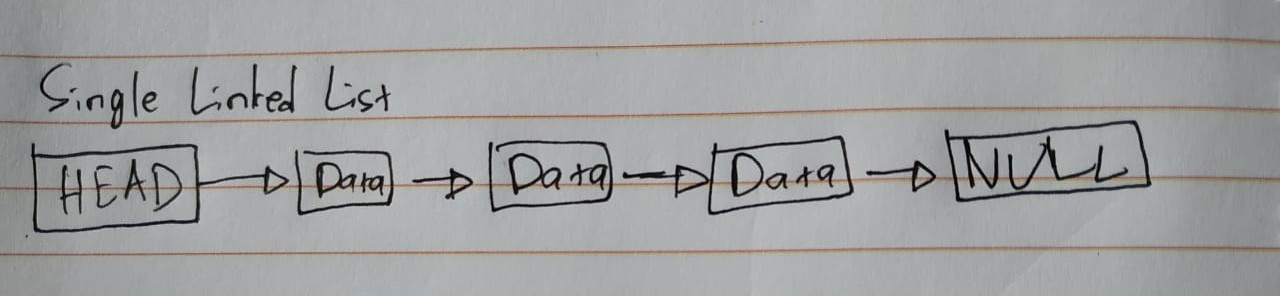
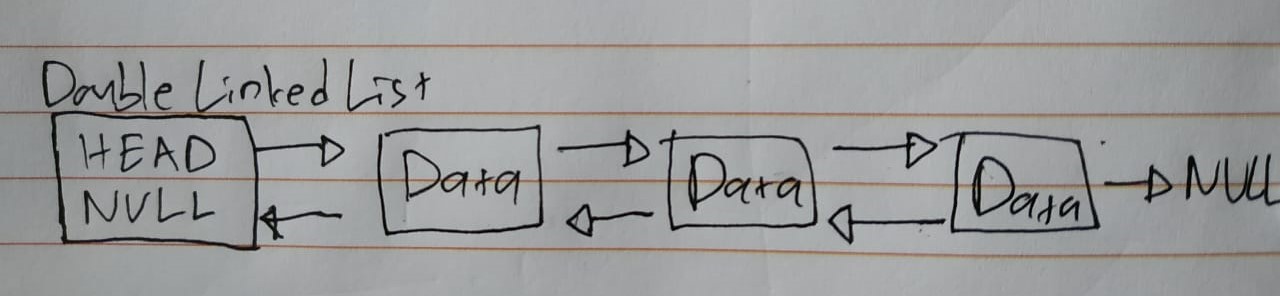
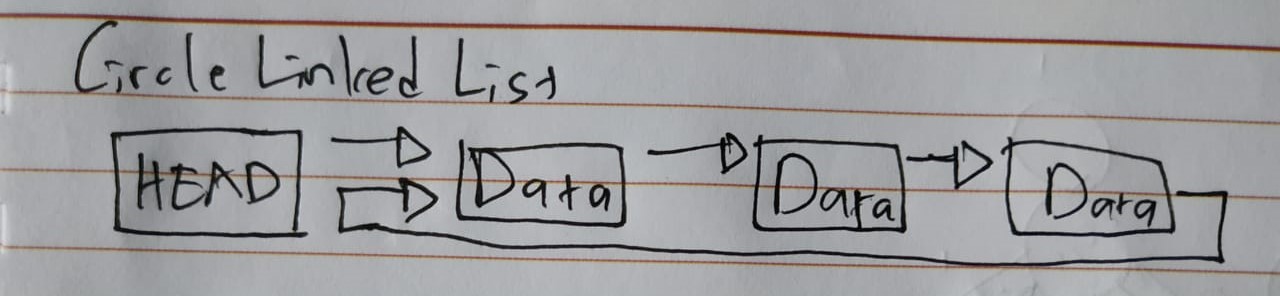
* **Linked List**

1. Single, Double, Circular Linked List in Graphical Order:







1. Differences between Array and Linked List:

|  |  |  |
| --- | --- | --- |
| **Comparison** | **Array** | **Linked List** |
| Data | Consistent set with fixed number of data items | Ordered set with variable number of data items |
| Size | Specified when declared | Grow and shrink during execution |
| Storage Allocation | Allocated during compile time | Assigned during run time |
| Element Order | Stored consecutively | Stored randomly |
| Element Access | Specify the array index or subscript | Using traverse from first node in the list |
| Element Insert + Delete | Slow; needs shifting | Easier, fast, more efficient |
| Searching | Binary and linear search | Linear search |
| Memory required | Less | More |

1. Floyd-Warshall Algorithm is **an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph**. This algorithm works for both the directed and undirected weighted graphs. It does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

Pseudocode/Algorithm:

programiz.com

n = no of vertices

A = matrix of dimension n\*n

for k = 1 to n

for i = 1 to n

for j = 1 to n

Ak[i, j] = min (Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j])

return A

* **Stack and Queue**

1. Differences between Stack and Queue:

|  |  |  |
| --- | --- | --- |
| **Comparison** | **Array** | **Linked List** |
| Working Principal | Last In First Out (LIFO) | First In First Out (FIFO) |
| Structure | Same end is used to insert and delete elements | One end is used for insertion and another end is used for deletion of elements |
| Number of pointers used | 1 | 2 (Simple Queue) |
| Operations performed | Push and Pop | Enqueue and Dequeue |
| Variants | It does not have variants. | It has variants like circular queue, priority queue, doubly ended queue |

1. Prefix, Infix, Postfix

|  |  |  |  |
| --- | --- | --- | --- |
| **Prefix** | **Infix** | **Postfix** | **Notes** |
| + \* A B / C D | A \* B + C / D | A B \* C D / + | multiply A and B, divide C by D, add the results |
| / \* A + B C D | A \* (B + C) / D | A B C + \* D / | add B and C, multiply by A, divide by D |
| \* A + B / C D | A \* (B + C / D) | A B C D / + \* | divide C by D, add B, multiply by A |

* **Prefix notation** (also known as "Polish notation"): + X Y

Operators are written before their operands. The expressions given above are equivalent to / \* A + B C D

* **Infix notation**: X + Y

Operators are written in-between their operands. This is the usual way we write expressions. An expression such as A \* ( B + C ) / D is usually taken to mean something like: "First add B and C together, then multiply the result by A, then divide by D to give the final answer."

* **Postfix notation** (also known as "Reverse Polish notation"): X Y +

Operators are written after their operands. The infix expression given above is equivalent to A B C + \* D /

Implementation:

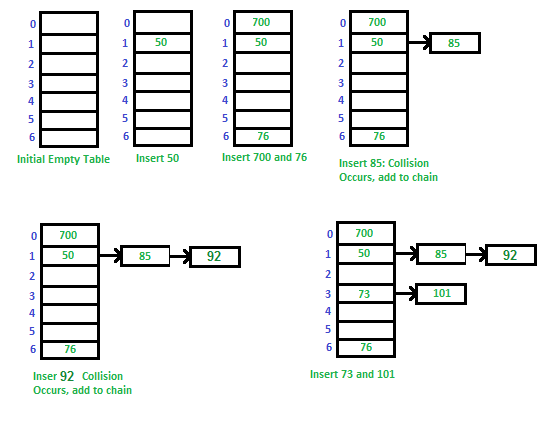
* + - * **Prefix**: is often used for operators that take a single operand (e.g. negation) and function calls.
      * **Infix**:very common in mathematics, used in most computer languages (e.g. a simple Infix calculator).
      * **Postfix**: is slightly easier to evaluate in simple circumstances, such as in some calculators (e.g. a simple Postfix calculator), as the operators really are evaluated strictly left-to-right
* **Hashing and Hash Tables**

1. Hash table, hash functions, collision:
   * Hash table: Tables which can be searched for an item in O(1) time using a hash function to form an address from the key.
   * Hash function: Function which, when applied to the key, produces a integer which can be used as an address in a hash table.
   * Collision: When a hash function maps two different keys to the same table address, a collision is said to occur.
2. 2 methods of collision:
   * Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Source: <https://www.geeksforgeeks.org/hashing-set-2-separate-chaining/>

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



* + Open Addressing:

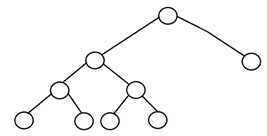
All elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



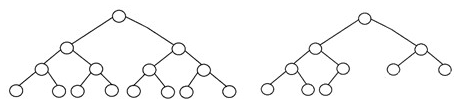
* **Binary Search Tree**
  1. 5 Binary Trees (source: <https://www.upgrad.com/blog/5-types-of-binary-tree/>):
     1. Full

has either zero children or two children. It means that all the nodes in that binary tree should either have two child nodes of its parent node or the parent node is itself the leaf node or the external node.



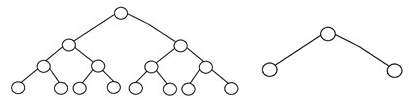
* + 1. Complete

all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also, in the last or the lowest level of this binary tree, every node should possibly reside on the left side.



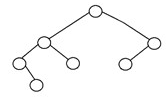
* + 1. Perfect

all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree. A perfect binary tree having height ‘h’ has 2h – 1 node.



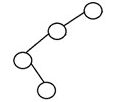
* + 1. Balanced

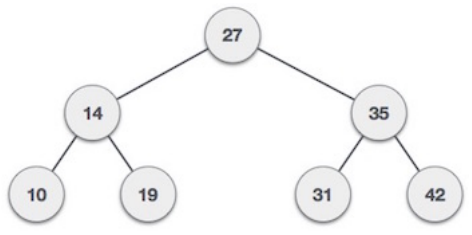
the tree height is O(logN), where ‘N’ is the number of nodes. In a balanced binary tree, the height of the left and the right subtrees of each node should vary by at most one.



* + 1. Degenerate

every internal node has only a single child.





* 1. Insertion: **24, 18, 55**
     1. Check if **24** is lower or higher than *27* 🡪 lower
     2. **24** moves to the left side, check if **24** is lower or higher than *14* 🡪 higher
     3. **24** moves to the right side, check if **24** is lower or higher than *19* 🡪 higher
     4. **24** moves to the right side, and become the child of *19*.
     5. Check if **18** is lower or higher than *27* 🡪 lower
     6. **18** moves to the left side, check if **18** is lower or higher than *14* 🡪 higher
     7. **18** moves to the right side, check if **18** is lower or higher than *19* 🡪 lower
     8. **18** moves to the left side, and become the child of *19*.
     9. Check if **55** is lower or higher than *27* 🡪 higher
     10. **55** moves to the right side, check if **55** is lower or higher than *35* 🡪 higher
     11. **55** moves to the right side, check if **55** is lower or higher than *42* 🡪 higher
     12. **55** moves to the right side, and become the child of *42*.
  2. Deletion: **27, 35, 42**
     1. Search for **27** 🡪 found at root 🡪 deleted 🡪 find another root; either biggest number on left side or smallest number on right side 🡪 left side 🡪 biggest number is 24 🡪 24 is the new root.
     2. Search for **35** 🡪 Check if **35** is lower or higher than 24 🡪 higher
     3. Move to the right side 🡪 found **35** 🡪 delete **35** 🡪 find new parent, take the left side/left child which is *31* 🡪 set *31* as new parent 🡪 *42* stays in place and does not move anywhere, and become the child of *31*.
     4. Search for **42** 🡪 Check if **42** is lower or higher than *24* 🡪 higher
     5. Move to the right side 🡪 Check if **42** is lower or higher than *31* 🡪 higher
     6. Move to the right side 🡪 found **42** 🡪 delete **42** 🡪 *55* become the child of *31* now.