:**

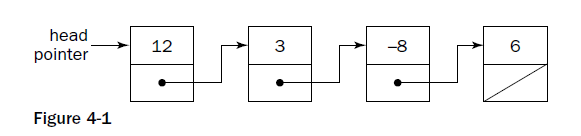
* Types of linked list:

1. Single, 2. Double, 3. Circular

* Head: first element
* Tail: last element
* Elements are referenced with pointers.
* Implementation stacks, queues, graph, etc.,
* **Advantages:**
* Allocates the memory when required.
* Insertion and deletion is easy
* Stacks and Queues are easily executed.
* Reduces the access time.
* **Disadvantages:**
* Because of pointer extra memory required.
* Elements cannot be accessed randomly needs to accessed sequentially.
* Reverse traversing is difficult

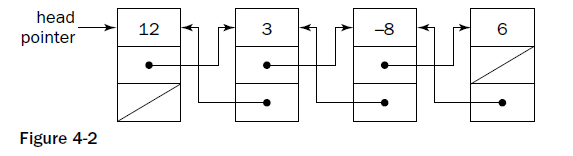
**Singly Linked List:**

* Each element has a reference to next element.
* Tail element has “null” reference.
* Traverse in forward direction.
* **Demerits**:
* We need to start from head element to locate any element because each element point to the adjacent.
* Find Mth-to-the last element in NOT time and space efficient.



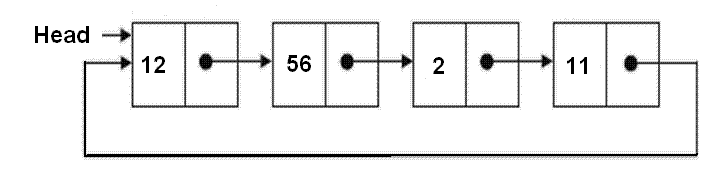
**Doubly linked List:**

* Each element has a reference to next element and previous element.
* Head and tail has null reference.
* The list can traverse in both directions.



**Circular Linked List:**

* Have no ends or tails.
* Tail element reference to head element.
* During traversal you have to track the starting point otherwise you’ll be in loop.
* E.g. a. multiple applications running in a personal computer, OS gives a fixed slot to all for running.



**Basic Linked List operations:**

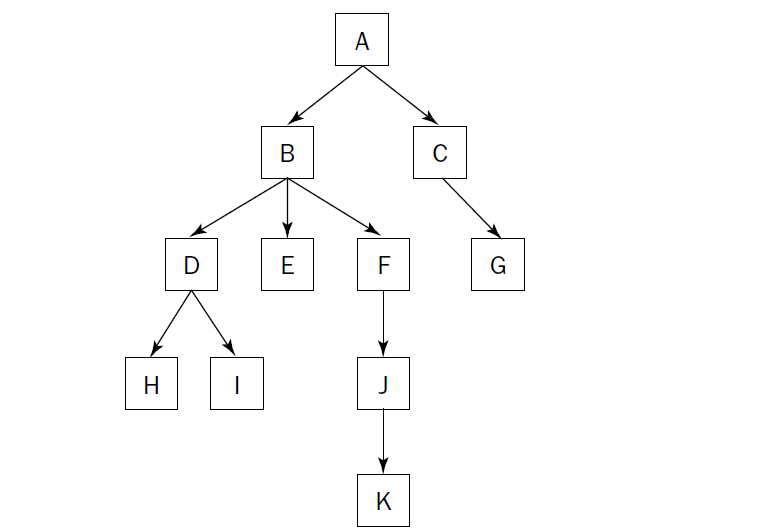
* The head of a single linked list must be always track; otherwise it will be lost in memory.
* When we insert/delete update head first.
* During insert/delete operation update the point reference of the adjacent element.

**Stacks:**

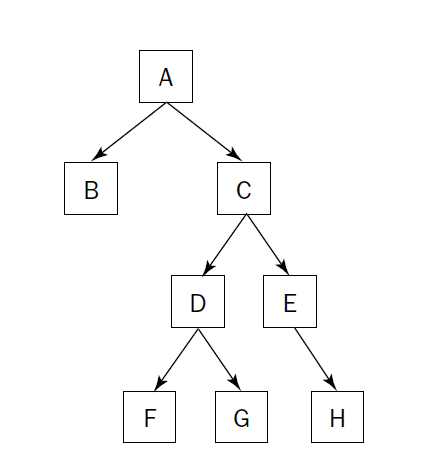
* **LIFO** (similar to add/remove plates to a stack of plates)
* **Operations: push, pop** (increase the memory when push a new element happens)
* One of the way to implement using **dynamic array**

**Trees and Graphs:**

* Trees are made up of nodes, it start with zero and have references to several other nodes. Each node has only one other node referencing it.
* Mainly to stored sorted data.



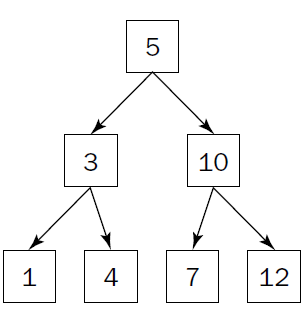
* Tree vocabulary:
  + Parent (Root node does not have parent node)
  + Child
  + Descendant: All child, grandchild, great grandchild, etc., nodes.to a particular parent node is called Descendants.
  + Ancestor: All parent, grand parent, great parent, etc., to a particular child node is called ancestors.
  + Leaves: Don’t have any child nodes. E.g. G, H, I, and K
* Binary Trees:
  + No more than two children.



* + When the element has no node then corresponding reference is null.

**Binary Search tree:**

* The value held by a left node is less than or equal to its child node.
* The value held by a right node is greater than or equal to its child node.
* Locating a particular node is faster.
* Many tree operations can be implemented recursively.



**Heaps:**

* Refer min-heap, max-heap above.
* Heaps are trees with a twist.
* Very useful in search mechanism.

**Traversals:**

* Similar to search, instead of stopping in a particular node, we visit every node in the tree.

1. **Preorder**: Visit the Node, Recur Left, Recur Right (**NLR**).
2. **Inorder**: Recur left, Visit the node, recur the right (**LNR**).
3. **Postorder**: Visit left, right, then node (**LRN**).

**Algorithm preorder:**

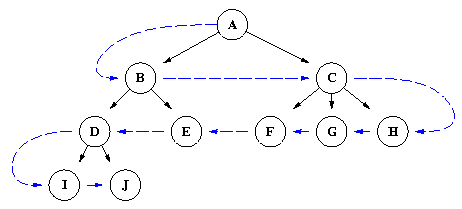
1. Visit the root
2. Traverse the left subtree, i.e., call preorder(left-subtree)
3. Traverse the right subtree i.e., call postorder(right-subtree)

**Deletion:**

1. Deleting a leaf node (node without children)
2. Deleting a node with one child
3. Deleting a node with two children.

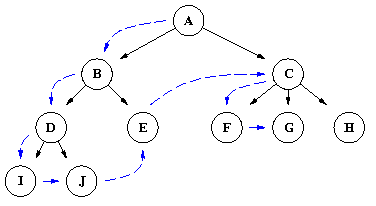
**Breadth-First search:**

* 1. Start from root, 2. Move left to right, 3. Move left to right (snake movement).
* It uses additional memory.



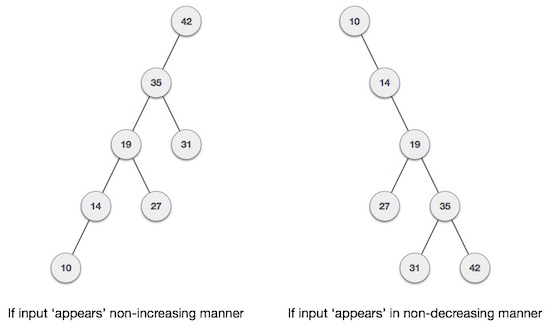
**Depth-First search:**

* 1. Start from root node, 2. Search one level of the tree down as many levels until the target found, 3. Continue the nearest ancestor.
* Lower memory requirements.



**AVL (**Adelson, Velski, Landis):

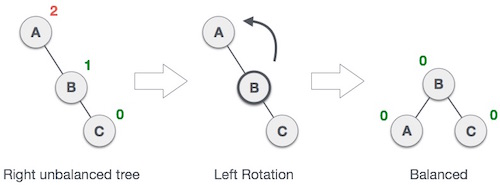
* AVL is a self-balancing binary tree where the heights of left and sub trees cannot be more than 1.



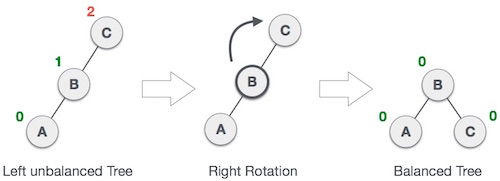
* In above binary search tree the linear search operation is worse.
* AVL tree checks the height of left and right sub-trees and assures the difference is not more than 1 (so possible values are 1, -1, 0). This is called **balance factor**.
* **Balance factor = height (left sub tree) – height (right sub tree)**
* If it is not balanced, use any of the below techniques to balance it.

1. Left rotation (Single rotation)
2. Right rotation (Single rotation)
3. Left-Right rotation (double rotation)
4. Right-Left rotation (double rotation)

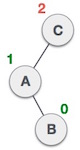
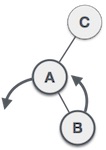
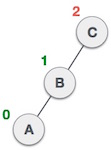
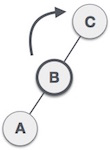
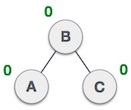
* Minimum required two nodes.
* **Left rotation:**



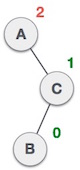
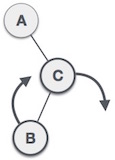
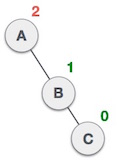
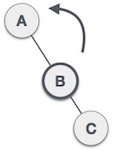
* **Right rotation:**



* **Left-Right rotation:**

1. 2. 3. 4. 5. 

* **Right-Left rotation:**

1. 2. 3.  4.  5. 