Data Structure

Table Of Contents.

Data Structures

Data Structure is a way to store data elements into the memory (i.e. into the main memory) in an organized manner so that operations like addition, deletion, traversal, searching, sorting etc... can be performed on it efficiently.

- There is a need of data structure to acheive following 3 things in programming:

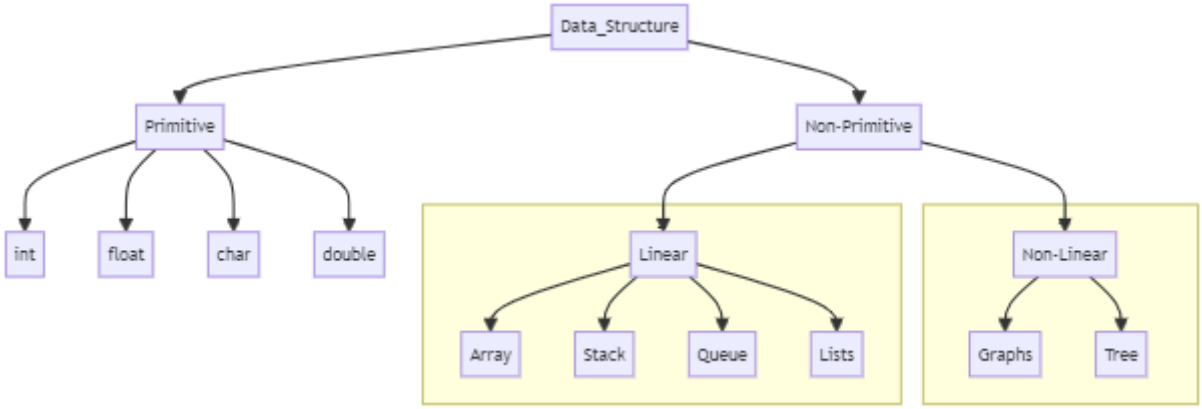
1. Efficiency
2. Abstraction
3. Reusability

Types of Data structures

1. Primitive data structures.
2. Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category.



1. **Linear/Basic:** Data elements gets stored into the memory in a linear manner and hence can be accessed linearly.
    - Array
    - Structure & Union
    - Class
    - Linked List
    - Stack
    - Queue
2. **Non-linear/Advanced:** Data elements gets stored into the memory in a non-linear nmanner and hence can be accessed non-linearly.
    - Tree (Hierarchical)
    - Graph

---

1. **Array:** An array is a data structure that stores a collection of elements of the same data type in Contigious manner. Arrays are indexed, which means that each element in the array can be accessed by its index. The index of the first element in an array is 0, and the index of the last element is one less than the length of the array.
2. **Structure:** It is a collection of logically related similar and disimmilar type of elements gets stored into the memory collectively (as a single entity/record). Sizeof of structure is sum of size of all its members.
3. **Union:** Union is same like structure, except, memory allocation i.e. size of union is the size of max size member defined in it and that memory gets shared among all its members for effective memory utilization (can be used in a special case only).

4. **Class:** Inside class we can combine/collect "data members" (variables) as well as "member functions" (function can be used to perform operations on, data), wheras in a structure we can combine/collect only data members.

Program

- A program is a set of instructions written in any programming language (like C, C++, Java, Python, Assembly etc...) given to the machine to do specific task.
- An algorithm is a template whereas a program is an implementation of an algorithm.

Algorithm

- An algorithm is a finite set of instructions written in human understandable language (like english), if followed, acomplishes a given task.
- An algorithm is a finite set of instructions written in human understandable language (like english) with some programming constraints, if followed, acomplishes a given task, such an algorithm also called as **pseudocode**

Example: An algorithm to do sum of all array elements

```
Algorithm ArraySum(A, n)//whereas A is an array of size n
{
sum=0;//initially sum is 0
for( index = 1 ; index ⩽ size ; index++ ) {
sum += A[ index ];//add each array element into the sum
}
return sum;
}
```

In this algorithm, traversal/scanning operation is applied on an array. Initially sum is 0, each array element gets added into to the sum by traversing array sequentially from the first element till last element and final result is returned as an output.

Problem : "Sorting": it is a process to arrange data elements in a collection/list of elements either in an ascending order or in a descending order.

- Sorting Algorithms/Solutions :

1. Selection Sort
2. Bubble Sort
3. Insertion Sort
4. Quick Sort
5. Merge Sort
6. Radix Sort
7. Bucket Sort

8. Shell Sort

- One problem has many solutions, and hence we need to select an efficient solution.
- To decide efficiency of an algorithms we need to do their analysis
- **Analysis of an algo** is a work to calculating how much **"time"** i.e. computer time and **"space"** i.e. computer memory an algo needs to run to completion.

Time Complexity

The time needed by an algorithm expressed as a function of the size of a problem is called the TIME COMPLEXITY of the algorithm. The time complexity of a program is **the amount of computer time it needs to run** to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity

The space complexity of a program is the **amount of memory it needs to run** to completion.

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions. **Data space:** Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

Asymptotic Analysis: It is a mathematical way to calculate time complexity and space complexity of an algorithm without implementing it in any programming language.

In this type of analysis, analysis can be done on the basis of basic operation in that algorithm. **e.g.** In searching & sorting algorithms comparison is the basic operation and hence analysis gets done on the basis of no. of comparisons, in addition of matrices algorithms addition is the basic operation and hence on the basis of addition operation.

Classification of Algorithms: If ‘n’ is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

Running Time	Description

Running Time	Description
Constant (1)	Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.
Logarithmic (log n)	When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction. When n is a million, log n is doubled whenever n doubles, log n increases by a constant, but log n does not double until n increases to 2n.
Linear (n)	When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.
Linearithmic (n.log n)	This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving them independently, and then combining the solutions. When n doubles, the running time more than doubles.
Quadratic (n^2)	When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop). Whenever n doubles, the running time increases fourfold.
Cubic (n^3)	Similarly, an algorithm that processes triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eightfold.
Exponential (2^n)	Few algorithms with exponential running time are likely to be appropriate for practical use. Such algorithms arise naturally as “brute–force” solutions to problems. Whenever n doubles, the running time squares.

1. **Best case time complexity:** If an algo takes min amount of time to complete its execution then it is referred as best case time complexity.
2. **Average case time complexity:** If an algo takes neither min nor max amount of time to complete its execution then it is referred as an average case time complexity.
3. **Worst case time complexity:** If an algo takes max amount of time to complete its execution then it is referred as worst case time complexity.

Asympotic Notations:

1. **Big Omega(Ω):** This notation is used to denote best case time complexity
2. **Big Theta(θ):** This notation is used to denote an average case time complexity
3. **Big Oh(O):** This notation is used to denote worst case time complexity

---

Analyzing Algorithm

Suppose ‘M’ is an algorithm, and suppose ‘n’ is the size of the input data. Clearly the complexity `f(n)` of M increases as n increases. It is usually the rate of increase of `f(n)` we want to examine. This is usually done by

comparing `f(n)` with some standard functions. The most common computing times are: `O(1), O(log2 n), O(n), O(n. log2 n), O(n^2), O(n^3), O(2^n), n! and n^n.`

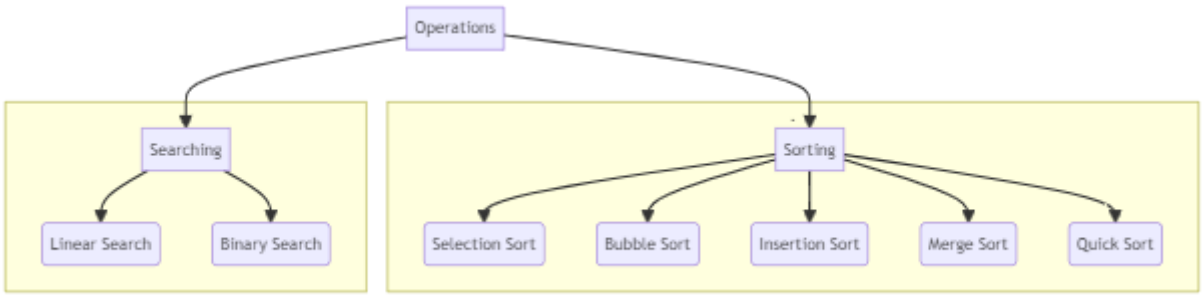
Numerical Comparison of Different Algorithms

Sr. No.	log n	n	n. log n	n^2^	n^3^	2^n^
1	0	1	0	1	1	1
2	1	2	2	4	8	4
3	2	4	8	16	64	16
4	3	8	24	64	512	256
5	4	16	64	256	4096	65536

Rules

1. If running time of any algo has additive/ substractive/ multiplicative/ divisive constant, then it can beneglected. `e.g. O(n+3) => O(n)`
2. If running time of any algo is having polynomial, then in its time complexity only leading term can be considered. `e.g. O( n^2 - n ) => O(n^2)` `O( n^3 + n + 3 ) => O(n^3)`

Operations



Searching

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

1. Linear or sequential search
2. Binary search

Linear Search/Sequential Search

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful. Suppose there are ‘n’ elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two

comparisons are necessary and so on. On an average you need [(n+1)/2] comparison's to search an element. If search is not successful, you would need 'n' comparisons.

Linear Search

arr	30	20	90	60	80	70	100	50	40	10
	0	1	2	3	4	5	6	7	8	9
							index			

		No. of comparisons		Running Time	Time Complexity
Key = 30	Best Case	1	if key is found at very first position in an array in only 1 comparison	O(1)	Ω(1)
Key = 10	Worst Case	10	if key is found at last position in n no. of comparisons	O(n)	O(n)
Key = 55	Worst Case	10	if key does not exists in an array	O(n)	O(n)
Key = 100	Avergae Case	N/2	if key is found at any inbetween position	O(n/2)	θ(n)
			if key is found at any position between 1-8 in above array		

Algorithm:

```
Algorithm LinearSearch(A, size, key)
{
    for (index = 1; index ⩽ size; index++)
    {
        if (key == A[index]) // Key is element we are searching for
            return true;
    }
    return false;
}
```

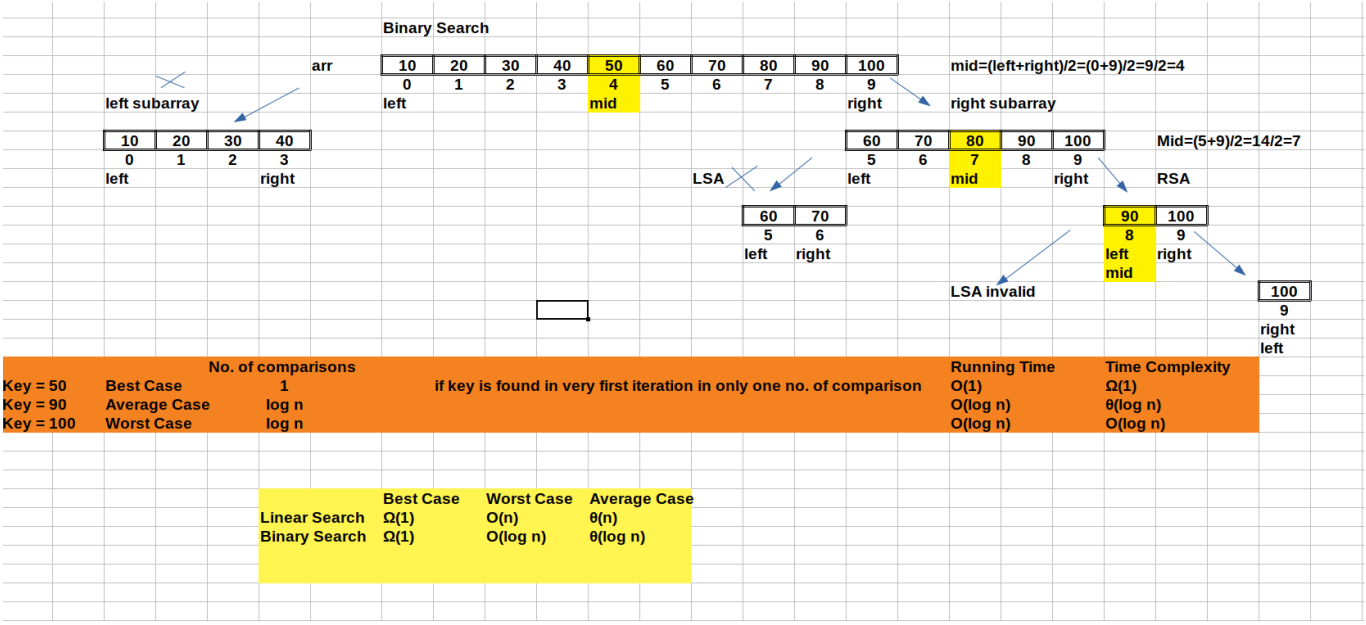
1. **Best Case:** If key is found at very first position in only 1 no. of comparison then it is considered as a best case and running time of an algorithm in this case is **O(1) = Ω(1)**
2. **Worst Case:** If either key is found at last position or key does not exists, maximum n no. of comparisons takes place, it is considered as a worst case and running time of an algorithm in this case is **O(n) = O(n)**
3. **Average Case:** If key is found at any in between position it is considered as an average case and running time of an algorithm in this case is **O(n/2) = θ(n)**

Binary Search/Logarithmic Search

This algorithm follows divide-and-conquer approach. To apply binary search on an array prerequisite is that array elements must be in a sorted manner.

- In Binary search we jump into the middle of the file, where we find key `a[mid]`, and compare 'x' with `a[mid]`.
  - If `x = a[mid]` then the desired record has been found.
  - If `x < a[mid]` then 'x' must be in that portion of the file that precedes `a[mid]`.
  - If `a[mid] > x`, then further search is only necessary in that part of the file which follows `a[mid]`.

If we use recursive procedure of finding the middle key `a[mid]` of the un-searched portion of a file, then every un-successful comparison of 'x' with `a[mid]` will eliminate roughly half the un-searched portion from consideration.



Algorithm:

```
Algorithm BinarySearch(A, key, n) // whereas A is an array of size "n"
{
    left = 1 right = n while (left ⩽ right) // till subarray is valid
    {
        mid = (left + right) / 2;
        if (key == A[mid])
            return true;
        if (key < A[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return false;
}
```

1. **Best Case:** if the key is found in very first iteration at mid position in only 1 no. of comparison it is considered as a best case and running time of an algorithm in this case is **O(1) = Ω(1)**.
2. **Worst Case:** if either key is not found or key is found at leaf position it is considered as a worst case and running time of an algorithm in this case is **O(log n) = O(log n)**.
3. **Average Case:** if key is not found in the first iteration and it is found at non-leaf position it is considered as an average case and running time of an algorithm in this case is **O(log n) = θ(log n)**.

- While comparing two algo's, we always decides effieciency of an algo depends on worst case time complexity. worst case time complexity of linear search = `O(n)` worst case time complexity of binary search = `O(log n)` **`O(log n) < O(n)` binary search algo is efficient than linear search**

Sorting

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words is arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order.

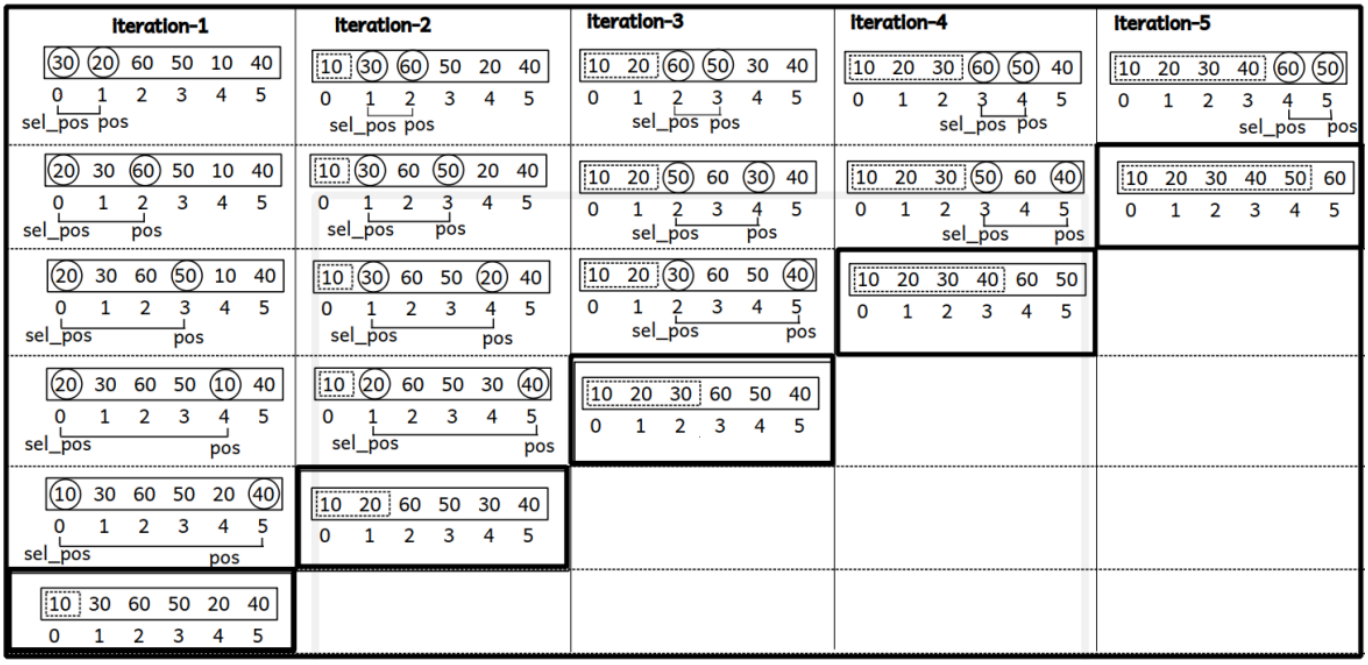
1. Selection sort
2. Bubble sort
3. Insertion sort

Selection Sort

Best Case : Ω(n^2^) Worst Case : O(n^2^) Average Case : θ(n^2^)

Selection sort will not require no more than `n-1` interchanges. Suppose x is an array of size n stored in memory. The selection sort algorithm first selects the smallest element in the array x and place it at array position 0; then it selects the next smallest element in the array x and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array. The array is passed through `(n-1)` times and the smallest element is placed in its respective position in the array as detailed below:

- Iteration 1: Find the location j of the smallest element in the array `x[0]`, `x[1]`, `x[n-1]`,and then interchange `x[j]` with `x[0]`. Then `x[0]` is sorted.
- Iteration 2: Leave the first element and find the location j of the smallest element in the sub-array `x[1], x[2], . . . . x[n-1]`, and then interchange `x[1]` with `x[j]`. Then `x[0]`,`x[1]` are sorted.
- Iteration 3: Leave the first two elements and find the location j of the smallest element in the sub-array `x[2], x[3], . . . . x[n-1]`, and then interchange `x[2]` with `x[j]`. Then `x[0], x[1], x[2]` are sorted.
- Iteration (n-1): Find the location j of the smaller of the elements `x[n-2]` and `x[n-1]`, and then interchange `x[j]` and `x[n-2]`. Then `x[0]`, `x[1], . . . . x[n-2]` are sorted. Of course, during this pass `x[n-1]` will be the biggest element and so the entire array is sorted.



- iteration 1: `O(n-1) -> O(n)`
- iteration 2: `O(n-2) -> O(n)`

- iteration (n-1): -> `O(n)` total no. of comparisons = `n(n-1) = O( n^2 - n ) => O(n^2)`

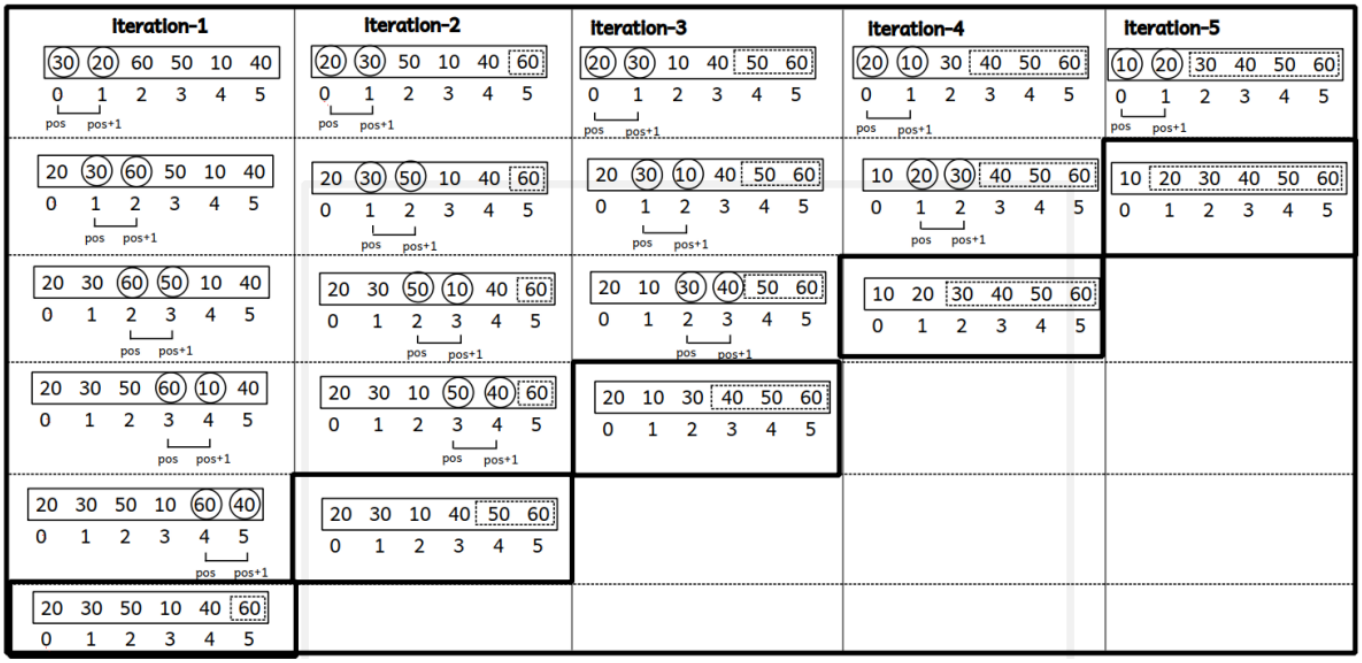
Algorithm:

```
Algorithm SelectionSort(A, n){
    for (sel_pos = 1; sel_pos < n; sel_pos++){ // O(n)
        for (pos = sel_pos + 1; pos ⩽ n; pos++){ // O(n*n)
            if (A[sel_pos] > A[pos])
                SWAP(A[sel_pos], A[pos]);
        }
    }
}
```

Bubble Sort

Best Case : Ω(n) -> it occures only if all array elements are already sorted **Worst Case : O(n^2^) Average Case: θ(n^2^)**

The basic idea of bubble sort is to iterate through the file sequentially several times. In each iteration, we compare each element in the file with its successor `i.e., X[i]` with `X[i+1]` and interchange two element when they are not in proper order. By this logic in first iteration largest element gets setteled at last position, in second iteration second largest element gets setteled at second last position and so on, in max `(n-1)` no. of iterations all elements gets arranged in a sorted manner. Bubble sort is also called as exchange sort.



- This algo is also called as "sinking sort".
- Selection sort & bubble sort algo's are simple but these algo's are not efficient for larger input size array.

Algorithm:

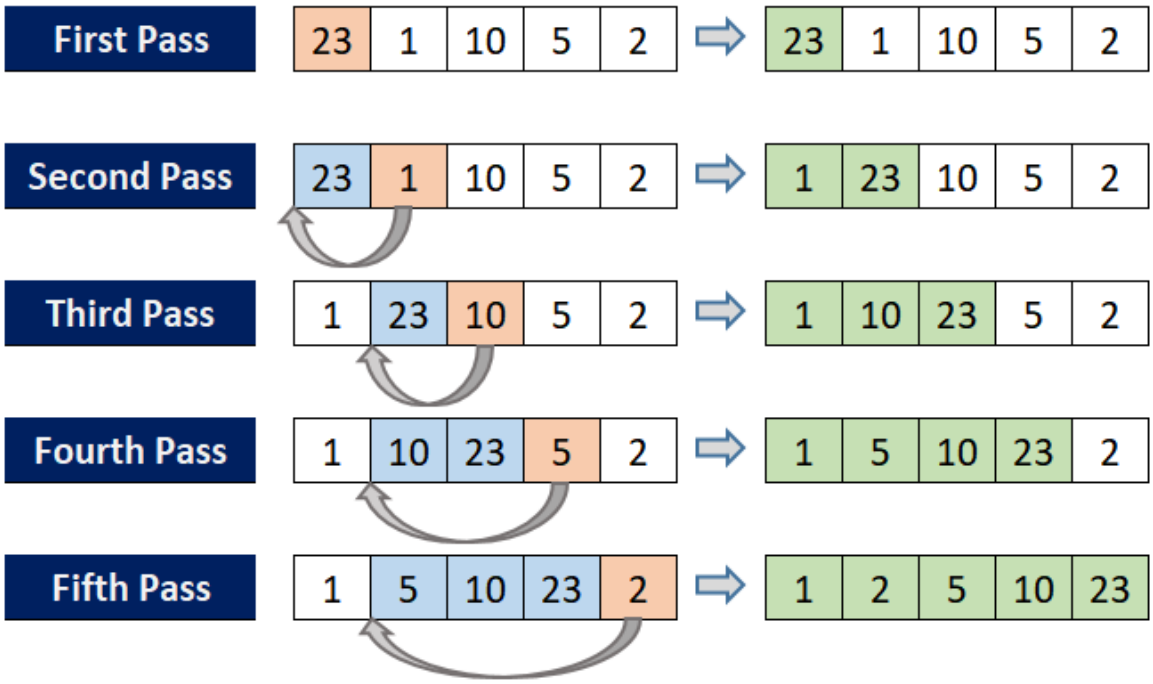
```
Algorithm BubbleSort(A, n){
    for (pass = 0; pass < n - 1; pass++){ // O(n)
        for (i = 0; i < n - 1 - pass; i++){ // O(n*n)
            if (A[i] > A[i + 1])
                SWAP(A[i], A[i + 1]);
        }
    }
}
```

Insertion Sort

Best Case : Ω(n) -> if array elements are already arranged in a sorted manner. **Worst Case : O(n^2^) Average Case: θ(n^2^)**

Insertion Sort is Similar to arranging playing cards in left hand

1. Divide into Sorted and Unsorted Regions:
    - The array is divided into two regions: the left part is sorted, and the right part is unsorted.
    - Initially, the left part (sorted region) contains the first element, and the right part (unsorted region) contains the rest.
2. Iterate through the Unsorted Region:
    - For each element in the unsorted region, compare it with elements in the sorted region.
    - Move elements in the sorted region that are greater than the current element to the right.
3. Insert the Element:
    - Insert the current element into its correct position in the sorted region.
4. Repeat:
    - Repeat steps 2 and 3 until all elements are in the sorted region.



- Insertion sort algortihm is an efficient algorithm for smaller input size array.

Algorithm:

```
Algorithm InsertionSort(A, n){
    for (i = 1; i < n; i++){
        key = A[i];
        j = i - 1;
        // Move elements greater than key to the right
        while (j ⩾ 0 && A[j] > key){
            A[j + 1] = A[j];
            j = j - 1;
        }
        // Insert key into its correct position
        A[j + 1] = key;
    }
}
```

Merge Sort

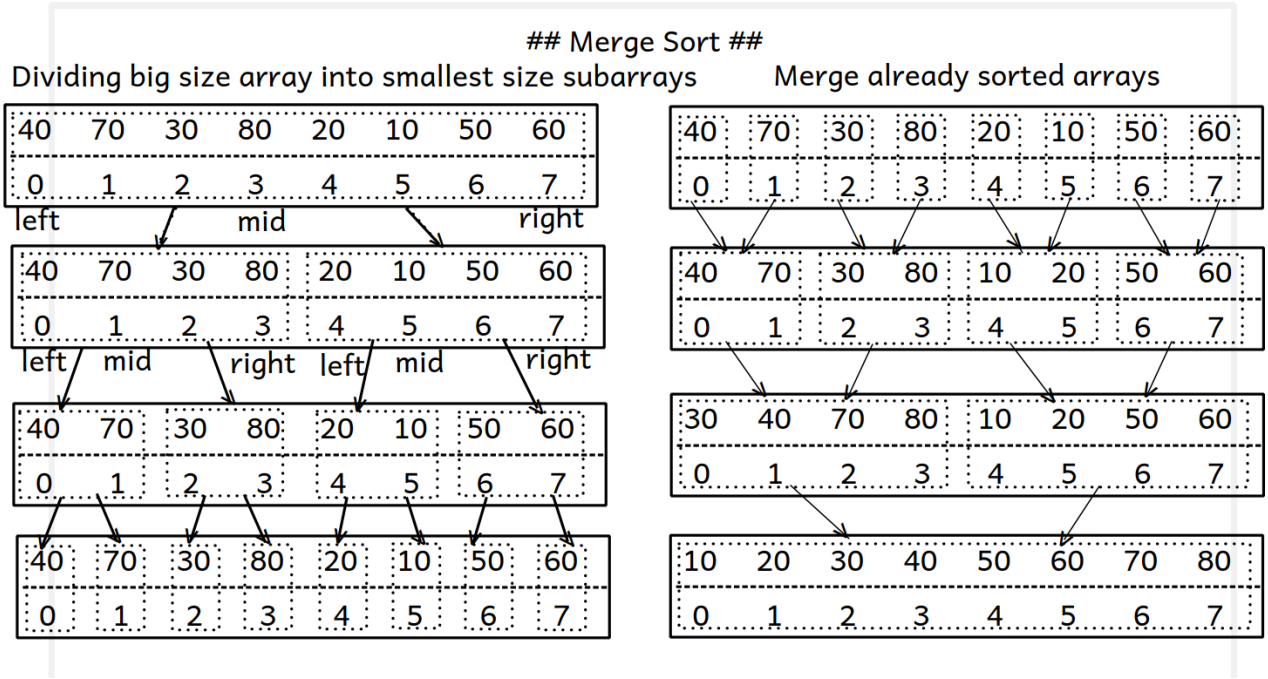
Best Case : Ω(n log n) Worst Case : O(n log n) Average Case: θ(n log n)

Merge Sort is a divide-and-conquer algorithm that follows these steps to sort an array or list:

1. Divide: The array is divided into two halves. This process is repeated recursively until each sub-array contains only one element.

2. Conquer: The two halves are then sorted individually using the same sorting algorithm.

3. Merge: The sorted sub-arrays are merged to produce a single sorted array. During the merging process, elements are compared and arranged in the correct order.



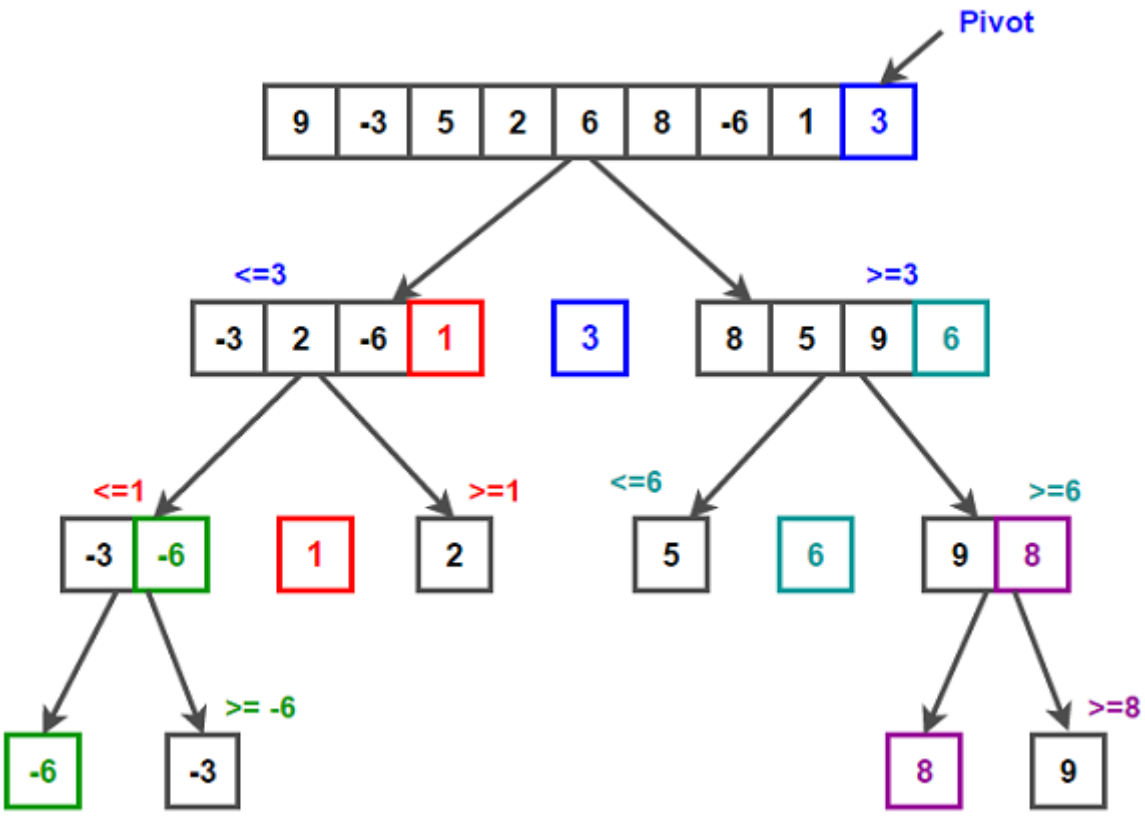

```
Algorithm MergeSort(arr, left, right){
    if (left < right){
        // Find the middle point to divide the array
        middle = (left + right) / 2;

        // Recursively sort the first and second halves
        MergeSort(arr, left, middle);
        MergeSort(arr, middle + 1, right);

        // Merge the sorted halves
        Merge(arr, left, middle, right);
    }
}
Algorithm Merge(arr, left, middle, right){
    // Calculate sizes of two subarrays to be merged
    n1 = middle - left + 1;
    n2 = right - middle;

    // Create temporary arrays to hold the data
    leftArray[1..n1] = arr[left..middle];
    rightArray[1..n2] = arr[middle + 1..right];

    // Merge the two arrays
    i = 1; // Initial index of first subarray
    j = 1; // Initial index of second subarray
    k = left; // Initial index of merged subarray

    while (i ⩽ n1 && j ⩽ n2){
        if (leftArray[i] ⩽ rightArray[j]){
```

```
            arr[k] = leftArray[i];
            i++;
        }
        else{
            arr[k] = rightArray[j];
            j++;
        }
        k++;
    }
    // Copy remaining elements of leftArray[] if any
    while (i <= n1){
        arr[k] = leftArray[i];
        i++;
        k++;
    }
    // Copy remaining elements of rightArray[] if any
    while (j <= n2){
        arr[k] = rightArray[j];
        j++;
        k++;
    }
}
```

Quick Sort

Best Case : Ω(n log n) Worst Case : O(n^2^) Average Case : θ(n log n)

- pivot element is selected.
- Elements which are smaller than pivot are shifted towards left side of pivot, and elements which are greater than pivot are shifted towards right side of pivot.
- Elements which are at left side of pivot will be referred as "left partition", whereas elements which are at right side of pivot will be referred as "right partition", and pivot ele gets settled at its appropriate position
- Apply partitioning further on left partition as well on right partition, till size of subarray is greater than 1.



Algorithm:

```
Algorithm QuickSort(arr, left, right)
{
    pivot = arr[left]; // Choose pivot element from the array
    // Initialize pointers for partitioning
    i = left;
    j = right;
    // Partition the array
    while (i < j)
    {
        // Find an element on the left side greater than the pivot
        while (i ⩽ right && arr[i] ⩽ pivot)
            i++;
        // Find an element on the right side smaller than or equal to the
pivot
        while (arr[j] > pivot)
            j--;
        // Swap elements to ensure the left side has smaller elements and
the right side has larger elements
        if (i ⩽ j)
        {
            SWAP(arr[i], arr[j]);
        }
    }
    // Swap the pivot to its correct position in the sorted array
    SWAP(arr[j], arr[left]);
}
```

	Algorithm	Best Case	Worst Case	Average Case	
1	Selection Sort	$\Omega(n^2)$	$O(n^2)$	$\theta(n^2)$	
2	Bubble Sort	$\Omega(n)$	$O(n^2)$	$\theta(n^2)$	Best Case occure if array ele's are already sorted – can be achieved at an implemetation level
3	Insertion Sort	$\Omega(n)$	$O(n^2)$	$\theta(n^2)$	Best Case occure if array ele's are already sorted – efficient by design
4	Merge Sort	$\Omega(n \log n)$	$O(n \log n)$	$\theta(n \log n)$	This algo takes extra space on an array
5	Quick Sort	$\Omega(n \log n)$	$O(n^2)$	$\theta(n \log n)$	worst case in quick sort rarely occures – if array ele's are exists exactly in a reverse order

Insertion Sort	efficient for smaller input size array
Quick Sort	efficient for larger input size array

Limitations of an Array

- Array is a collection of logically related similar elements
- Array is "static", i.e. size of an array cannot either grow or shrink during runtime.
  - `int arr[ 100 ];`
  - max we can store 100 elements into above arr
  - If we store 95 elements into above array -> `5*4=20` bytes memory gets wasted
- Addition & deletion operations on array are not efficient as it takes `O(n)` time.

To overcome above two limitations of an array data structure, "linked list" data structure has beed design

- Linked list data structure have:
  - Dynamic nature
  - Addition & deletion operations are efficient - `O(1)` time

Size of an array is fixed	**Size of a list is not fixed**
Memory is allocated from **stack**	Memory is allocated from **heap**
Specify the number of elements during declaration **(compile time)**	Not necessary to specify the number of elements during declaration **(memory allocated during run time)**
Occupies **less memory** than a linked list for the same number of elements	Occupies **more memory**
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room	Inserting a new element at any position can be carried out easily
Deleting an element from an array is not possible	Deleting an element is possible

Linked List Concepts

- A linked list is a **non-sequential** collection of data items. It is a **dynamic** data structure.
- For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.
- The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as **each data item contains the address of the next data**

item.

Advantages of linked lists:

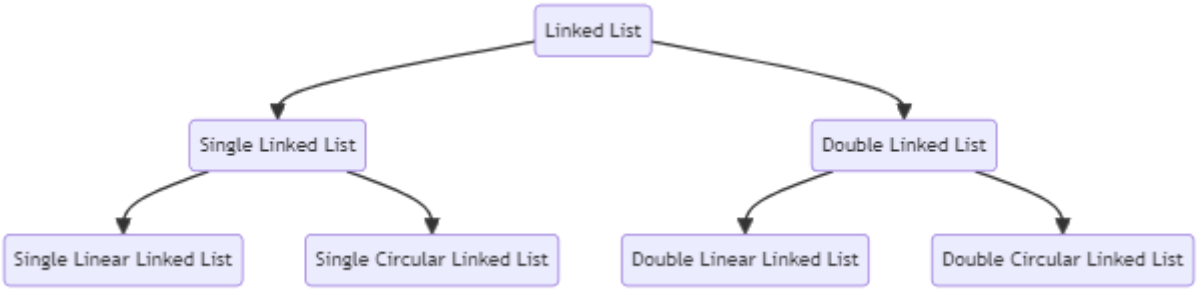
1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre-allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists

1. **Singly Linked List:** it is type of linked list in which each element/node contains an address of its next node.
    - Singly linked list furter divided into two types:
        1. Singly Linear Linked List
        2. Singly Circular Linked List
2. **Doubly Linked List:** it is type of linked list in which each element/node contains an address of its next node as well as an address of its previous node.
    - Doubly linked list furter divided into two types:
        1. Doubly Linear Linked List
        2. Doubly Circular Linked List



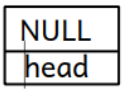
Singly Linear Linked List

A linked list allocates space for each element separately in its own block of memory called a **"node"**.

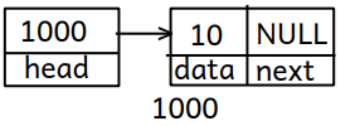
- The front of the list is a pointer to the **head**, it always contains address of first node, if list is not empty
- Each node contains two fields;
    - a **"data"** field to store whatever element, and
    - a **"next"** field which is a pointer used to link to the next node. Last node's next part points to `NULL`.

- Each node is allocated in the heap using `malloc()`, so the node memory continues to exist until it is explicitly de-allocated using `free()`.

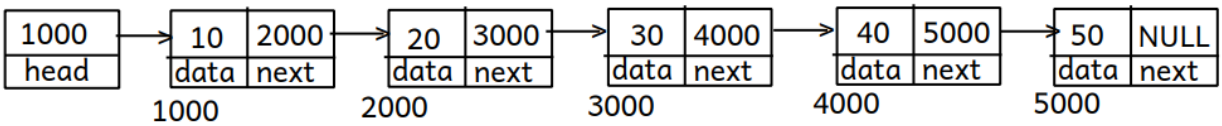
1) singly linear linked list --> list is empty



2) singly linear linked list --> list contains only one node



3) singly linear linked list --> list contains more than one nodes



- Basically we can perform addition and deletion operations onto the linked list

1. **Addition:** We can add node into the singly linear linked list by 3 ways:
    1. Add node into the list at last position
    2. Add node into the list at first position
    3. Add node into the list at specific position
2. **Deletion:** We can delete node into the singly linear linked list by 3 ways:
    1. Delete node at last position
    2. Delete node at first position
    3. Delete node at specific position

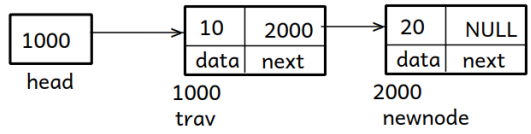
Add node into the list at last position

To add a node to the last position in a singly linked list, you need to follow these steps:

1. Create a New Node:
    - Allocate memory for a new node.
    - Set the data of the new node with the value you want to add.
    - Set the next pointer of the new node to `NULL` since it will be the last node.
2. Traverse the List:
    - Start at the head of the linked list.
    - Traverse the list until you reach the last node. You can do this by checking if the next pointer of the current node is `NULL`.
3. Attach the New Node:

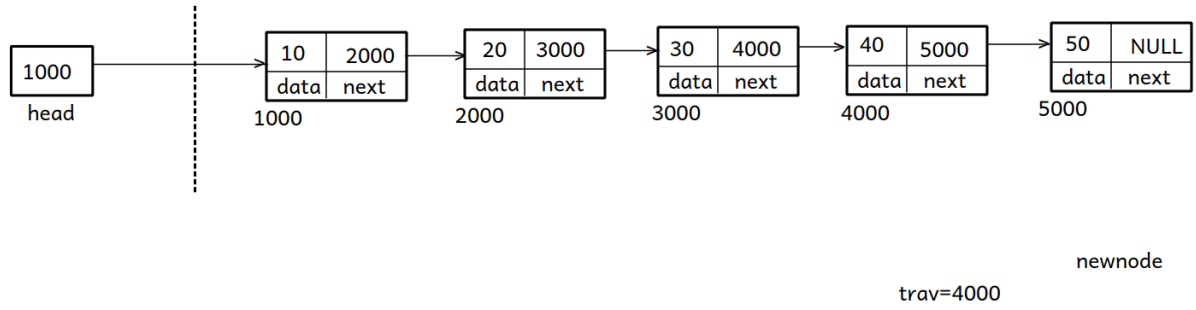
- Once you reach the last node, update the next pointer of the last node to point to the new node you created in step 1.

add node into slll at last position: if list is empty



---

add node into slll at last position: if list is not empty



Here's a simple example in C to illustrate:

```c
typedef struct node{
    int data;//4 bytes
    struct node *next;//self-referential pointer variable - 4 bytes
}node_t;

//create an empty list
node_t *head = NULL;

void add_node_at_last_position(int data){
    //1. create a newnode
    node_t *newnode = create_node(data);

    //2. if list is empty -- attach newly created node to the head
    if( head == NULL ){
        //store an addr of newly created node into the head pointer
        head = newnode;
    }
    else{//if list is not empty

        //start traversal from the first node
        node_t *trav = head;

        //traverse the list till last node
        while( trav->next != NULL ){
            trav = trav->next;//moves trav pointer to its next node
        }
```

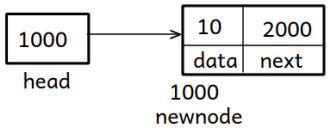
```
        //attach newly created node to the last node
        //i.e. store an addr of newly created node into the next part of
last node
        trav→next = newnode;
    }
}

node_t *create_node(int data){
    //1. allocate memory dynamically for a node
    node_t *temp = (node_t *)malloc(sizeof(node_t));

    //2. initialize members of the node
    temp→data = data;
    temp→next = NULL;

    //3. return starting addr of newly created node to the calling function
    return temp;
}
```

Add node into the list at first position

To add a node to the first position in a singly linked list, follow these steps:

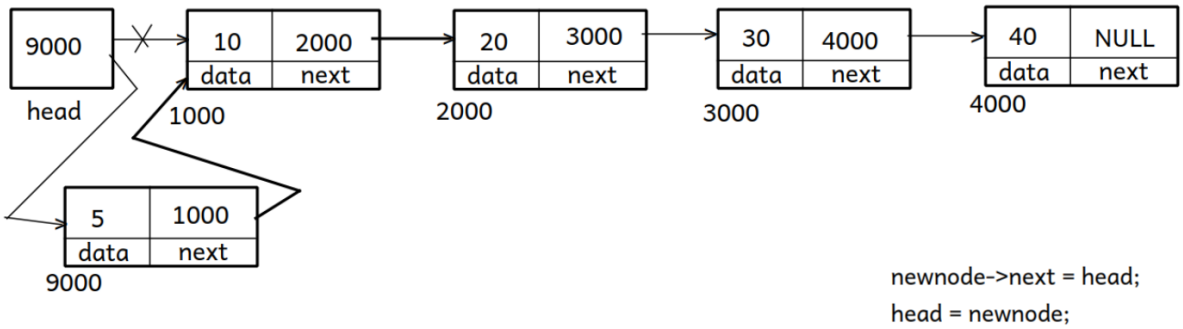
1. Create a New Node:
    - Allocate memory for a new node.
    - Set the data of the new node with the value you want to add.
    - Set the next pointer of the new node to the current head of the list.
2. Update Head Pointer:
    - Update the head pointer to point to the new node.

add node into slll at first position: if list is empty



---

add node into slll at first position: if list is not empty



Here's a simple example in C to illustrate:

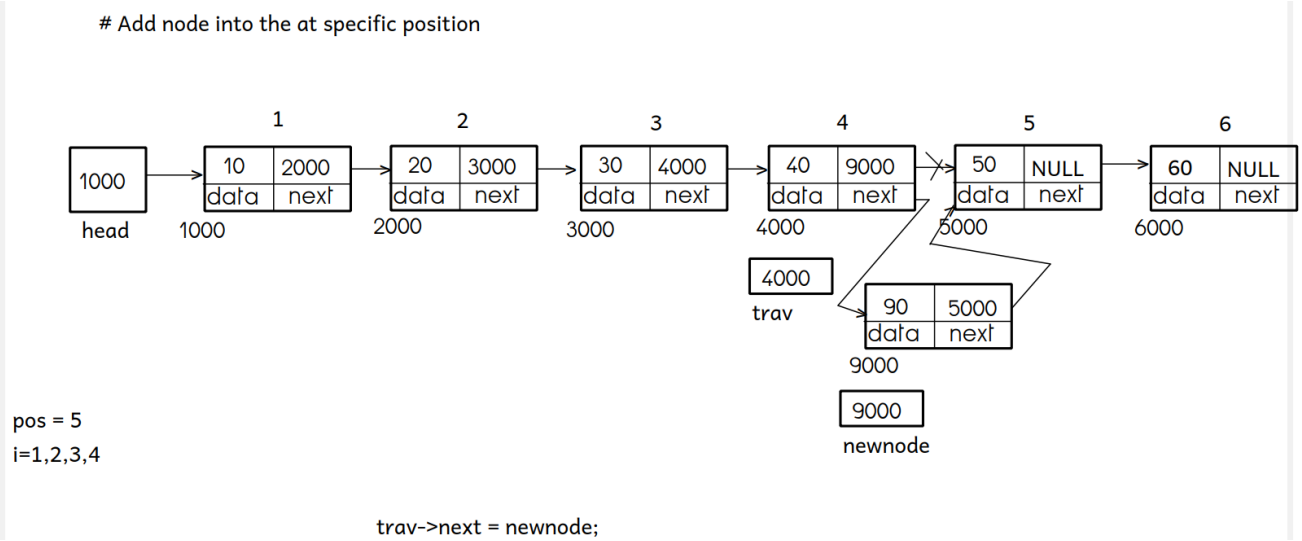
```c
typedef struct node{
    int data;//4 bytes
    struct node *next;//self-referential pointer variable - 4 bytes
}node_t;

//create an empty list
node_t *head = NULL;

void add_node_at_first_position(int data){
    //1. create a newnode
    node_t *newnode = create_node(data);

    //2. if list is empty -- attach newly created node to the head
    if( head == NULL ){
        //store an addr of newly created node into the head pointer
        head = newnode;
    }
    else{//if list is not empty
        //store an addr of cur first node into the next part of newly
created node
        newnode→next = head;
        //attach newly created node to the head
        head = newnode;
    }
}
node_t *create_node(int data){
    //1. allocate memory dynamically for a node
    node_t *temp = (node_t *)malloc(sizeof(node_t));
```

```
    //2. initialize members of the node
    temp→data = data;
    temp→next = NULL;

    //3. return starting addr of newly created node to the calling function
    return temp;
}
```

Add node into the list at specific position

To add a node to a specific position in a singly linked list, follow these steps:

1. Create a New Node:
    - Allocate memory for a new node.
    - Set the data of the new node with the value you want to add.
2. Traverse to the Desired Position:
    - Traverse the linked list until you reach the node just before the desired position. Keep track of the current node and the next node.
3. Update Pointers:
    - Set the next pointer of the new node to the next node in the list.
    - Set the next pointer of the current node to the new node.



Here's a simple example in C to illustrate:

```
// node structure
typedef struct node{
    int data;          // 4 bytes
    struct node *next; // self-referential pointer variable - 4 bytes
} node_t;
```

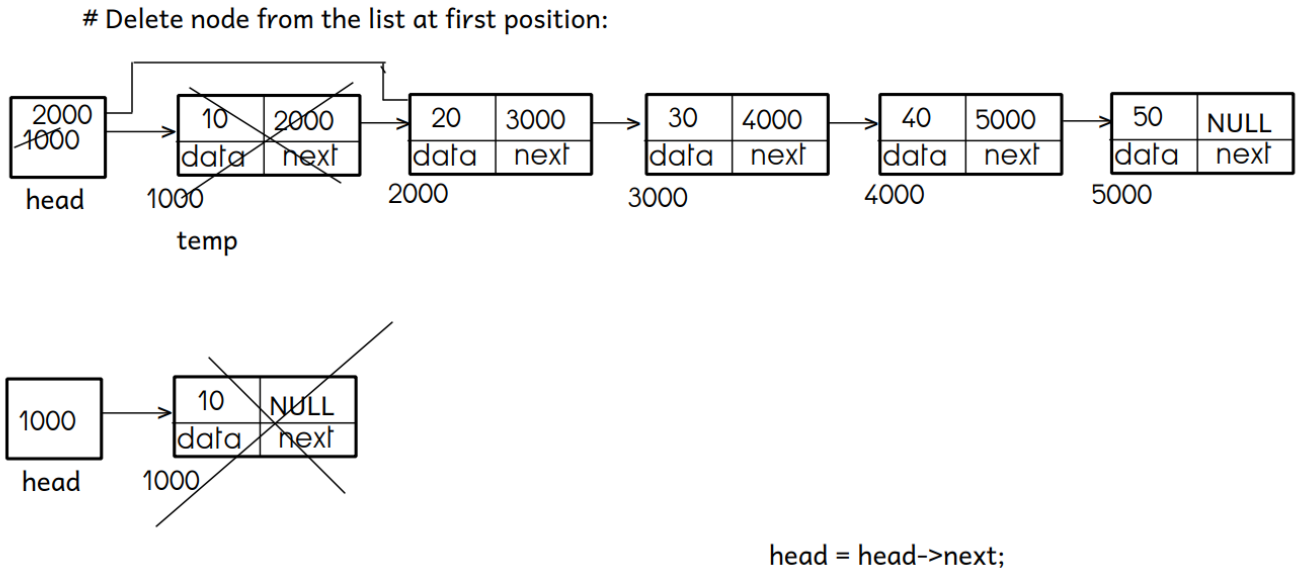
```c
// create an empty list
node_t *head = NULL;

int count_nodes(void){
    int cnt = 0;

    // if list is not empty
    if (head != NULL){
        node_t *trav = head;
        while (trav != NULL){
            cnt++;
            trav = trav->next;
        }
    }
    return cnt;
}

node_t *create_node(int data){
    // 1. allocate memory dynamically for a node
    node_t *temp = (node_t *)malloc(sizeof(node_t));
    if (temp == NULL){
        printf("malloc() failed !!!\n");
        exit(1);
    }
    // 2. initialize members of the node
    temp->data = data;
    temp->next = NULL;

    // 3. return starting addr of newly created node to the calling
function
    return temp;
}

void add_node_at_specific_position(int data, int pos){
    if (pos == 1) // position the first position
        add_node_at_first_position(data);
    else if (pos == count_nodes() + 1) // if pos is the last position
        add_node_at_last_position(data);
    else{ // if position is in between position

        // create a newnode
        node_t *newnode = create_node(data);
        // start traversal from the first node
        node_t *trav = head;
        int i = 1;

        // traverse the list till (pos-1)th node
        while (i < pos - 1){
            i++;
            trav = trav->next;
        }
        // store an addr of cur (pos)th node into the next part of newly
created node
```

```
        newnode→next = trav→next;
        // store an addr of newly created node into the next part (pos-1)th
node
        trav→next = newnode;
    }
}
```

Delete node at first position

To delete a node at the first position in a singly linked list, you need to perform the following steps:

1. Check if the list is empty.
2. If the list is not empty, update the head pointer to point to the second node (the node currently at the second position).
3. Free the memory allocated for the node that was at the first position.



Here's a simple example in C to illustrate:

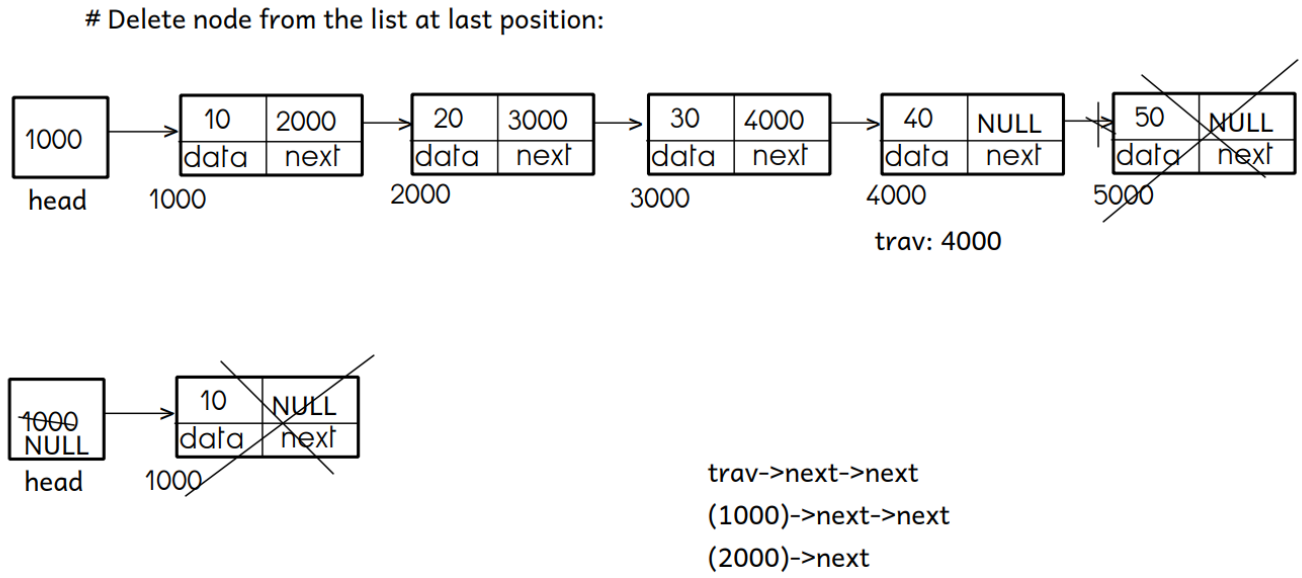
```
void delete_node_at_first_position(void){
    // check list is not empty
    if (head != NULL){
        // if list contains only one node
        if (head→next == NULL){
            // delete the node and make head as `NULL`
            free(head);
            head = NULL;
        }
        else{
```

```
            // store an addr of cur first node into temp which is to be
deleted
            node_t *temp = head;
            // attach cur second node to the head
            head = head→next;
            // delete the node
            free(temp);
            temp = NULL;
        }
    }
    else{
        printf("list is empty !!!\n");
    }
}
```

Delete node at last position

To delete a node at the last position in a singly linked list, you need to traverse the list to the second-to-last node and update its next pointer to `NULL`. Here's the general algorithm:

1. If the list is empty or contains only one node, set the head to `NULL`.
2. Otherwise, traverse the list until you reach the second-to-last node.
3. Update the next pointer of the second-to-last node to `NULL`.
4. Free the memory allocated for the last node.



Here's a simple example in C to illustrate:

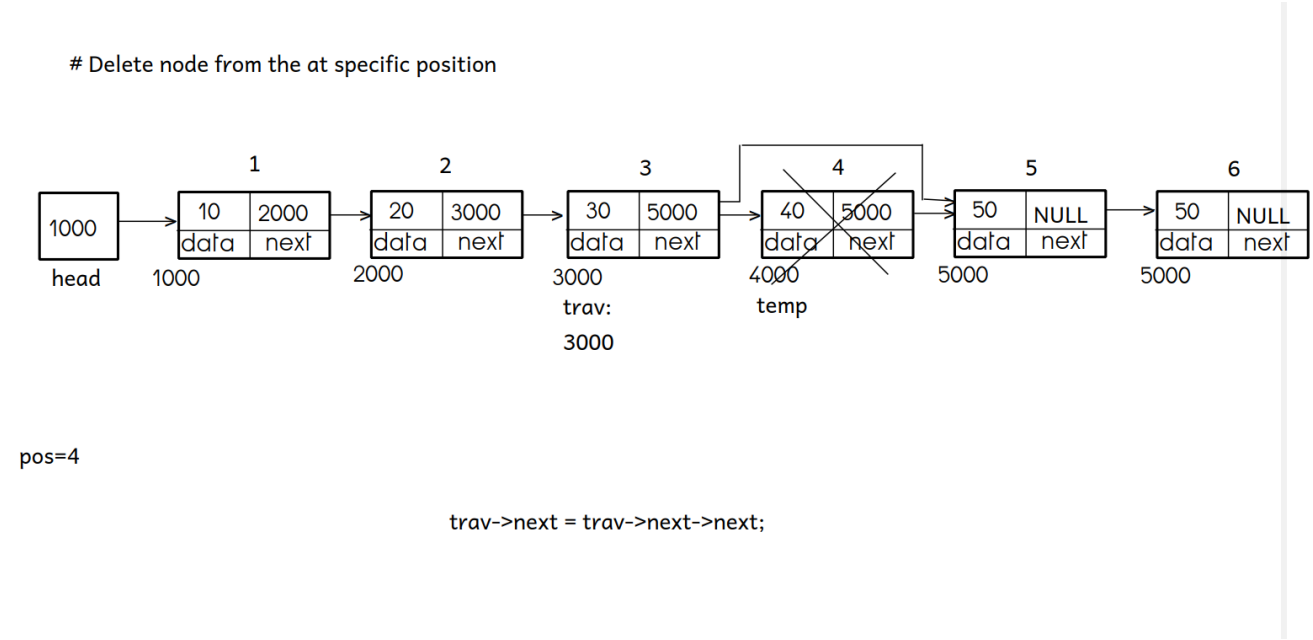
```c
void delete_node_at_last_position(void){
    // check list is not empty
    if (head != NULL){
        // if list contains only one node
        if (head->next == NULL){
            // delete the node and make head as `NULL`
            free(head);
            head = NULL;
        }
        else{ // if list contains more than one node
            // start traversal from first node
            node_t *trav = head;
            // traverse the list till second last node
            while (trav->next->next != NULL)
                trav = trav->next;

            // delete last node
            free(trav->next);
            // make next part of cur second last node as `NULL`
            trav->next = NULL;
        }
    }
    else{
        printf("list is empty !!!\n");
    }
}
```

Delete node at Specific Position

To delete a node at a specific position in a singly linked list, you need to traverse the list until you reach the node just before the target position, and then update the next pointer of that node to skip the node at the target position. Additionally, you should free the memory of the node being deleted.

Here's the general algorithm:

1. If the list is empty, do nothing.
2. If the target position is the first node, update the head to point to the second node (if it exists).
3. Otherwise, traverse the list until you reach the node just before the target position.
4. Update the next pointer of the preceding node to skip the node at the target position.
5. Free the memory of the node at the target position.



Here's a simple example in C to illustrate:

```c
void add_node_at_specific_position(int data, int pos)
{
    if (pos == 1) // position the first position
        add_node_at_first_position(data);
    else if (pos == count_nodes() + 1) // if pos is the last position
        add_node_at_last_position(data);
    else{ // if position is in between position
        // create a newnode
        node_t *newnode = create_node(data);
        // start traversal from the first node
        node_t *trav = head;
        int i = 1;

        // traverse the list till (pos-1)th node
        while (i < pos - 1){
            i++;
            trav = trav->next;
        }
        // store an addr of cur (pos)th node into the next part of newly
created node
        newnode->next = trav->next;
        // store an addr of newly created node into the next part (pos-1)th
node
        trav->next = newnode;
    }
}
```

Limitations of Singly Linear Linked List

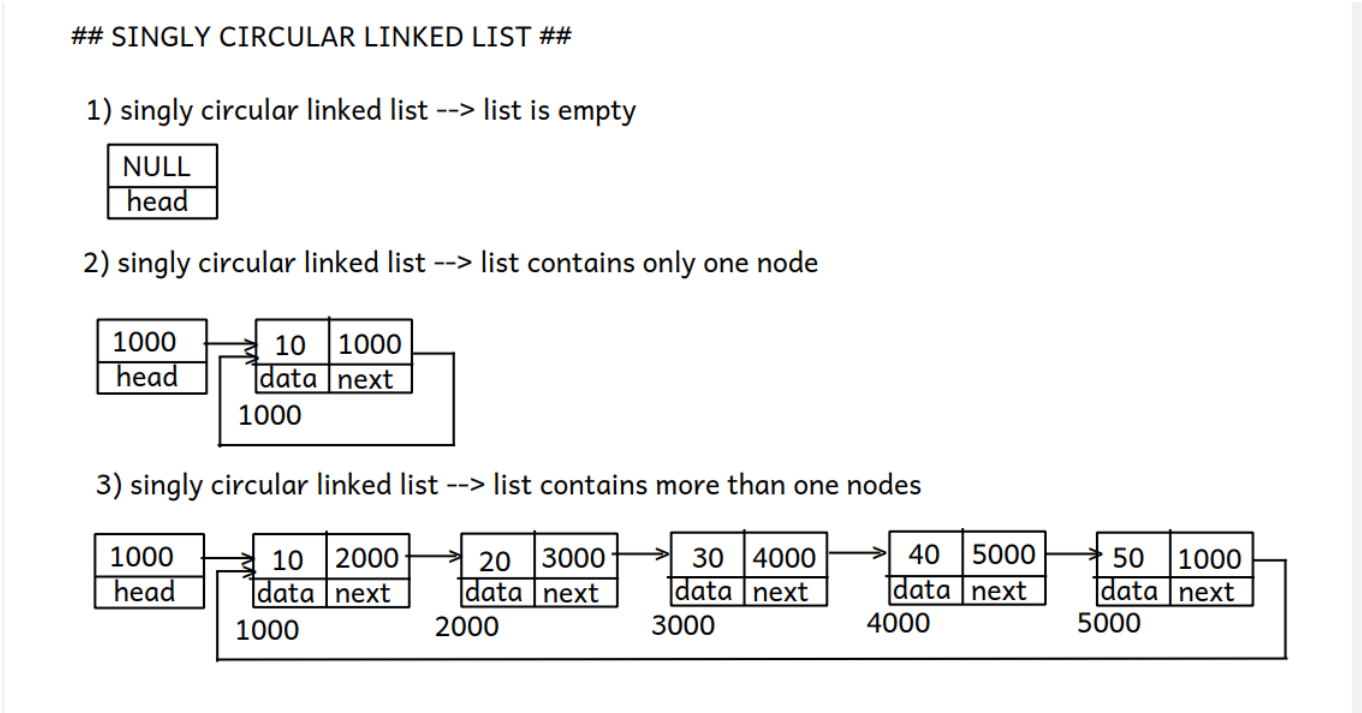
Singly linked lists have several limitations, which influence their suitability for different scenarios. Here are some of the main limitations:

1. Extra Pointer Storage
2. Traversal is difficult
3. Reverse traversing causes memory wastage
4. Issues with random access
5. `add_last` & `delete_last` operations are not effcient as it takes `O(n)` time
6. We can start travesal from first node, and list can be traversed only in a forward direction.
7. We cannot access prev node of any node from it
8. We cannot revisit any node

- To overcome this limitation "Singly Circular Linked List" has been designed.

Singly circular Linked List

- It is just a single linked list in which the link field of the last node points back to the address of the first node.
- A circular linked list has no beginning and no end.
- It is necessary to establish a special pointer called head pointer always pointing to the first node of the list.
- Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement.
- In circular linked list no `NULL` pointers are used, hence all pointers contain valid address.




```c
typedef struct node{
    int data;
    struct node *next;
} node_t;
```

```c
node_t *head = NULL;

void display_list(void){
    if (head != NULL){
        node_t *trav = head;
        printf("list is : ");

        do{
            printf("%4d", trav->data);
            trav = trav->next;
        } while (trav != head);

        printf("\n");
    }
    else{
        printf("list is empty !!!\n");
    }
}

node_t *create_node(int data){
    node_t *temp = (node_t *)malloc(sizeof(node_t));
    if (temp == NULL){
        printf("malloc() failed !!!\n");
        exit(1);
    }
    temp->data = data;
    temp->next = NULL;
    return temp;
}

void add_node_at_last_position(int data){
    // create a new node
    node_t *newnode = create_node(data);

    // if list is empty
    if (head == NULL){
        head = newnode;
        newnode->next = head;
    }
    else{ // if list is not empty
        // start traversal from the first node
        node_t *trav = head;

        // traverse the list till last node
        while (trav->next != head)
            trav = trav->next;

        // attach newly created node to the last node
        trav->next = newnode;
        // store an addr of first node into next part of newly created node
        newnode->next = head;
    }
}
```

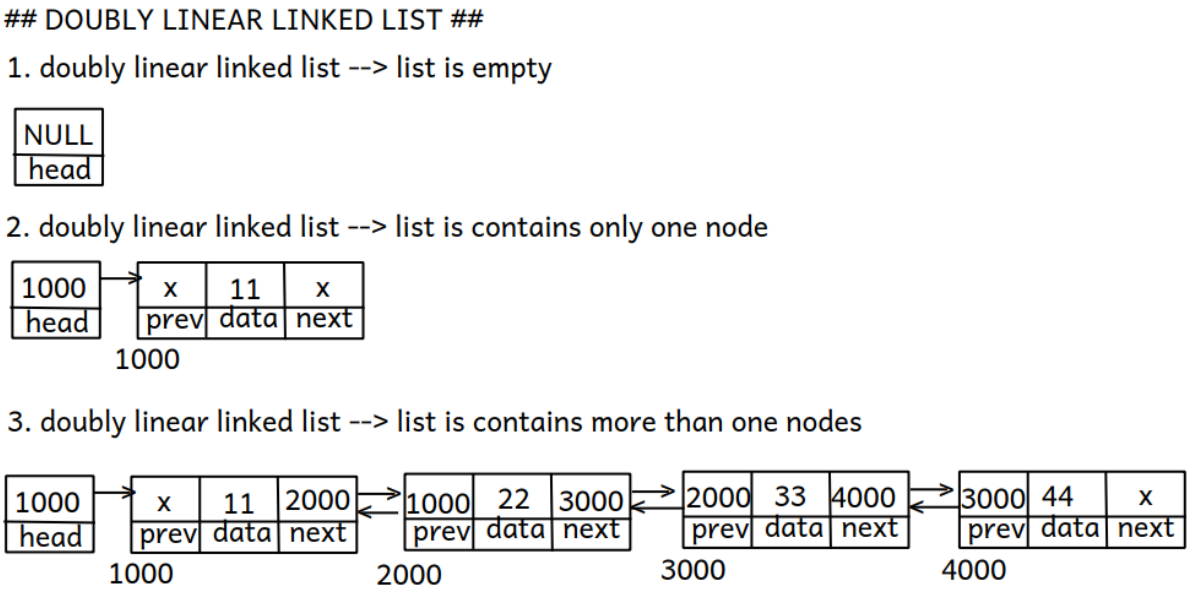
Limitations of Singly Circular Linked List

- Add last, delete last & add first, delete first operations are not efficient as it takes `O(n)` time.
- We can starts traversal only from first node and can traverse the SCLL only in a forward direction.
- Previous node of any node cannot be accesed from it
- To overcome these limitation Doubly Linear Linked List has been designed.

Doubly Linear Linked List

A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

- **Left link(prev):** left link points to the predecessor node
- **Data:** right link points to the successor node
- **Right link(next):** The data field stores the required data.
- Prev part of first node and next part of last node contains `NULL`.



Limitations of Doubly Linear Linked List

In DLLL, `add_last()` & `delete_last()` functions takes `O(n)` time, and hence DCLL has been designed.

Doubly Circular Linked List

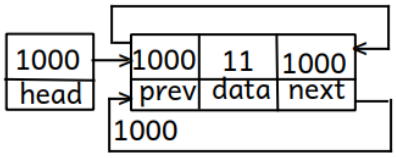
A circular double linked list has both successor pointer and predecessor pointer in circular manner. The objective behind considering circular double linked list is to simplify the insertion and deletion operations performed on double linked list.

- Prev part of first node contains of addr of last node, and next part of last node contains an addr of the first node.

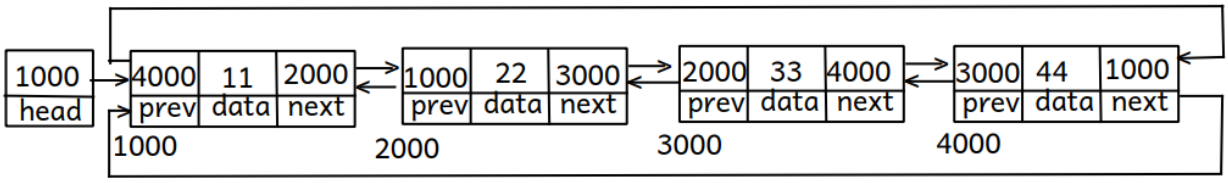
DOUBLY CIRCULAR LINKED LIST

1. doubly circular linked list --> list is empty

2. doubly circular linked list -> list is contains only one node



3. doubly circular linked list --> list is contains more than one nodes



- DCLL: `add_last()`, `add_first()`, `delete_first()` & `delete_last()` operations takes `O(1)` time.

Difference between array and linked list

1. **Static vs. Dynamic:**
    - **Arrays are static** data structures. Once you define the size of an array, it remains fixed throughout the program execution.
    - **Linked lists are dynamic** data structures. They can grow or shrink in size during runtime, making them more flexible.
2. **Efficiency of Addition and Deletion:**
    - Addition and deletion operations in arrays have a time complexity of `O(n)`, where n is the size of the array. This is because elements may need to be shifted to accommodate the changes.
    - Linked lists, especially singly linked lists, have `O(1)` time complexity for insertion and deletion at the beginning or end. For operations in the middle, it's `O(n)` due to traversal.
3. **Memory Allocation:**
    - Array elements are typically stored in the **stack** section of the main memory.
    - Linked list elements are dynamically allocated and stored in the **heap** section of memory.
4. **Access Time:**
    - Accessing elements in an array is more efficient than in a linked list. Arrays support **direct or random access** using the index, `i.e., O(1)` time complexity.
    - **Linked lists require sequential access**, starting from the head, to reach a specific element. This results in `O(n)` time complexity for accessing an arbitrary element.

5. **Memory Usage:**
    - Storing n elements in an **array may take less space** than storing the same number of elements in a linked list. This is because, in a linked list, **additional space is needed for storing pointers or references.**
6. **Searching Operation:**
    - Searching can be more efficient in arrays, especially if they are sorted. Binary search, which has a time complexity of `O(log n)`.
    - Linked lists do not support efficient binary search, and their searching typically takes `O(n)` time.
7. **Link Maintaining**
    - In an array link between elements is maintained by the compiler.
    - In a linked list there is need to maintains link between elements

In summary, the choice between arrays and linked lists depends on the specific requirements of the task at hand. Arrays are more suitable for scenarios where random access and efficient search are crucial, while linked lists offer advantages in dynamic situations where frequent insertions and deletions are performed.

Applications of Linked Lists

Certainly! Linked lists are versatile data structures and find applications in various domains. Here are additional applications of linked lists:

1. **Memory Management in Operating Systems:**
    - Linked lists are often used in memory management algorithms in operating systems. They can represent free blocks of memory in a dynamic allocation system.
2. **Undo Functionality in Text Editors:**
    - Text editors often use linked lists to implement the undo functionality. Each edit operation is stored as a node in a linked list, allowing for easy traversal and reversal of changes.
3. **Music and Video Playlists:**
    - Linked lists can be used to implement playlists in music and video players. Each node in the list represents a song or video, and the "next" pointer points to the next item in the playlist.
4. **Symbol Table in Compilers:**
    - Compilers use symbol tables to keep track of variables and their scopes. Linked lists can be employed to represent these symbol tables, facilitating efficient lookup and management of symbols.
5. **Dynamic Memory Allocation in C Programs:**
    - Linked lists can be used for dynamic memory allocation in C programs. Each node can represent a dynamically allocated memory block, forming a chain of allocated segments.
6. **Hash Table Chaining:**

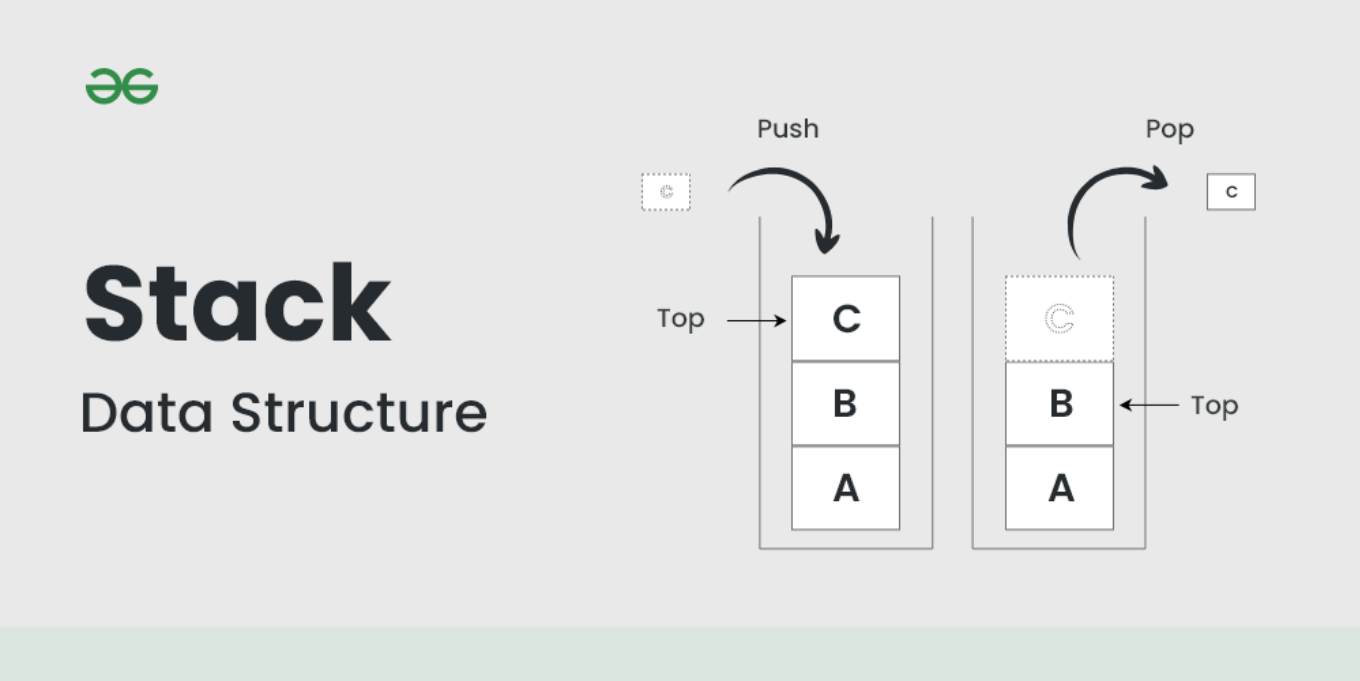
- In hash table implementations, linked lists are often used for collision resolution. When two or more elements hash to the same index, they can be stored in a linked list at that index.

7. **Garbage Collection Algorithms:**
    - Garbage collection algorithms in programming languages often use linked lists to manage and track memory that can be freed.
8. **Sparse Matrix Representation:**
    - In the representation of sparse matrices, linked lists can be used to store only non-zero elements efficiently, saving memory and improving access time.
9. **Graph Algorithms:**
    - Linked lists are commonly used to represent adjacency lists in graph algorithms. Each node in the list represents a vertex, and its linked nodes represent adjacent vertices.
10. **Dynamic Data Structures:**
    - Linked lists are well-suited for scenarios where the size of the data structure may change during runtime, such as in dynamic stacks, queues, and other dynamic data structures.
11. **Polynomial Representation:**
    - Linked lists can represent polynomials efficiently, with each node storing a term of the polynomial.
12. **Job Scheduling in Real-time Systems:**
    - Linked lists can be used in real-time systems for job scheduling, where tasks with varying execution times are dynamically managed.

These applications demonstrate the flexibility and usefulness of linked lists in different fields and scenarios. Linked lists are particularly valuable when the size of the data or the frequency of insertions and deletions is unpredictable and dynamic.

Stack

- A stack is a collection or list of elements where elements are added and removed only from one end, known as the "top" end.
- It follows the Last In, First Out (LIFO) or First In, Last Out (FILO) principle, where the last element added is the first one to be removed.
- Three basic operations on a stack:
    1. **Push:** Inserting or adding an element to the top of the stack.
    2. **Pop:** Removing or deleting the element from the top of the stack.
    3. **Peek:** Obtaining the value of the topmost element without modifying the stack.
- Stack is dynamic in nature, and it is commonly used for managing function calls, expressions, and undo functionalities.



Static Stack (Array Implementation)

- Implementation using a static array (fixed size).
- Example structure:

```
struct Stack {
    int arr[SIZE];
    int top;
};
```

Operations on Stack

1. **Push:**
    - Check if the stack is not full (`top != SIZE-1`).
    - Increment the value of `top` by 1.
    - Insert an element into the stack at the top position.

```
void push(stack_t *ps){
    int ele;
    // step1: check stack is not full
    if (ps->top != SIZE - 1){
        printf("enter an ele: ");
        scanf("%d", &ele);
        // step2: increment the value of top by 1
        ps->top++;
```

```
            // step3: insert an ele into the stack at top position
            ps->arr[ps->top] = ele;
        }
        else{
            printf("stack overflow !!!\n");
        }
    }
```

2. **Pop:**

   - Check if the stack is not empty (`top != -1`).
   - Decrement the value of `top` by 1 (deleting the element).

```
    void pop(stack_t *ps){
        // step1: check stack is not empty
            if (ps->top != -1){
                // step2: decrement the value of top by 1
                //( i.e. we are deleting an ele from the stack).
                ps->top--;
                printf("popped element on top\n");
            }
            else{
                printf("stack underflow !!!\n");
            }
    }
```

3. **Peek:**

   - Check if the stack is not empty (`top != -1`).
   - Return the value of the topmost element without modifying `top`.

```
    int peek_element(stack_t *ps)
    {
        // step1: check stack is not empty
        if (ps->top != -1)
    {
            // step2: return the value of topmost ele
            // (without incrementing/decrementing top).
            int ele = ps->arr[ps->top];
            printf("topmost ele is: %d\n", ele);
        }
        else
        {
            printf("stack underflow !!!\n");
```

```
        }
    }
```

Dynamic Stack (Linked List Implementation)

- Implementation using a linked list.
- Example Strcture

```
struct stack
{
    int data;
    struct stack *next;
};
node *head=NULL;
node *top = NULL;
```

- Two common approaches for push and pop:
    1. **Push (Add to Last):**
        - Use a function like `add_last()` to insert a new node at the end of the linked list.
    2. **Pop (Delete Last):**
        - Use a function like `delete_last()` to remove the last node from the linked list.
    - Alternatively, you can use a linked list where push corresponds to `add_first()` and pop corresponds to `delete_first()`.

The dynamic stack implemented using a linked list allows for a flexible size and efficient memory utilization. Each node in the linked list represents an element in the stack, and operations are performed by updating the pointers accordingly.

Stack Applications

1. **Control Flow in Programs:**
    - Operating Systems use a stack to control the flow of program execution, managing function calls, and storing local variables and return addresses.
2. **Recursion:**
    - Stacks are used internally in operating systems to manage function calls and local variables in recursive programs.
3. **Undo & Redo Functionalities:**
    - Stacks are employed to implement undo and redo functionalities in applications, allowing users to revert or redo their actions.

4. **Graph and Tree Traversal:**
    - Stacks are crucial for implementing depth-first search (DFS) traversal in trees and graphs, where they help keep track of visited nodes and the exploration path.
5. **Expression Conversion and Evaluation:**
    - Stacks are used to implement algorithms for converting and evaluating expressions.
        - **Infix to Postfix Conversion:** Convert an infix expression (e.g., a+b) to its equivalent postfix form (e.g., ab+).
        - **Infix to Prefix Conversion:** Convert an infix expression to its equivalent prefix form (e.g., +ab).
        - **Prefix to Postfix Conversion:** Convert a prefix expression (e.g., +ab) to its equivalent postfix form (e.g., ab+).
        - **Postfix Expression Evaluation:** Evaluate a postfix expression (e.g., ab+).

Expressions, as combinations of operands and operators, can be represented in different notations: infix, prefix, and postfix. Each notation has its own rules for the placement of operators and operands.

These applications showcase the versatility of stacks in managing program execution, algorithmic operations, and expression manipulations. Stacks provide a simple and efficient way to handle various computational tasks.

We consider five binary operations: +, -, *, / and $ or ^ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest):

Operator	Precedence	Value
^ $	Highest	3
* / %	Next highest	2
+ -	Lowest	1

Algorithm to Convert Infix to Postfix

Input: Infix expression: `a*b*c/d*e+f*g-h`

Output: Postfix expression: `ab*c*d/e*fg*+h-`

Stack Contains Operators and it is Initially empty

Step 1: Start scanning the infix expression from left to right.

Step 2: Pseudo-code.

```
for each element in the infix expression {
    if (current element is an operand) {
        append it to the postfix expression
    } else { // current element is an operator
```

```
        while (!is_stack_empty && priority(topmost element) ⩾
priority(current element)) {
            pop element from the stack and append it to the postfix
expression
        }
        push current element onto the stack
    }
}
```

Step 3: Repeat the above steps until the end of the infix expression.

Step 4: Pop all remaining elements from the stack one by one and append them to the postfix expression.

Algorithm to Convert Infix to Prefix

Input: Infix expression: `a*b*c/d*e+f*g-h`

Output: Prefix expression: `-+*/\*\*abcde*fgh`

Stack Contains Operators and it is Initially empty

Step 1: Start scanning the infix expression from right to left.

Step 2: Pseudo-code.

```
for each element in the infix expression {
    if (current element is an operand) {
        append it to the prefix expression
    } else { // current element is an operator
        while (!is_stack_empty && priority(topmost element) >
priority(current element)) {
            pop element from the stack and append it to the prefix
expression
        }
        push current element onto the stack
    }
}
```

Step 3: Repeat the above steps until the end of the infix expression.

Step 4: Pop all remaining elements from the stack one by one and append them to the prefix expression.

Step 5: Reverse the prefix expression.

Algorithm for Prefix to Postfix Conversion

Input: Prefix expression: `-+/**abcdefgh`

Output: Postfix expression: `ab*c*d/e+f-`

Stack Contains Operators and it is Initially empty

Step 1: Start scanning the prefix expression from left to right.

Step 2: Pseudo-code.

```
for each element in the prefix expression {
    if (current element is an operand) {
        push it onto the stack
    } else { // current element is an operator
        // Pop two elements from the stack
        op1 = pop from the stack
        op2 = pop from the stack

        // Concatenate the operands with the current operator and push the
result back onto the stack
        result = op1 + op2 + current element
        push result onto the stack
    }
}
```

Step 3: Repeat the above steps until the end of the prefix expression.

Step 4: The final result is on the top of the stack.

Algorithm for Postfix Evaluation

Input: Postfix expression: `45*8+73/-4+`

Output: Result: `30`

Stack Contains Operators and it is Initially empty

Step 1: Start scanning the postfix expression from left to right.

Step 2: Pseudo-code.

```
for each element in the postfix expression {
    if (current element is an operand) {
        push it onto the stack
    } else { // current element is an operator
        // Pop two elements from the stack
        op2 = pop from the stack
```

```
            op1 = pop from the stack

            // Perform the operation and push the result back onto the stack
            result = op1 (current element) op2
            push result onto the stack
        }
    }
```

Step 3: Repeat the above steps until the end of the postfix expression.

Step 4: Pop the final result from the stack.

Example Execution: Postfix expression: `45*8+73/-4+`

Stack (Operands):

- Initially empty

Execution:

- Operand '4': Push onto the stack.
- Operand '5': Push onto the stack.
- Operator '*': Pop '5' and '4' from the stack, perform multiplication, push '20' onto the stack.
- Operand '8': Push onto the stack.
- Operator '+': Pop '8' and '20' from the stack, perform addition, push '28' onto the stack.
- Operand '7': Push onto the stack.
- Operand '3': Push onto the stack.
- Operator '/': Pop '3' and '7' from the stack, perform division, push '2' onto the stack.
- Operator '-': Pop '28' and '2' from the stack, perform subtraction, push '26' onto the stack.
- Operand '4': Push onto the stack.
- Operator '+': Pop '4' and '26' from the stack, perform addition, push '30' onto the stack.

Final Result: Result: `30`

Queue

A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a “FIFO” or “First-in-first-out” list.

The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations on queues:

- enqueue: which inserts an element at the end of the queue.
- dequeue: which deletes an element at the start of the queue.

Your explanations are accurate and provide a good overview of different types of queues. Let's summarize the key points:

1. **Linear Queue:**

- Basic queue structure where elements are added at the rear (enqueue) and removed from the front (dequeue).
- Follows the First In, First Out (FIFO) principle.

2. **Circular Queue:**

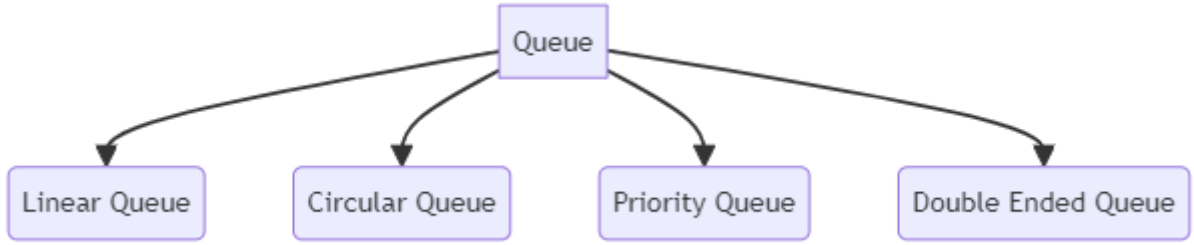
- Similar to linear queue but with a circular arrangement of elements.
- Enqueue and dequeue operations can be performed efficiently in a circular manner.

3. **Priority Queue:**

- Allows elements to be added randomly without considering priority.
- The element with the highest priority is dequeued first.
- Can be implemented using a linked list or efficiently using a binary heap.

4. **Double-Ended Queue (Deque):**

- Allows elements to be added or removed from both ends.
- Supports four basic operations in `O(1)` time:
  1. `push_front` (add at the front): `add_first()`
  2. `push_back` (add at the rear): `add_last()`
  3. `pop_front` (remove from the front): `delete_first()`
  4. `pop_back` (remove from the rear): `delete_last()`
- Can be implemented using a doubly circular linked list for efficient operations.



Implementation of Linear Queue using Array

1. Initialization:

- Define a structure to represent the queue, including an array to store elements

- Initialize front and rear pointers to -1 indicating an empty queue.

```c
#define SIZE 100
typedef struct queue
{
    int arr[SIZE];
    int rear;
    int front;
} queue_t;
```

2. Enqueue (Insertion):

- Verify if the queue is not full before inserting an element.
- Increment the `rear` pointer.
- Insert the new element at `arr[rear]`.

```c
void enqueue(queue_t *pq)
{
    int ele;
    // Step 1: Check if the queue is not full
    if (!is_queue_full(pq))
    {
        printf("Enter an element: ");
        scanf("%d", &ele);
        // step2: increment the value of rear by 1
        pq->rear++;
        // step3: insert/push an ele into the queue from rear end
        pq->arr[pq->rear] = ele;

        // step4: if( front == -1 ), front = 0
        if (pq->front == -1)
            pq->front = 0;

        printf("%d is inserted into the queue\n", ele);
    }
    else
    {
        printf("Queue is full!!!\n");
    }
}
```

3. Dequeue (Deletion):

- Ensure the queue is not empty before attempting to dequeue.

- Retrieve the element at the front (`arr[front]`).
- Increment the `front` pointer.

```
void dequeue(queue_t *pq)
{
    // Step 1: Check if the queue is not empty
    if (!is_queue_empty(pq))
    {
        // Step 2: Get the element at the front of the queue
        int ele = get_front(pq);

        // Step 3: Increment front, considering the linear nature of the
queue
        pq->front++;
        printf("%d is deleted from the queue\n", ele);
    }
    else
    {
        printf("Queue is empty!!!\n");
    }
}
```

4. Peek (Front Element):

- Confirm the queue is not empty before peeking.
- Return the element at the front (`arr[front]`).

```
int get_front(queue_t *pq)
{
    // Return the value of an element at the front position
    return (pq->arr[pq->front]);
}
```

5. Check if Queue is Empty:

- Check if `rear` is `-1` or less than `front` to determine if the queue is empty.

```
int is_queue_empty(queue_t *pq)
{
    return (pq->rear == -1 || pq->front > pq->rear);
}
```

6. Check if Queue is Full:

- Verify if `rear` is equal to `SIZE - 1` to determine if the queue is full.

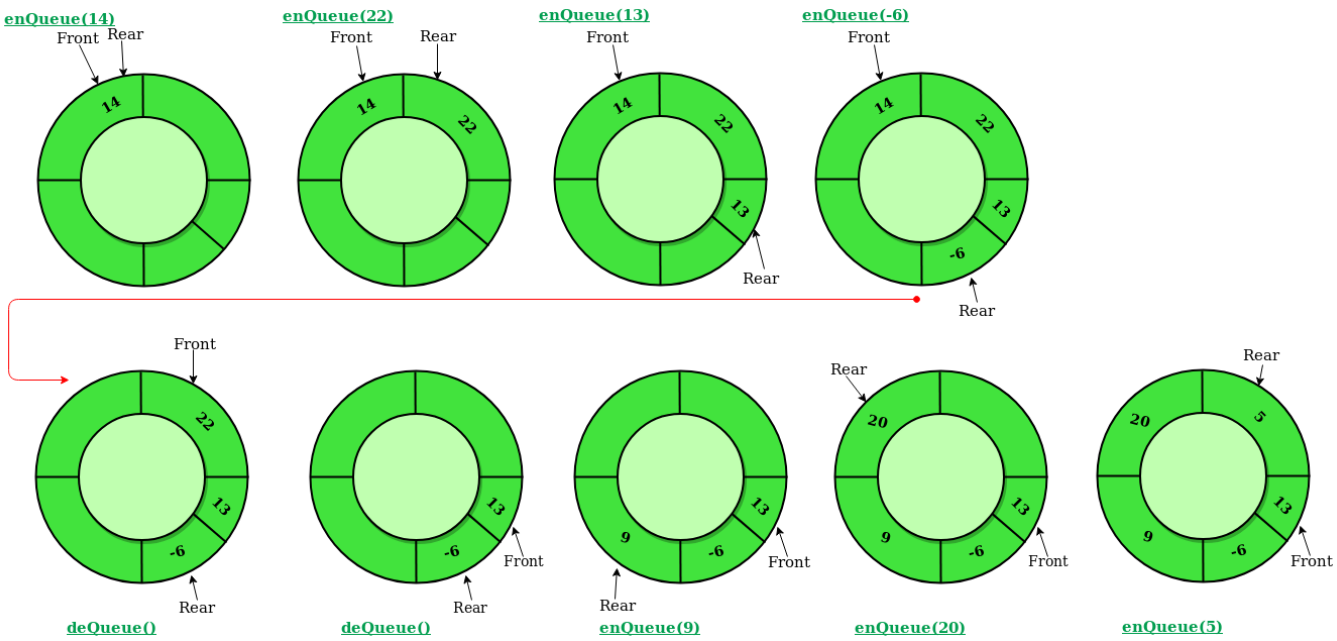
```
int is_queue_full(queue_t *pq)
{
    return ( pq→rear == SIZE-1 );
}
```

Limitations of Linear Queue:

- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.
- Signaling queue full: vacant places cannot be reutlized in a linear queue

---

Implementation of Circular Queue using Array



1. Initialization:

- Define a structure to represent the queue, including an array to store elements
- Initialize front and rear pointers to -1 indicating an empty queue.

```
#define SIZE 100
typedef struct Circular_queue
{
    int arr[SIZE];
    int rear;
```

```
    int front;
} queue_t;
```

2. Enqueue (Insertion):

- Verify if the queue is not full before inserting an element.
- Increment the `rear` pointer.
- Insert the new element at `arr[rear]`.

```
void enqueue(queue_t *pq)
{
    int ele;
    // Step 1: Check if the queue is not full
    if (!is_queue_full(pq))
    {
        printf("Enter an element: ");
        scanf("%d", &ele);
        // Step 2: Increment the value of rear by 1
        pq->rear = (pq->rear + 1) % SIZE;
        /*rear++; ⇒ rear = rear + 1
            for incrementing value of rear by 1
            rear = (rear+1)%SIZE
            if rear = 0 ⇒ rear = (rear+1)%SIZE ⇒ (0+1)%5 ⇒ 1%5 ⇒ 1
            if rear = 1 ⇒ rear = (rear+1)%SIZE ⇒ (1+1)%5 ⇒ 2%5 ⇒ 2
            if rear = 4 ⇒ rear = (rear+1)%SIZE ⇒ (4+1)%5 ⇒ 5%5 ⇒ 0
            */

        // Step 3: Insert/push an element into the queue from the rear end
        pq->arr[pq->rear] = ele;

        // Step 4: If front == -1, set front to 0
        if (pq->front == -1)
            pq->front = 0;

        printf("%d is inserted into the queue\n", ele);
    }
    else
    {
        printf("Queue is full!!!\n");
    }
}
```

3. Dequeue (Deletion):

- Ensure the queue is not empty before attempting to dequeue.
- Retrieve the element at the front (`arr[front]`).

- If there is only one element in queue then reset `rear` and `front` to `-1` else increment the `front` pointer.

```
void dequeue(queue_t *pq)
{
    // Step 1: Check if the queue is not empty
    if (!is_queue_empty(pq))
    {
        // Step 2: Get the element at the front of the queue
        int ele = get_front(pq);

        // Step 3: Check if there is only one element in the queue
        if (pq->front == pq->rear)
        {
            // If yes, reset front and rear to indicate an empty queue
            pq->rear = pq->front = -1;
        }
        else
        {
            // Step 4: Increment front, considering the circular nature of
the queue
            pq->front = (pq->front + 1) % SIZE;
            /*front++; ⇒ front = front + 1
            for incrementing value of front by 1
            front = (front+1)%SIZE
            if front = 0 ⇒ front = (front+1)%SIZE ⇒ (0+1)%5 ⇒ 1%5 ⇒ 1
            if front = 1 ⇒ front = (front+1)%SIZE ⇒ (1+1)%5 ⇒ 2%5 ⇒ 2
            if front = 4 ⇒ front = (front+1)%SIZE ⇒ (4+1)%5 ⇒ 5%5 ⇒ 0
            */
        }
        printf("%d is deleted from the queue\n", ele);
    }
    else
    {
        printf("Queue is empty!!!\n");
    }
}
```

4. Peek (Front Element):

- Confirm the queue is not empty before peeking.
- Return the element at the front (`arr[front]`).

```
int get_front(queue_t *pq)
{
    // return the value of an ele which is at front position
```

```
    return (pq→arr[pq→front]);
}
```

5. Check if Queue is Empty:

- Check if `rear` is `-1` and equal to `front` to determine if the queue is empty.

```
int is_queue_empty(queue_t *pq)
{
    return (pq→rear == -1 && pq→front == pq→rear);
}
```

6. Check if Queue is Full:

- Verify if `front` is 1 more than rear, then queue is full.

```
int is_queue_full(queue_t *pq)
{
    return (pq→front == (pq→rear + 1) % SIZE);
   /*
   - if front is at next position of rear we can say queue is full:
    queue_full: front == (rear + 1) % SIZE
    for rear = 0, front = 1
    front == (rear + 1) % SIZE ⟹ (0+1) % 5 ⟹ 1 % 5 ⟹ 1

    for rear = 1, front = 2
    front == (rear + 1) % SIZE ⟹ (1+1) % 5 ⟹ 2 % 5 ⟹ 2

    for rear = 4, front = 0
    front == (rear + 1) % SIZE ⟹ (4+1) % 5 ⟹ 5 % 5 ⟹ 0
    */
}
```

Applications of Queue

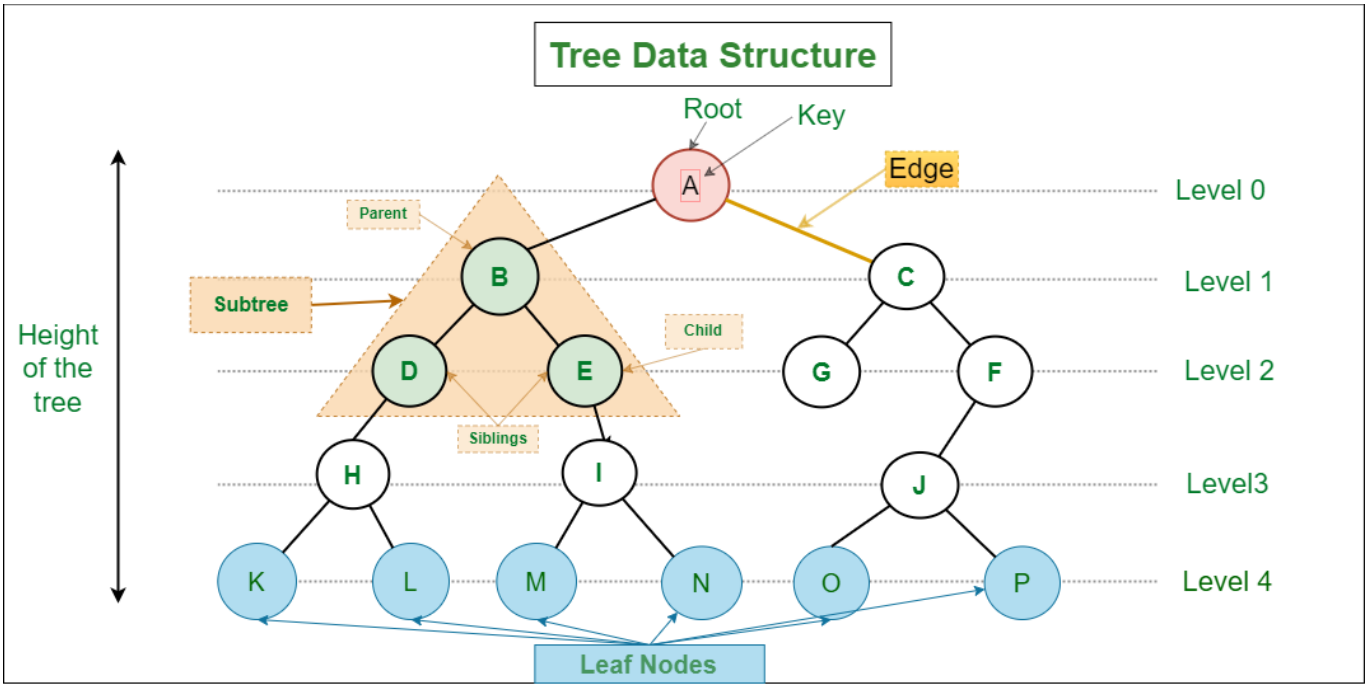
- **FIFO Data Processing:**
    - Queues are essential for applications where elements need to be processed in a First-In-First-Out manner.
- **Breadth-First Search (BFS) in Tree and Graph:**

- Queues play a key role in BFS algorithms for traversing tree and graph structures.

- **Kernel Data Structures:**
  - Queues are employed for kernel data structures in operating systems, managing tasks like ready queues, job queues, and message queues.
- **OS Scheduling:**
  - Queues are crucial for OS scheduling algorithms, including Priority CPU Scheduling and FCFS CPU Scheduling.
- **General OS Structures:**
  - Queues are used for various OS data structures like job queues, ready queues, and waiting queues.
- **Diverse OS Algorithms:**
  - Queues are integral to OS algorithms such as FCFS CPU Scheduling, Priority CPU Scheduling, and FIFO Page Replacement.

Tree

A "non-linear" and "advanced data structure," a tree is a collection of logically related finite elements. It features a specially designated element called the "root," with all other elements connected to the root in a hierarchical manner, forming a parent-child relationship.



- Elements in a tree are referred to as "nodes".
- Key Node Types:
  - Root Node: The first element in a tree.
  - Parent Node/Father: A node with child nodes.
  - Child Node/Son: Nodes connected to a parent.

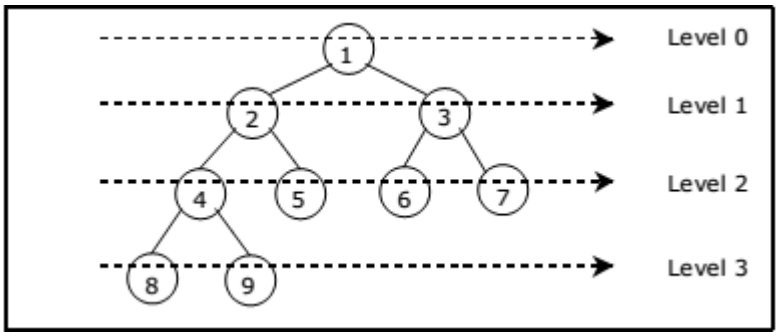
    - Grandparent/Grandfather: Ancestor of a parent node.
    - Grandchild/Grandson: Descendant of a child node.
    - Siblings/Brothers: Child nodes of the same parent.
- **Node Degrees:**
    - Degree of a Node: The number of child nodes it has.
    - Degree of a Tree: The maximum degree among all nodes.
- **Node Types:**
    - Leaf Node/Terminal Node/External Node: A node with a degree of 0, having no child nodes.
    - Non-leaf Node/Non-terminal Node/Internal Node: A node with a non-zero degree, having child nodes.
- **Hierarchy Concepts:**
    - Ancestors: Nodes in the path from the root to a given node, including the root (excluding the node itself).
    - Descendants: Nodes accessible from a given node.
    - Level of a Node: The level of its parent node plus 1, with the root node assigned level 0.
    - Level of a Tree: The maximum level of any node in the tree.
    - Depth of a Tree: Equivalent to the level of a tree.
    - Height of a Tree: The length of the longest path from the root node to a leaf node.
- In a tree data structure, nodes can have varying numbers of child nodes, and the structure can grow at any level.

Binary Tree

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**. A tree with no nodes is called as a **null** tree. Binary trees are easy to implement because they have a small, fixed number of child links. Because of this characteristic, binary trees are the most common types of trees and form the basis of many important data structures.

Assigning level numbers and Numbering of nodes for a binary tree:

- The nodes of a binary tree can be numbered in a natural way, level by level, left to right.
- The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent.



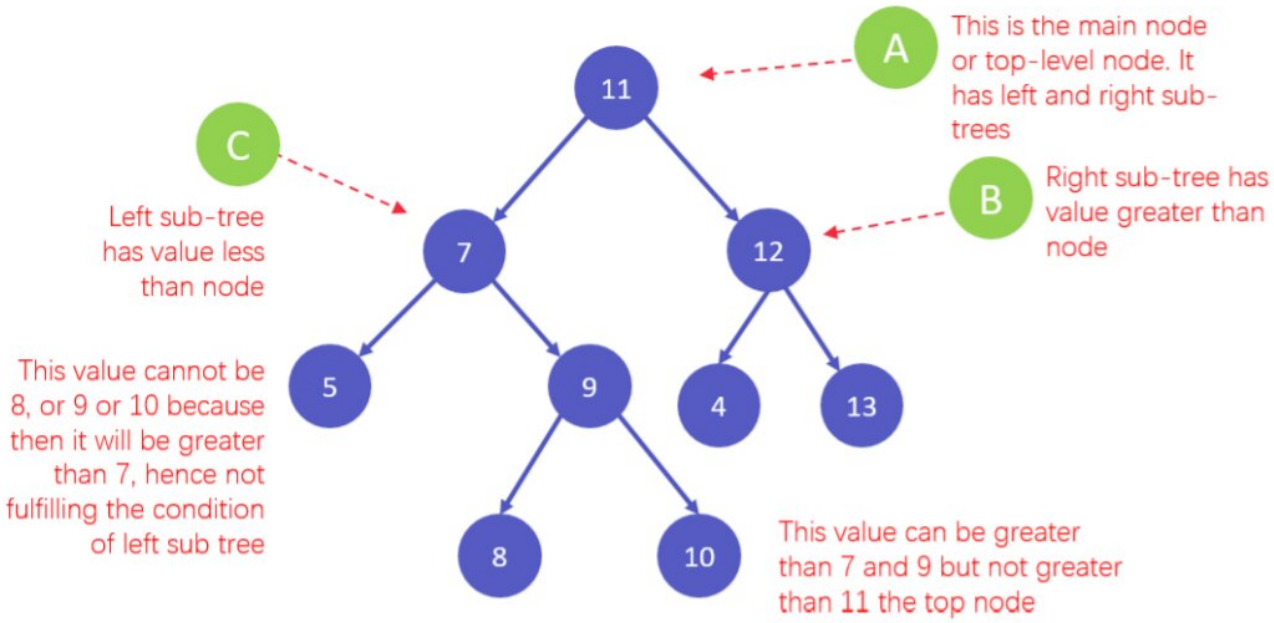
Properties of binary tree: Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then
    1. Maximum number of leaves = 2^h

    2. Maximum number of nodes = `2h + 1 - 1`
2. If a binary tree contains `m` nodes at level `l`, it contains at most `2m` nodes at level `l + 1`.
3. Since a binary tree can contain at most one node at level 0 (`the root`), it can contain at most `2l` node at level `l`.
4. The total number of edges in a full binary tree with `n` node is `n - 1`.

Binary Search Tree(BST)

It is a binary tree in which left child is always smaller than its parent and right child is always greater than or equal to its parent.



Depth-First Search (DFS)

Depth-First Search (DFS) is a traversal method used to explore and visit all the nodes of a binary search tree (BST) in a systematic way. There are three types of DFS traversal for a binary search tree:

1. **Inorder Traversal:**
    - In inorder traversal, we visit the nodes of the BST in the following order:
        1. Visit the **left** subtree.
        2. Visit the **root** node.
        3. Visit the **right** subtree.
    - This traversal results in visiting nodes in ascending order for a BST.

    Example:

```
    20
   /  \
10    30
```

```
 / \      \
5  15     40

Inorder Traversal: 5, 10, 15, 20, 30, 40
```

2. **Preorder Traversal:**
    - In preorder traversal, we visit the nodes of the BST in the following order:
        1. Visit the **root** node.
        2. Visit the **left** subtree.
        3. Visit the **right** subtree.
    - This traversal is useful for creating a copy of the tree.

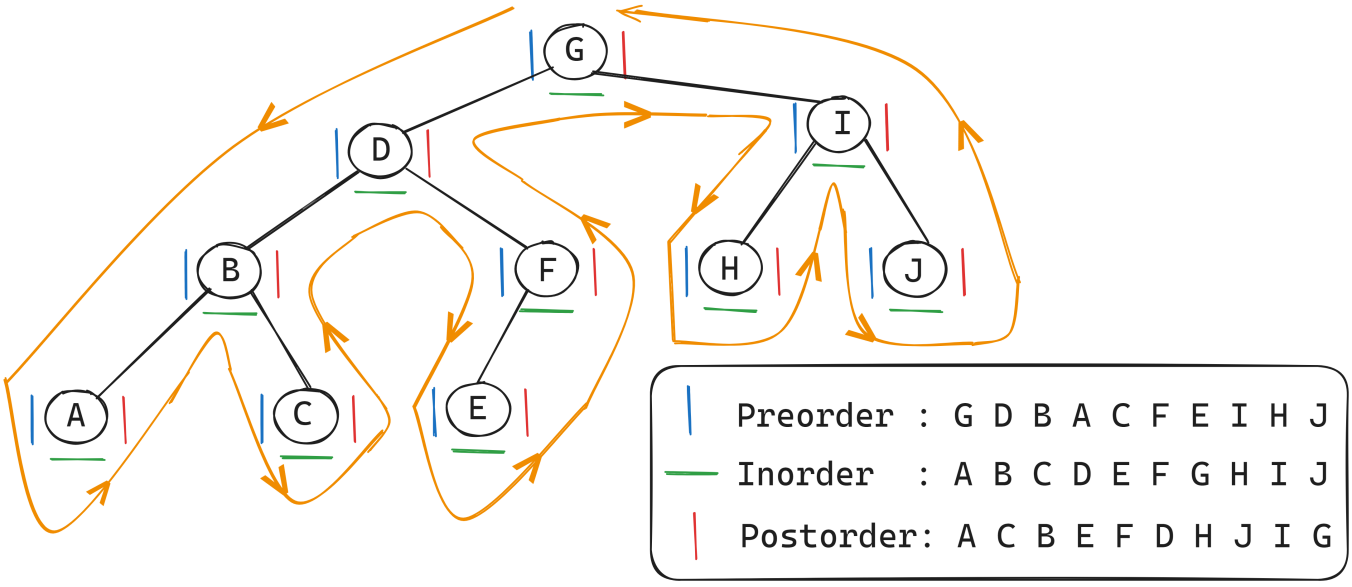
    Example:

```
     20
    /  \
  10    30
 / \      \
5  15     40

Preorder Traversal: 20, 10, 5, 15, 30, 40
```

3. **Postorder Traversal:**
    - In postorder traversal, we visit the nodes of the BST in the following order:
        1. Visit the **left** subtree.
        2. Visit the **right** subtree.
        3. Visit the **root** node.
    - This traversal is useful for deleting nodes from the tree.

    Example:

```
     20
    /  \
  10    30
 / \      \
5  15     40

Postorder Traversal: 5, 15, 10, 40, 30, 20
```



Breadth-First Search (BFS)/Level Order Traversal

Breadth-First Search (BFS) is a traversal method used to explore and visit all the nodes of a binary search tree (BST) in a level-by-level order. It starts from the root node and explores all the nodes at the current depth before moving on to nodes at the next depth. The traversal is performed using a queue data structure.

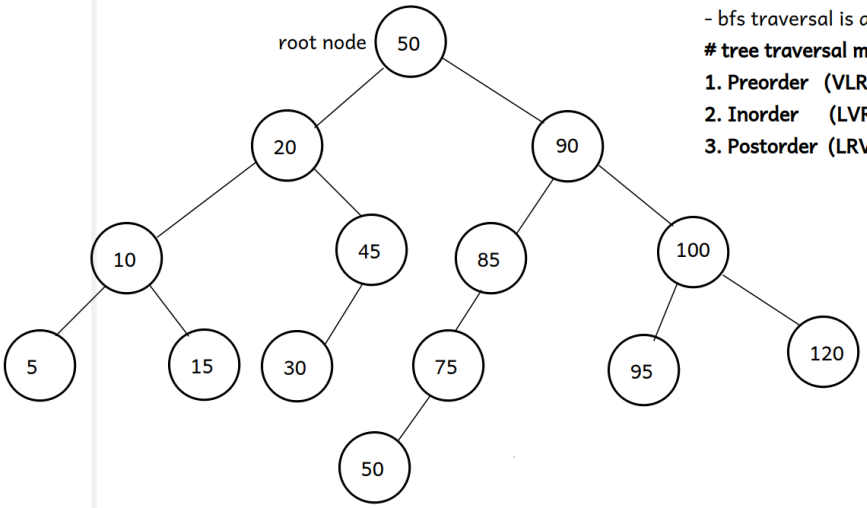
Example:

```
      20
     /  \
   10    30
  / \      \
 5  15     40

BFS Traversal: 20, 10, 30, 5, 15, 40
```

In the example, BFS starts at the root (20) and visits nodes level by level, first 20, then 10 and 30, and finally 5, 15, and 40. The order of visiting nodes at the same level is from left to right.

BFS traversal is helpful in various scenarios, such as finding the shortest path between two nodes, analyzing connectivity in a graph, and exploring a tree layer by layer.

input order of an ele's for BST: 50 20 90 85 10 45 30 100 15 75 95 120 5 50



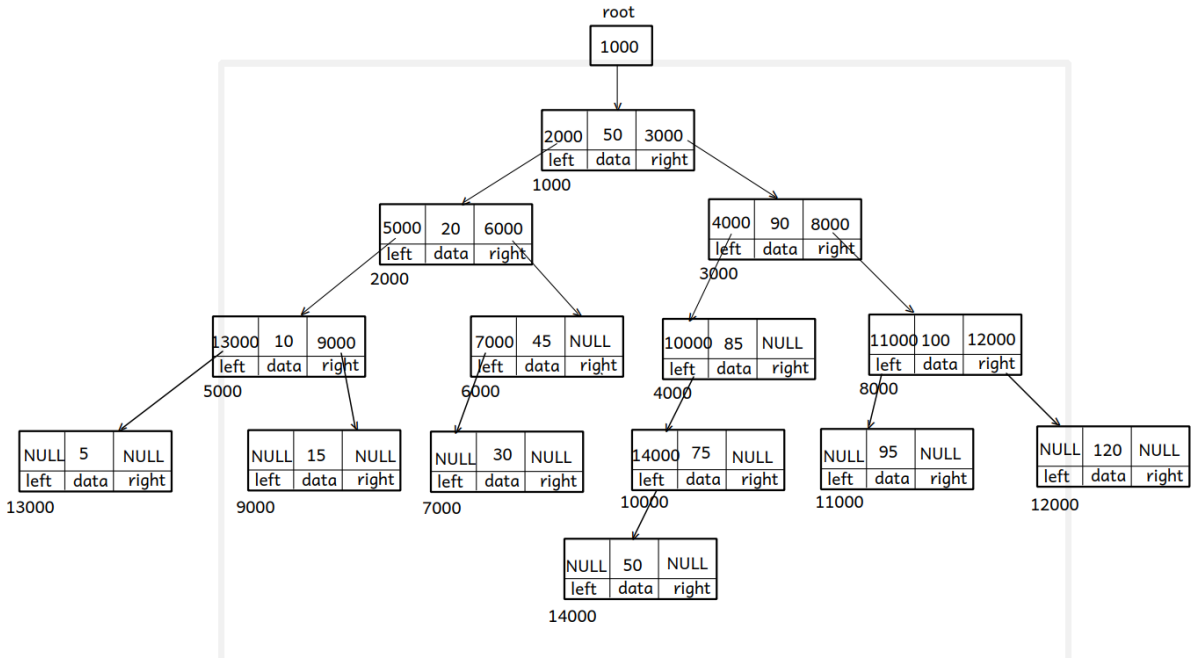
1. dfs traversal: 50 20 10 5 15 45 30 90 85 75 50 100 95 120
2. bfs traversal: 50 20 90 10 45 85 100 5 15 30 75 95 120 50
- bfs traversal is also called as "levelwise traversal".
 # tree traversal methods on BST:
1. Preorder (VLR) : 50 20 10 5 15 45 30 90 85 75 50 100 95 120
2. Inorder (LVR): 5 10 15 20 30 45 50 50 75 85 90 95 100 120
3. Postorder (LRV): 5 15 10 30 45 20 50 75 85 95 120 100 90 **50**

Implementation of BST using Linked List:

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node
{
    struct node *left;
    int data;
    struct node *right;
} node_t;

// create an empty bst
node_t *root = NULL;

void add_node(int data);
node_t *create_node(int data);
void preorder(node_t *trav);
void inorder(node_t *trav);
void postorder(node_t *trav);

int main(void)
{
    // 50 20 90 85 10 45 30 100 15 75 95 120 5 50
    add_node(50); add_node(20); add_node(90); add_node(85); add_node(10);
    add_node(45); add_node(30); add_node(100); add_node(15); add_node(75);
    add_node(95); add_node(120); add_node(5); add_node(50);

    printf("PREORDER : "); preorder(root); printf("\n");
    printf("INORDER  : "); inorder(root); printf("\n");
    printf("POSTORDER: "); postorder(root); printf("\n");

    return 0;
```

```
}

void add_node(int data)
{
    // create a newnode
    node_t *newnode = create_node(data);

    // if bst is empty - attach newly created node to the root
    if (root == NULL)
    {
        root = newnode;
    }
    else // if bst is not empty → find appropriate pos of that node
    {
        // traverse the bst and find an appropriate position
        node_t *trav = root;

        while (1)
        {
            if (data < trav→data) // node will be a part of its left
subtree
            {
                if (trav→left == NULL) // if left subtree of cur node is
empty
                {
                    // attach newly created node as left child to the cur
node
                    trav→left = newnode;
                    break;
                }
                else // if cur node is having left subtree
                {
                    trav = trav→left; // goto its left subtree
                }
            }
            else // node will be a part its right subtree
            {
                if (trav→right == NULL) // if right subtree of a cur node
is empty
                {
                    // attach newly created node as right child of cur node
                    trav→right = newnode;
                    break;
                }
                else // if cur node is having right subtree
                {
                    trav = trav→right; // goto its right subtree
                }
            }
        }
    }
}
void preorder(node_t *trav)
{
```

```
    if (trav == NULL)
        return;

    // VLR
    printf("%4d", trav->data);
    preorder(trav->left);
    preorder(trav->right);
}
void inorder(node_t *trav)
{
    if (trav == NULL)
        return;

    // LVR
    inorder(trav->left);
    printf("%4d", trav->data);
    inorder(trav->right);
}
void postorder(node_t *trav)
{
    if (trav == NULL)
        return;

    // LRV
    postorder(trav->left);
    postorder(trav->right);
    printf("%4d", trav->data);
}
node_t *create_node(int data)
{
    // allocate memory dynamically for a node
    node_t *temp = (node_t *)malloc(sizeof(node_t));
    if (temp == NULL)
    {
        printf("malloc() failed !!!\n");
        exit(1);
    }

    // initialize its members
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;

    // return an addr of newly created node to the calling function
    return temp;
}
```



---

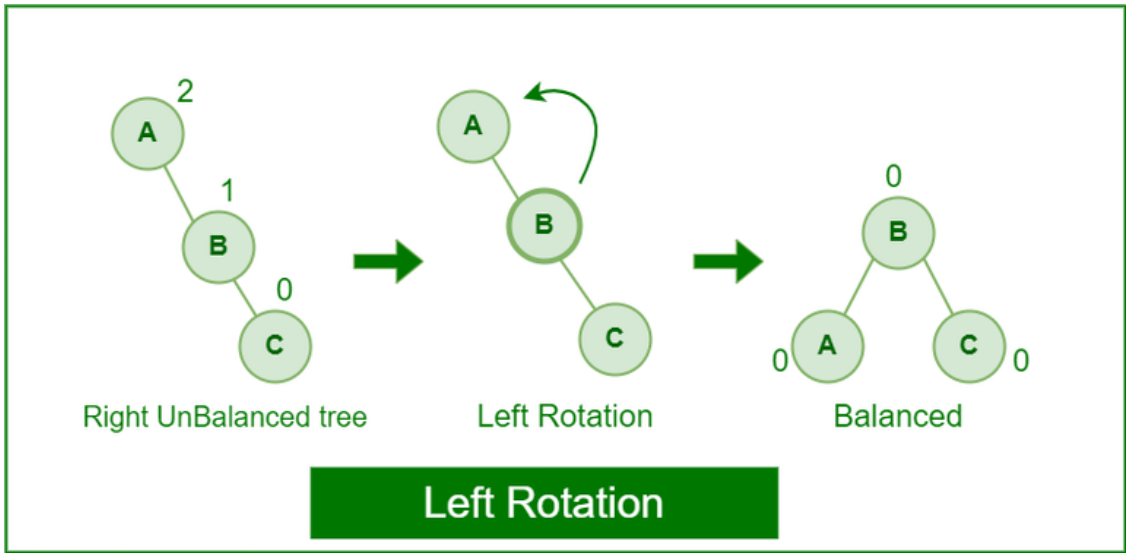
1. **Height of a Node:**
   - Height of a node = `max(ht. of left subtree || ht. of its right subtree ) + 1`
   - The height of every leaf node = 1.
2. **Height of a Tree:**
   - height of a tree = max height of any node in a given tree
3. **Skewed Binary Search Trees:**
   - **Right skewed binary search tree:** It is a BST in which only right link is used to stored an addr of child nodes, i.e. left link of each node is `NULL`.
   - **Left skewed binary search tree:** It is a BST in which only left link is used to stored an addr of child nodes, i.e. right link of each node is `NULL`.
4. **Maximum Height of a BST:**
   - max ht. of a BST = "n", where "n" = no. of ele's in a BST.
5. **Imbalanced BST:**
   - An imbalanced BST is a tree with the maximum height for a given input size.
   - Operations like addition, deletion, and searching become inefficient in an imbalanced BST.
6. **Balanced Binary Search Tree:**
   - A balanced BST has the minimum height for a given input size.
   - The minimum height of a BST is `log(n)`, where 'n' is the input size.
7. **Balanced Node and Balance Factor:**
   - A node is balanced if its balance factor is `-1`, `0`, or `+1`.

- balance factor of a node = (ht. of left subtree - ht. of its right subtree)
- If balance factor of a node < `-1` --> left imbalanced
- If balance factor of a node > `+1` --> right imbalanced

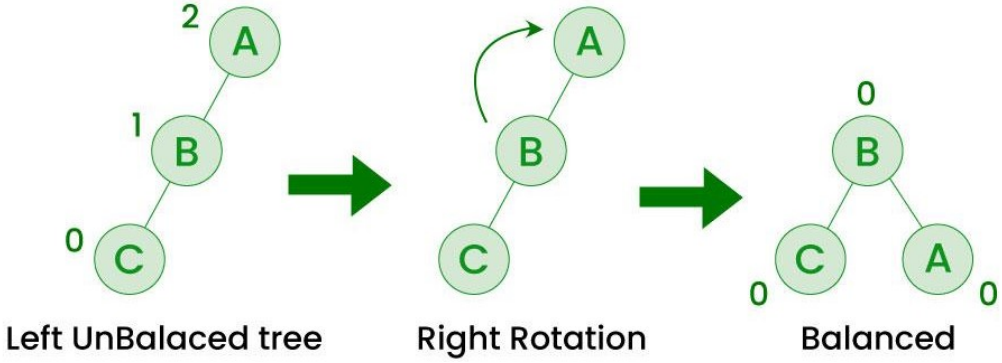
Balancing BST

Balancing a binary search tree (BST) involves performing rotations to maintain or restore balance factors of nodes in the tree. The most common rotations are the left rotation, right rotation, left-right rotation, and right-left rotation. Below are the basic steps to balance a BST:

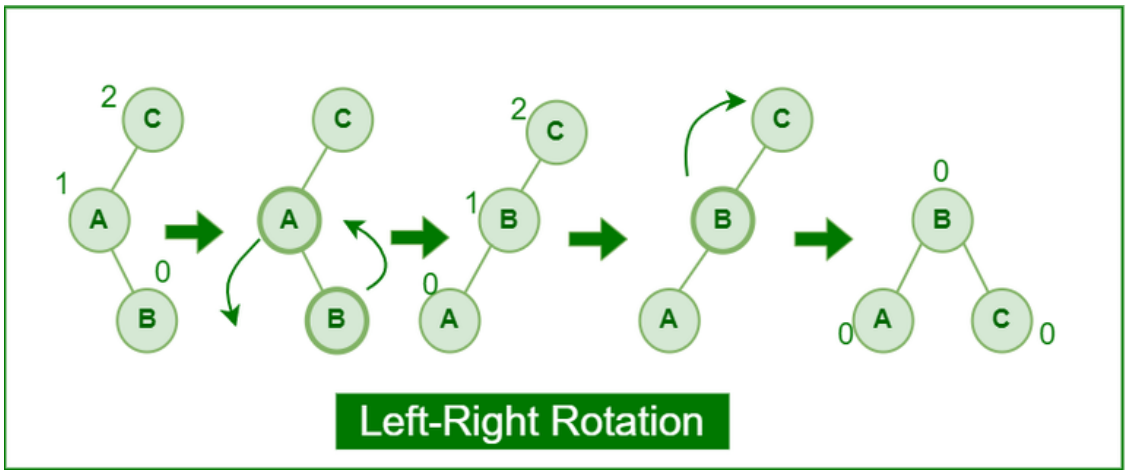
1. **Calculate Balance Factor:**
   - For each node in the BST, calculate the balance factor, which is the difference between the height of the left subtree and the height of the right subtree.
2. **Identify Imbalanced Nodes:**
   - Identify nodes with a balance factor less than -1 or greater than 1. These nodes are imbalanced and need to be balanced.
3. **Perform Rotations:**
   - Based on the balance factors and the relative positions of nodes, perform rotations to restore balance. The four fundamental rotations are:
     - **Left Rotation:** Rebalance a right-heavy subtree by rotating the node to the left.



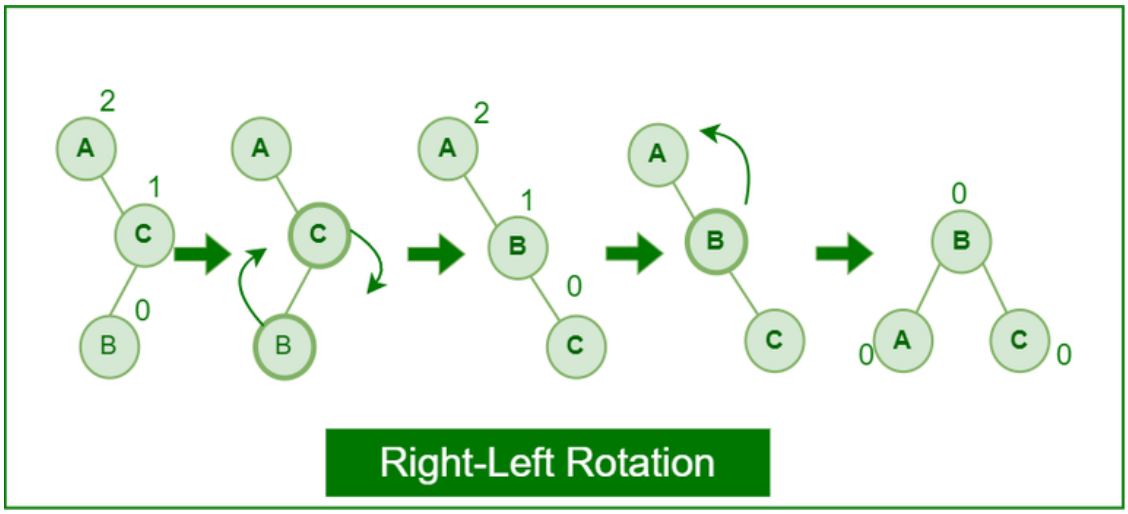
     - **Right Rotation:** Rebalance a left-heavy subtree by rotating the node to the right.



- **Left-Right Rotation:** Perform a left rotation on the left child followed by a right rotation on the root.



- **Right-Left Rotation:** Perform a right rotation on the right child followed by a left rotation on the root.



4. **Update Heights:**
    - After rotations, update the heights of the affected nodes.
5. **Recursively Balance:**
    - If a rotation results in further imbalances, recursively apply the balancing process until the entire tree is balanced.

By applying these rotations appropriately, the BST can be balanced, ensuring efficient operations. Keep in mind that maintaining balance during insertion and deletion operations is crucial for the overall performance of the BST.

Balancing Tree after tree is made if not efficient hence Self Balancing Tree are introduced.

Self-Balanced Binary Search Tree (AVL Tree)

- AVL trees are self-balanced binary search trees.
- During addition and deletion, AVL trees ensure that the tree remains balanced.
- Developed by mathematicians Adelson-Velsky and Landis.

Applications of AVL Tree:

1. It is used to index huge records in a database and also to efficiently search in that.
2. For all types of in-memory collections, including sets and dictionaries, AVL Trees are used.
3. Database applications, where insertions and deletions are less common but frequent data lookups are necessary Software that needs optimized search.
4. It is applied in corporate areas and storyline games.

Advantages of AVL Tree:

1. AVL trees can self-balance themselves.
2. It is surely not skewed.
3. It provides faster lookups than Red-Black Trees Better searching time complexity compared to other trees like binary tree.
4. Height cannot exceed `log(N)`, where, `N` is the total number of nodes in the tree.

Disadvantages of AVL Tree:

1. It is difficult to implement.
2. It has high constant factors for some of the operations.
3. Less used compared to Red-Black trees.
4. Due to its rather strict balance, AVL trees provide complicated insertion and removal operations as more rotations are performed.
5. Take more processing for balancing.

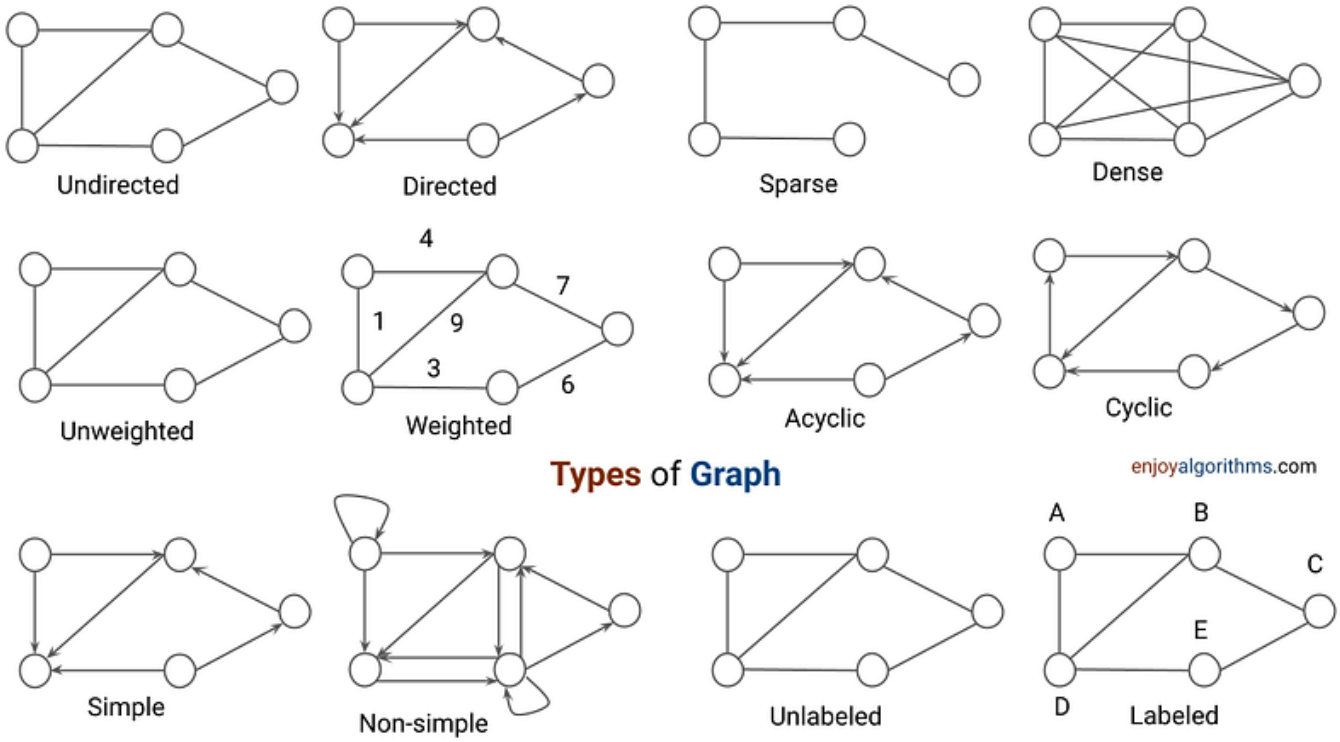
Graphs

A **graph** is a non-linear, advanced data structure that comprises a collection of logically related vertices (or nodes) and edges (or arcs). The edges may be ordered or unordered pairs of vertices, with the possibility of carrying weights, costs, or values.

- **Vertices (Nodes):** Finite elements denoted as {0, 1, 2, 3, 4}.
- **Edges (Arcs):** Finite ordered/unordered pairs of vertices, e.g., {(0,1), (0,2), ...}.
    - Unordered pairs imply an undirected edge.
    - Ordered pairs indicate a directed edge.
- **Loop:** An edge from a vertex to itself.
- **Connected Vertices:** Vertices with a path between them.
- **Connected Graph:** A graph where every vertex is connected to all others.
- **Isolated Vertex:** A vertex with no adjacent vertices.
- **Complete Graph:** All vertices are adjacent to each other.
- **Cycle:** A path where the start and end vertices are the same.

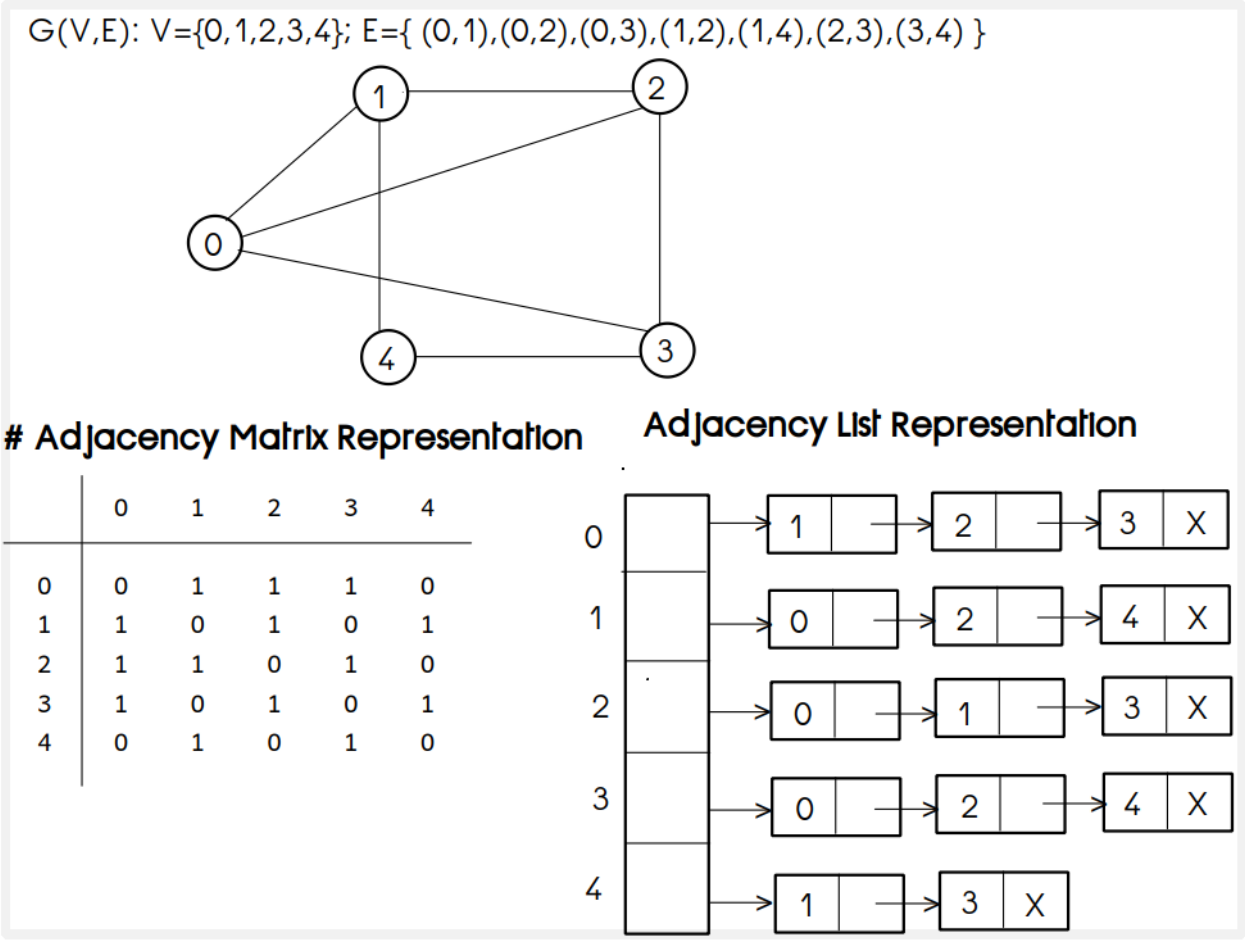
Types of Graphs

1. **Undirected Graph:** Contains unordered pairs of vertices (undirected edges).
2. **Directed Graph (Di-Graph):** Contains ordered pairs of vertices (directed edges).
3. **Weighted Graph:** Edges carry weight/cost/value.
4. **Unweighted Graph:** Edges do not carry weight.



Representation:

1. **Adjacency Matrix:** Utilizes a 2-D array.
2. **Adjacency List:** Utilizes an array of linked lists.

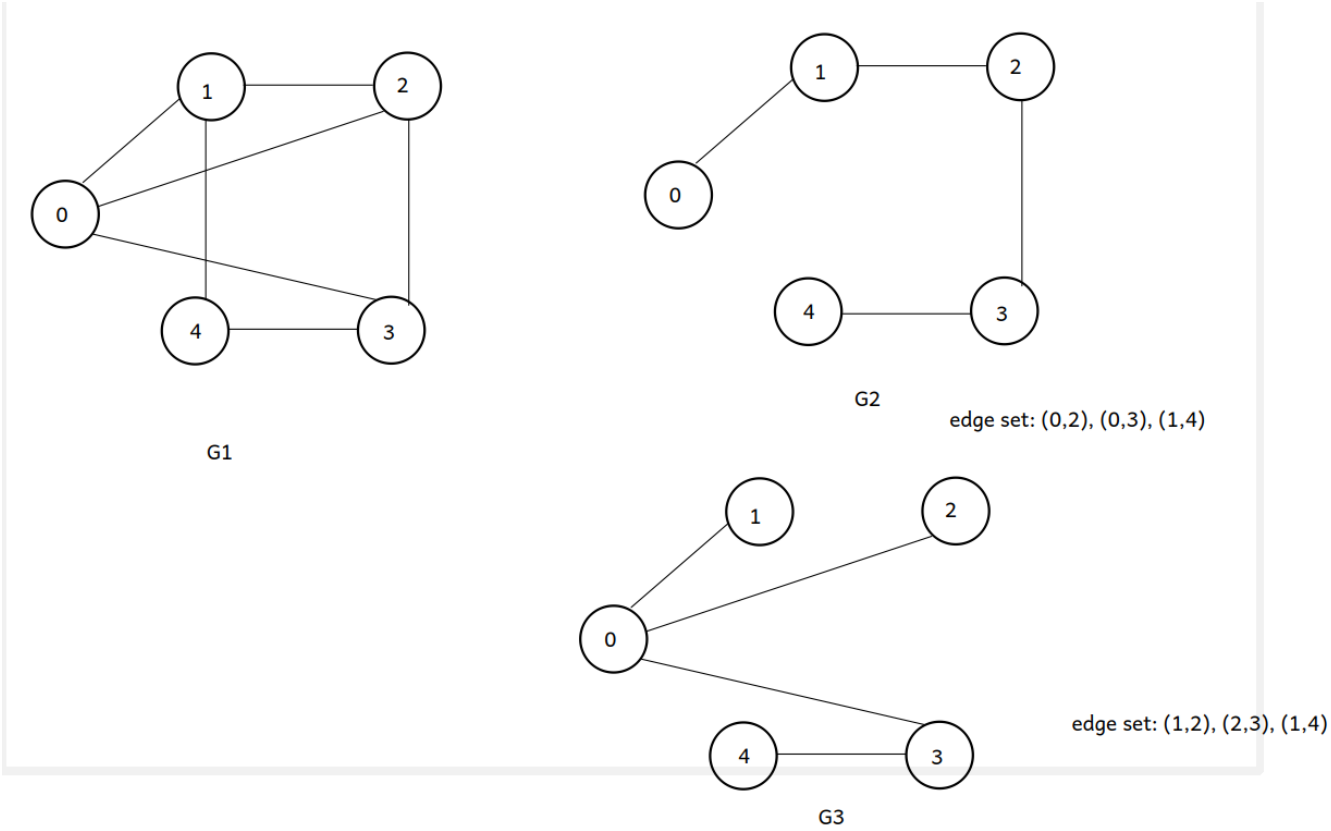


Relationship with Trees:

- Graphs can be trees, but trees cannot be graphs.
- A connected graph without cycles is a tree (specifically, a spanning tree).

Spanning Tree:

- **Definition:** A subgraph formed by removing edges while maintaining connectivity and avoiding cycles.
- **Properties:**
  - Must have a minimum of (V-1) edges (V = vertices).
  - A graph can have multiple spanning trees.



Minimum Spanning Tree (MST):

- **Objective:** To find the spanning tree with the minimum total edge weight.
- **Algorithms:**
  - **Prim's Algorithm:** Builds the MST by selecting the edge with the smallest weight at each step.
  - **Kruskal's Algorithm:** Sorts all the edges and adds them to the MST in ascending order of weight.
  - **Shortest Path:** Dijkstra's algorithm is used to find the shortest distance from a source vertex to all other vertices in a connected graph.