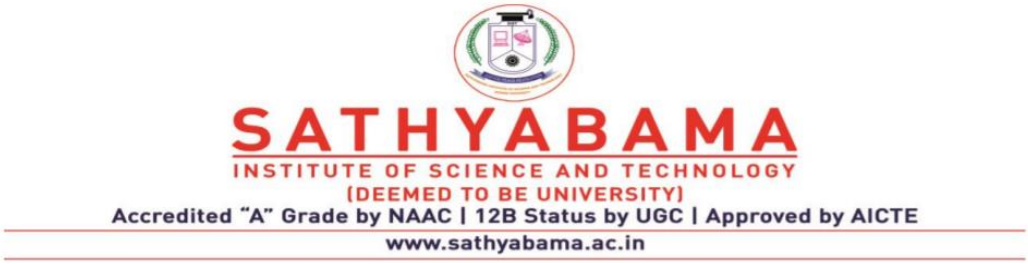


SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND INSTRUMENTAION ENGINEERING

UNIT III – SCSA1206 – C AND DATA STRUCTURES

UNIT – III

SEARCHING AND SORTING TECHNIQUES

SEARCHING:

- Searching algorithms are designed to check whether an element is present or not from any data structure where it is stored.
- Given, group of elements as and key element to be searched as input, any searching algorithm technique gives the position of the key element as the output.
- They are categorized into two types,

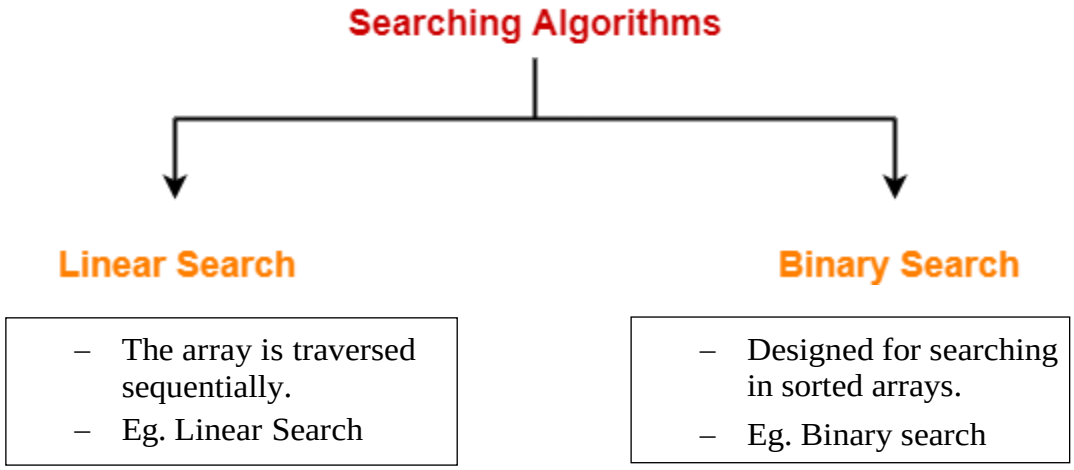


Fig. 3.1 Categorizes of Searching Algorithm

LINEAR SEARCH:

- Linear search is one of the sequential search algorithms, where the search starts from one end of the array and the target element is searched by comparing it with all the elements of array one by one in a sequence.

Working Steps:

1. Traverse the whole array using for loop
2. During every iteration, compare the target value with current value of the array element
   2.1 If the value matches, return the current index of the array.
   2.2 If the value doesn't match, move onto the next array element.
3. If no match is found return -1.

Algorithm linearsearch (arr [], target)

1. declare arr[], target variable.

2. for i = 0 to n-1
3. if arr[i]==target
4. return i
5. return -1
6. end

Working of Algorithm:

Example array:

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]
55	44	11	22	99	77

target =99

Table 3.1 Working of Linear Search Algorithm

i	arr [i]==target?	Condition Satisfied?
0	Check if 55==99	No
1	Check if 44==99	No
2	Check if 11==99	No
3	Check if 22==99	No
4	Check if 99==99	Yes

Advantage:

- Very simple and easy to implement.

Disadvantage:

- Consumes more time and space when compared with other searching algorithms.

Time Complexity:

- O(n) – n no. of comparisons is made to find the nth element.

BINARY SEARCH:

Binary Search is a searching algorithm that is used to search an element in a sorted array. It consumes less amount of time to search the element when compared with linear search.

Steps:

1. Find the mid element of the array using mid formula.
    a. If the target value is equal to middle element of the array return its index.

b. If not compare the mid element with target element.
   i. If the target value is greater than the number in middle index, pick elements from the right-side part of the array and start from step 1
   ii. If the target value is less than the number in middle index then pick elements from the left side part of the array and start from step 1

2. When the match is found, return the index of the corresponding element.
3. If not, return -1.

Algorithm binary search (arr [], target)

1. declare variables low, high, target //low=starting index, high=ending index
2. while low<high
3. mid = (low+high)/2
4. if arr[mid]<target
5. low = mid+1
6. else if arr[mid]>target
7. high = mid-1
8. else
9. return mid
10. return -1

Working of Algorithm:

Example array:

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]
55	44	11	22	99	77	111

target =99

Table 3.2 Working of Binary Search Algorithm

While low<high	**mid**	**Compare arr [mid] and target**	**low**	**high**
0<6	0+6/2 = 3	55<99	3	6
3<6	3+6/2 = 4	77<99	5	6
5<6	5+6/2 = 5	99==99	5	6

Advantage:

- Very fast and efficient when compared with other searching algorithms.

Disadvantage:

- The array or list taken should already be sorted in order to perform binary search.

Time Complexity:

- O(logn) – since, to find the nth element only less than or equal to n/2 comparisons are needed.

FIBONACCI SEARCH:

- Fibonacci Search Algorithm is a technique used to search an element in a sorted array.
- Here, array is searched by dividing it in terms of Fibonacci sequence numbers.

Algorithm fibonccisearch (arr [], target, n)

1. f0=0
2. f1=1
3. fm=f0+f1
4. offset=0
5. while fm<n
6. f0=f1
7. f1=fm
8. fm=f0+f1
9. while fm>0
10. i = min(offset+f0,fm)
11. if arr[i]<target
12. fm=f1
13. f1=f0
14. f0=fm-1
15. offset=i
16. else if arr[i]>target
17. fm=f1
18. f1=f0
19. f0=fm-1
20. else
21. return i

Working of Algorithm:

Example array:

arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]	arr[7]	arr[8]	arr[9]	arr[10]
11	22	33	44	55	66	77	88	99	111

target =77, n=10, f0=0, f1=1, fm=1 First while loop while fm<n

Table 3.3 Working of Fibonacci Search Algorithm

f0	f1	fm	Condition Satisfied?
0	1	1	1<10 Yes

1	1	2	2<10 Yes
1	2	3	3<10 Yes
2	3	5	5<10 Yes
3	5	8	8<10 Yes
5	8	13	13<10 No

Table 3.4 Final Values of Fibonacci Search Algorithm

f0	f1	fm	Offset	i	Compare arr [i] and target	Condition satisfied?
5	8	13	0	Min (5+0, 10) = 5	55<77	13>0 Yes
3	5	8	5	Min (5+3, 10) = 8	88>77	8>0 Yes
2	3	5	5	Min (5+2, 10) = 7	77==77	Return 7

Steps:

- Initial array's size is 13. For easy explanation array's index has been considered to start from one.
- At the end of first while loop, the array is divided into two parts consisting of five and eight elements each. (Fibonacci series number)
- And then the target element is compared with the last element of the array's first part. If it is lesser then the first array part is divided into two parts (based on Fibonacci numbers again)
- If the target element is greater than the second part is again divided into two parts. Whenever the target element and the array element become equal the particular position is returned.

Advantage:

- Very fast and efficient when compared with other searching algorithms.

Disadvantage:

- The array or list taken should already be sorted in order to perform binary search.

Time Complexity:

- O(logn) – to find the nth element only less than or equal to n/2 comparisons are needed.

SORTING:

- A sorting algorithm is used to re arrange a given array or list of elements according to a comparison operator on the elements.
- The two major categories of sorting algorithms are
    - Internal Sorting
        - All the data that has to be sorted can be adjusted at a time in the main memory
        - Eg. Bubble sort, Heap Sort
    - External Sorting
        - Data that has to be sorted cannot be accommodated in the memory at the same time ans=d some has to be kept in auxiliary memory such as hard disk.
        - Eg. External Merge Sort
- They are sub categorized into

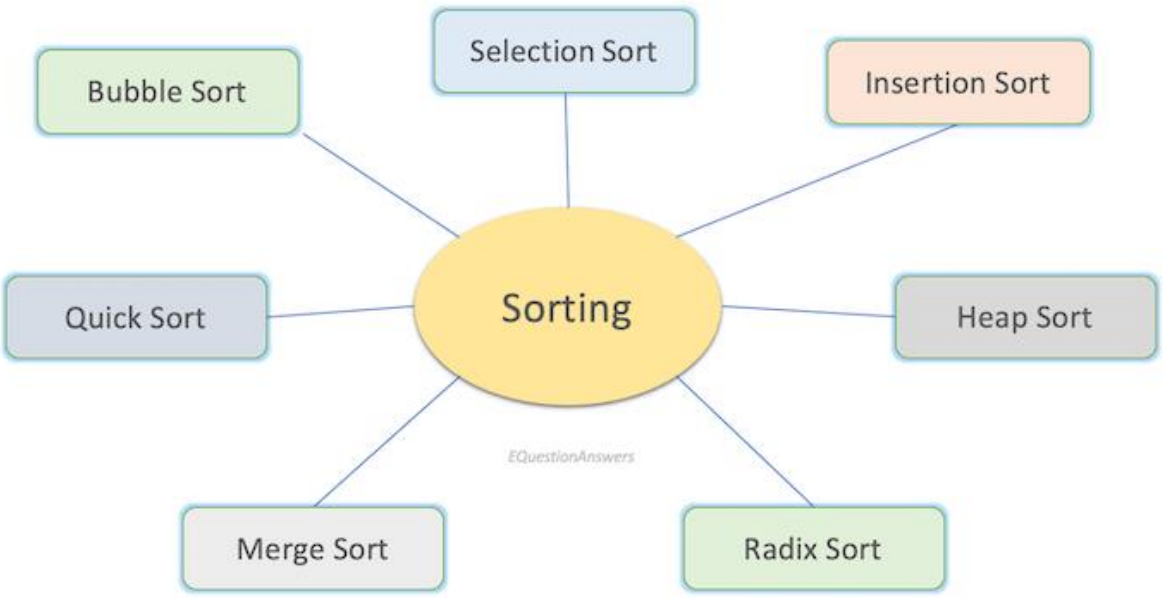


Fig. 3.2 Categorizes of Sorting Algorithm

BUBBLE SORT:

- Bubble Sort is the simplest sorting algorithm that involves repeatedly swapping the adjacent elements if they are not in the required order (ascending/descending)
- By swapping the adjacent elements, this technique bubbles out the largest element to the end and hence it was named as bubble sort.

Algorithm bubblesort (arr [], n)

1. for i=0 to n-1
2. for j=1 to n-1
3. if arr[j] < arr[j-1]
4. swap arr[j],arr[j-1]
5. return arr[]

Working of Algorithm:

Example array:

	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Before Sorting	90	50	80	10	30

Table 3.5 Working of Bubble Sort Algorithm

i	j	arr[i]	arr[j]	Swap?
0	1	50	90	Yes
0	2	80	90	Yes
0	3	10	90	Yes
0	4	30	90	Yes
1	1	80	50	No
1	2	10	80	Yes
1	3	30	80	Yes
1	4	90	80	No
2	1	10	50	Yes
2	2	30	50	Yes
2	3	80	50	No
2	4	90	80	No
3	1	30	10	No

3	2	50	30	No
3	3	80	50	No
3	4	90	80	No
4	1	30	10	No
4	2	50	30	No
4	3	80	50	No
4	4	90	80	No

After Sorting

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
10	30	50	80	90

Advantage:

- Simplest sorting algorithm and very easy to implement.

Disadvantage:

- Too slow and impractical when array is large.

Time Complexity:

- O(n2) – two for loops runs n*n times totally.

INSERTION SORT:

- Insertion sort is a simple algorithm that builds the final sorted array one by one.
- The lower part of the array is always sorted and any element to be inserted in this sorted sub array has to find its appropriate place to be inserted. And hence it's named as insertion sort.

Algorithm insertionsort (arr[], n, key, hole)

1. for i = 1 to n-1
2. key = arr[i]
3. hole=i
4. while hole>0 and arr[hole-1]>key
5. arr[hole] = arr[hole-1]
6. hole = hole-1

7. arr[hole] = key
8. return arr[]

Working of Algorithm:

Example array:

	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]
Before Sorting	90	50	80	10	30

I	key	hole	Hole>0	arr[hole-1]>key
1	50	1	Yes	Yes
1	50	0	No	-
2	80	2	Yes	Yes
2	80	1	Yes	No
3	10	3	Yes	Yes
3	10	2	Yes	Yes
3	10	1	Yes	Yes
3	10	0	No	-
4	30	4	Yes	Yes
4	30	3	Yes	Yes
4	30	2	Yes	Yes
4	30	1	Yes	No

Initial Array:

90	50	80	10	30
50	90	80	10	30
50		90	10	30
50	80	90	10	30
50	80		90	30
50		80	90	30
	50	80	90	30
10	50	80	90	30
10	50	80		90
10	50		80	90
10		50	80	90
10	30	50	80	90

Fig. 3.3 Working of Insertion Sort Algorithm

After Sorting	10	30	50	80	90

Advantage:

- Simple implementation and efficient for small datasets.

Disadvantage:

- Much less efficient for large datasets.

Complexity:

- O(n2) – two loops run n*n times for n variables.

Divide and Conquer Strategy:

- Divide and Conquer is an algorithm design strategy that works by recursively breaking down the given problem into two or more sub problems until it becomes to be solved directly.
- Eg. Quick Sort, Merge Sort.

QUICK SORT:

- Quick Sort is an efficient sorting algorithm that uses divide and conquer technique to sort the given set of elements in an array.
- It takes first/last element as pivot element and places the pivot element at its right place by comparing it with other elements.
- It recursively divides the array after pivot element reaches its appropriate place and the procedure is called again recursively for the remaining part of the array.

Algorithm Partition (arr[], start, end)

1. pivot = end
2. pindex = start
3. for i = start to n-2
4. if arr[i] < arr[pivot]
5. swap arr[i],arr[pindex]
6. pindex = pindex + 1
7. swap(arr[pivot],arr[pindex])
8. return pindex

Algorithm QuickSort (arr[],n,start,end)

1. if start<end
2. pindex = partition(arr[], start, end)
3. QuickSort(arr[], start, pindex-1)
4. QuickSort(arr[], pindex+1, end)

Working of Algorithm: Example array:

	arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]
Before Sorting	55	45	5	75	15	25

Partition algorithm: pivot = 5, pindex = 0

pivot	pindex	I	Compare arr[i] and arr[pivot]	Swap?	
5	0	0	55>25	No	55 45 5 75 15 25
5	0	1	45>25	No	55 45 5 75 15 25
5	0	2	5<25	Yes	5 45 55 75 15 25
5	1	3	75>25	No	5 45 55 75 15 25
5	1	4	15<25	Yes	5 15 55 75 45 25
Pindex = pindex + 1 = 1+1 = 2 Swap arr[pivot] and arr[pindex] and return pindex					5 15 25 75 45 55

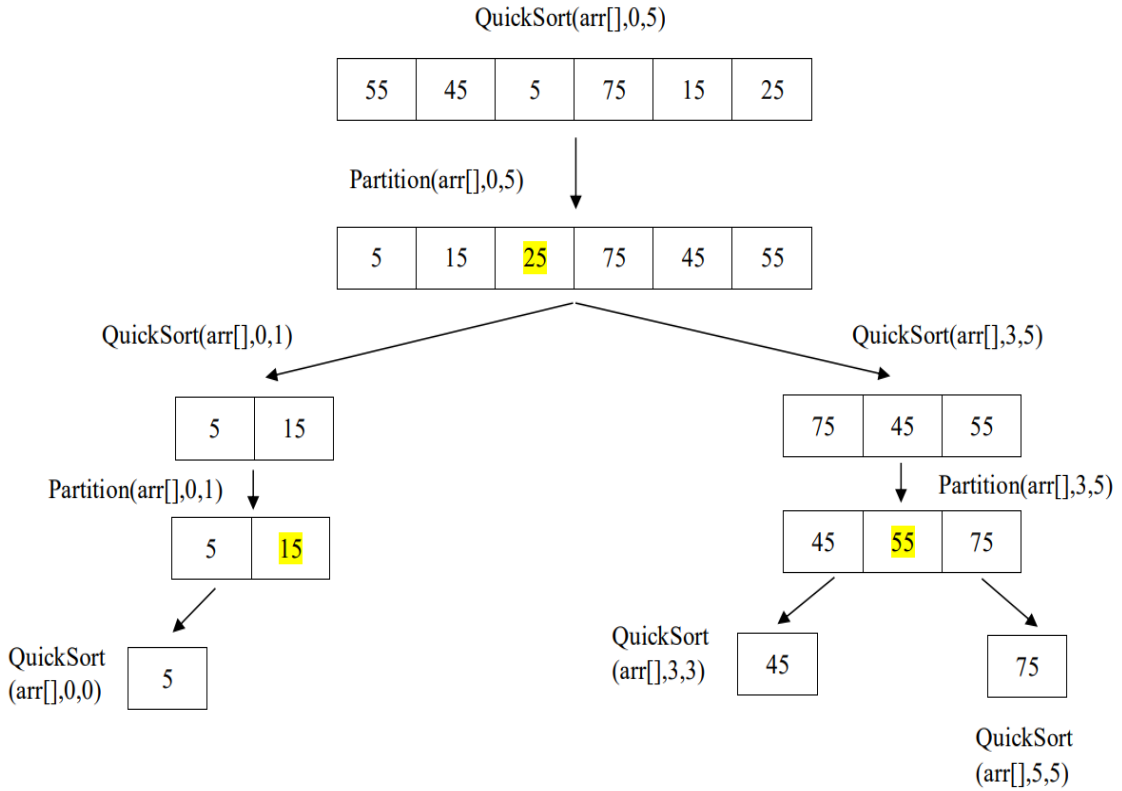


Fig. 3.4 Working of Quick Sort Algorithm

After Sorting

5	15	25	45	55	75

- The above tree explains the function call order for the considered example. The shaded number denotes the value of pindex after calling the partition sub routine.

Advantage:

- It is the most efficient sorting algorithm, when implemented well it can be three times faster than merge sort.

Disadvantage:

- In worst case, the number of comparisons will be more.

Complexity:

- O(nlogn): with n comparisons in partition algorithm and using divide and conquer strategy for recursive quicksort subroutine, the whole complexity sums to nlogn.

MERGE SORT:

- Merge Sort is a divide and conquer algorithm that recursively divides the array into sub arrays, sort those separately and merges together to get the final sorted array.
- It uses two functions, one for dividing and the other for sorting and merging.

Algorithm MergeSort (arr[], i, j, mid)

1. if i<j
2. mid = i+j/2
3. MergeSort(arr[],i,mid)
4. MergeSort(arr[],mid+1,j)
5. Merge(arr[],i,mid,mid+1,j)

Algorithm Merge(arr[], i1, i2, j1, j2)

1. declare temp[50],i,j,k
2. i=i1,j=j2,k=0
3. while i<=j1 and j<=j2
4. if arr[i]<arr[j]
5. temp[k] = arr[i]
6. k++, i++
7. else
8. temp[k] = arr[j]
9. k++,j++
10. while i<=j1
11. temp[k] = arr[i]

12. k++, i++
13. while j<=j2
14. temp[k] = arr[j]
15. k++, j++
16. move elements from temporary array to arr[i]

Algorithm Working:

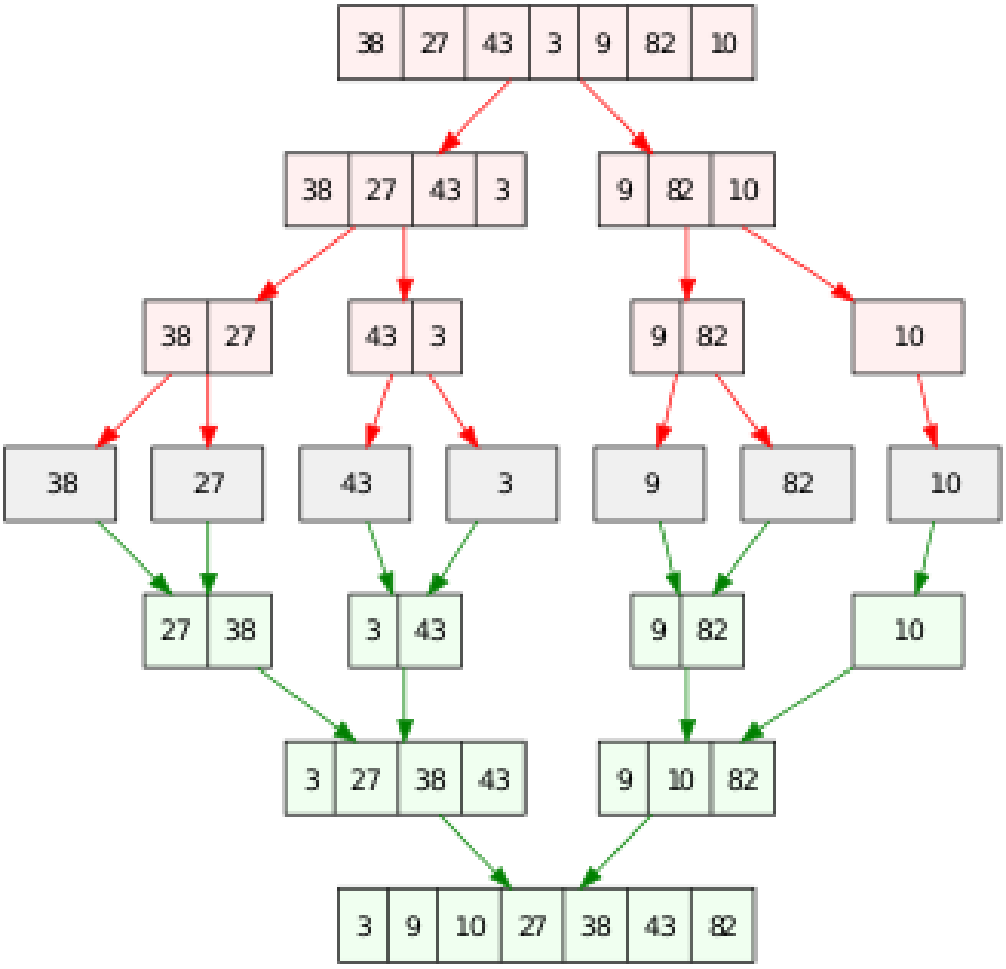


Fig. 3.5 Working of Merge Sort Algorithm

Advantage:

- Efficient general-purpose algorithm.

Disadvantage:

- In some cases, its performance is less when compared to quicksort.

Complexity:

- O(nlogn) - with n comparisons in merge algorithm and using divide and conquer strategy for recursive mergesort subroutine, the whole complexity sums to nlogn.

HEAP SORT:

- Heap Sort is a comparison-based sorting algorithm that uses heap data structure rather than linear time search to find the maximum.
- Here, the array elements are thought of as binary heap tree (where parent node is the maximum at all levels of the tree) and elements are sorted by changing places.

Algorithm HeapSort (arr[], n)

1. for i = n/2 – 1 to 0
2. heapify(arr[],i,n)
3. for i = 0 to n-1
4. swap arr[0] and arr[n-1]
5. heapify(arr[],i,0)

Algorithm heapify (arr[], n, i)

1. largest = i
2. left = 2i+1
3. right = 2i+2
4. if left<n and arr[left] > arr[largest]
5. largest = left
6. else if right<n and arr[right] > arr[largest]
7. largest = right
8. if largest != i
9. swap arr[i] and arr[largest]
10. heapify(arr[],n,largest)

Algorithm Working:

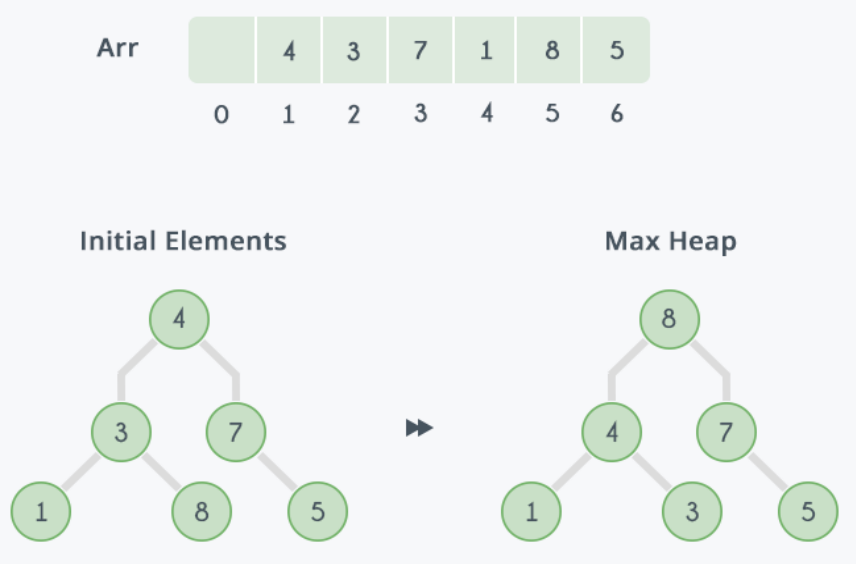


Fig. 3.6 Construction of Max Heap

After building max-heap, the elements in the array Arr will be:

Arr		8	4	7	1	3	5
	0	1	2	3	4	5	6

Step 1: 8 is swapped with 5.

Step 2: 8 is disconnected from heap as 8 is in correct position now and.

Step 3: Max-heap is created and 7 is swapped with 3.

Step 4: 7 is disconnected from heap.

Step 5: Max heap is created and 5 is swapped with 1.

Step 6: 5 is disconnected from heap.

Step 7: Max heap is created and 4 is swapped with 3.

Step 8: 4 is disconnected from heap.

Step 9: Max heap is created and 3 is swapped with 1.

Step 10: 3 is disconnected.

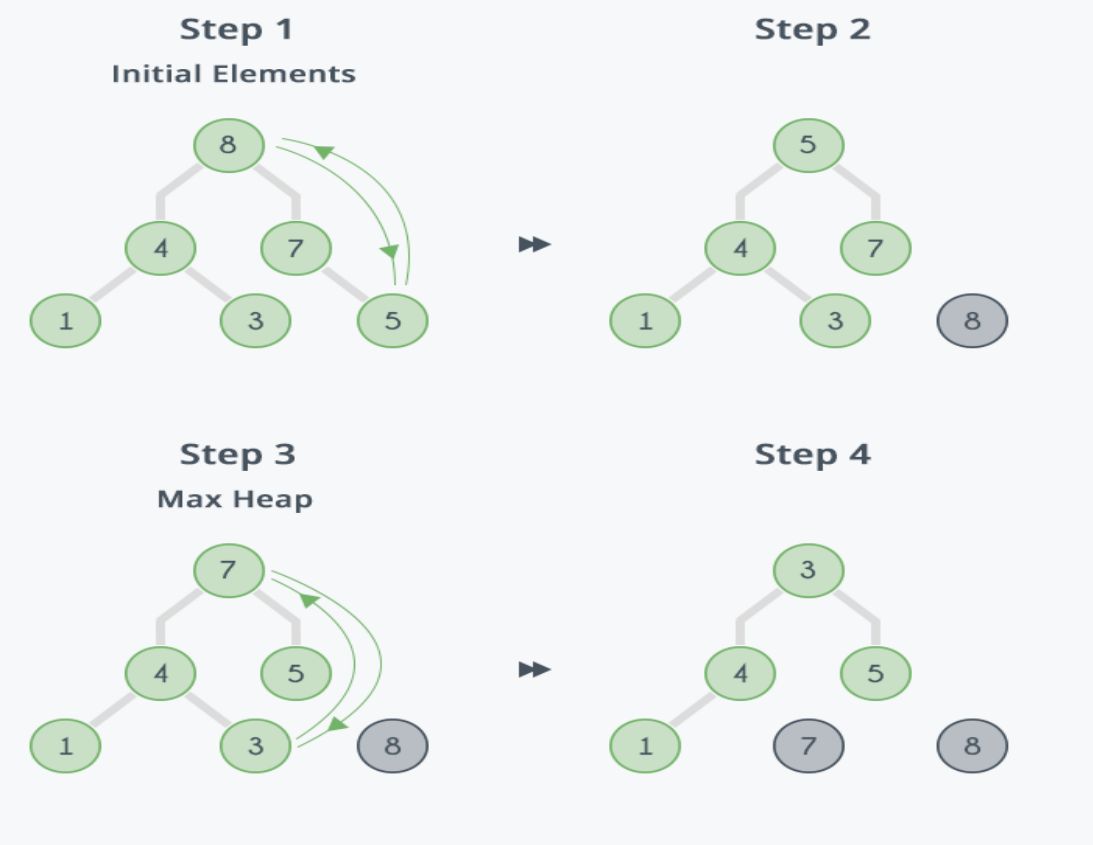


Fig. 3.7 Working of Heap Sort Algorithm (Step 1 to Step 4)

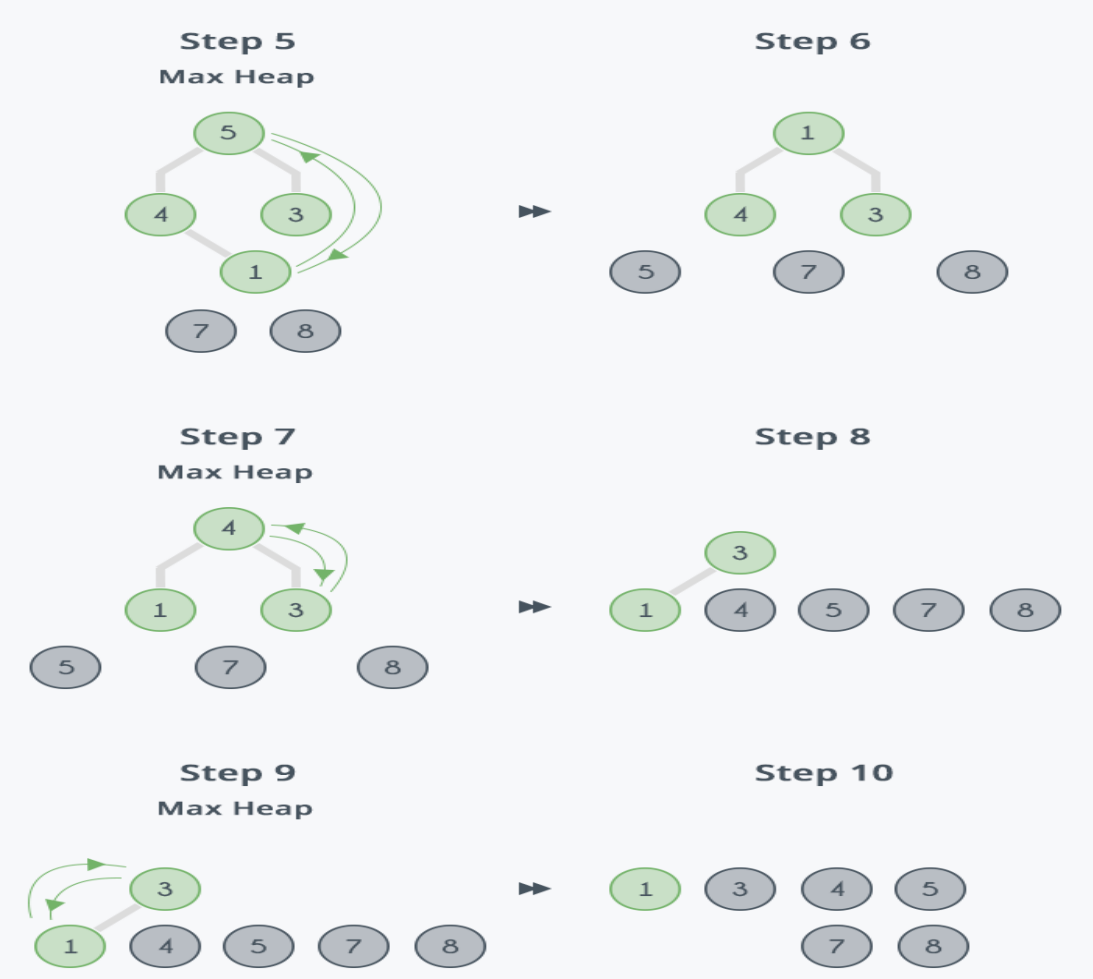


Fig. 3.6 Working of Heap Sort Algorithm (Step 5 to Step 10)

After all the steps, we will get a sorted array.

Arr		1	3	4	5	7	8
	0	1	2	3	4	5	6

Fig. 3.7 Sorted Array of Heap Sort Algorithm

Advantage:

- Very efficient and has a favourable worst-case complexity.

Disadvantages:

- Somewhat slower in practice in most of the machines.

Complexity:

- O(nlogn)

ANALYSIS OF SORTING TECHNIQUES:

- All the Sorting algorithms are used for single objective, to sort the elements in any sequential order, but still there are so many different algorithms available for sorting.
- This is because of the fact that each sorting algorithm has its own advantage and disadvantages, to overcome the disadvantage of the previous algorithm technique new technique is invented each time.
- To measure the efficiency of the sorting algorithms, time and space complexity is computed for each one separately.
- Time Complexity: total amount of time consumed to run the algorithm.
- Space Complexity: total amount of space computed by the algorithm.
- Since the space consumed might vary from machine to machine due to different hardwares used, time complexity is usually preferred to check the efficiency of the algorithm.
- The time complexity can be found for best case (input might be sorted already), average case (input will have randomly distributed elements) and worst case (input will have elements in reverse order)

Table 3.6 Analysis of Sorting Algorithms

Sorting Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	O(n)	O(n2)	O(n2)
Insertion Sort	O(n)	O(n2)	O(n2)
Quick Sort	O(nlogn)	O(nlogn)	O(n2)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)

SHELL SORT:

- Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.
- This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval. This interval is calculated based on Knuth's formula as −

Knuth's Formula

- h = h * 3 + 1where − h is interval with initial alue 1

- This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is O(n), where n is the number of items. And the worst case space complexity is O(n).

Algorithm

- Following is the algorithm for shell sort.
    - Step 1 − Initialize the value of h
    - Step 2 − Divide the list into smaller sub-list of equal interval h
    - Step 3 − Sort these sub-lists using insertion sort
    - Step 3 − Repeat until complete list is sorted

Algorithm Working:

- Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}

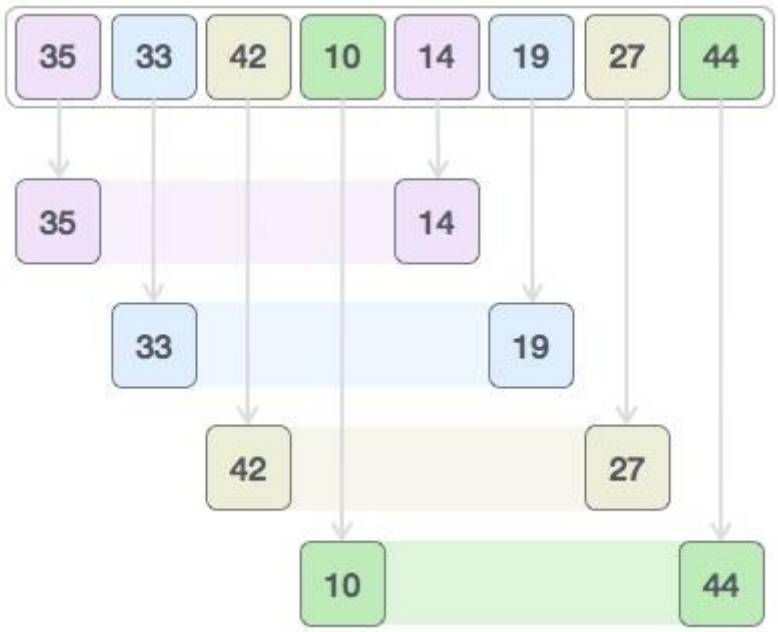
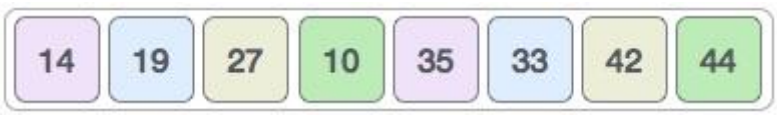
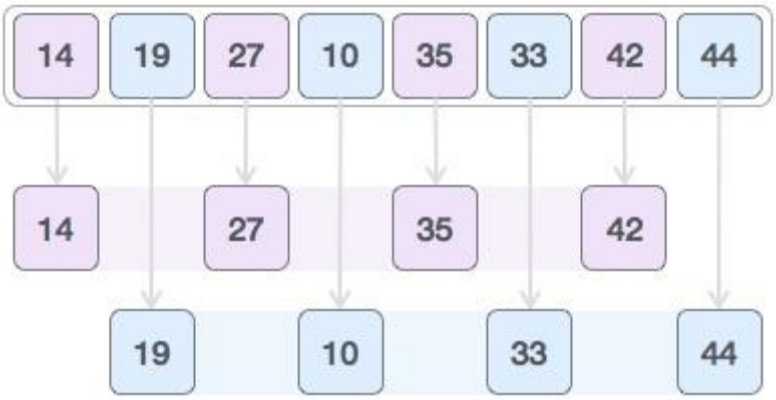


Fig. 3.8 Working of Shell Sort Algorithm

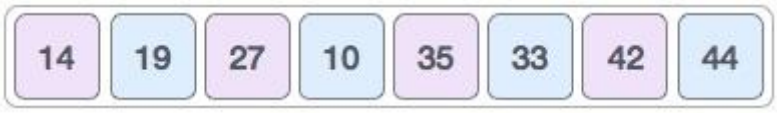
- We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −



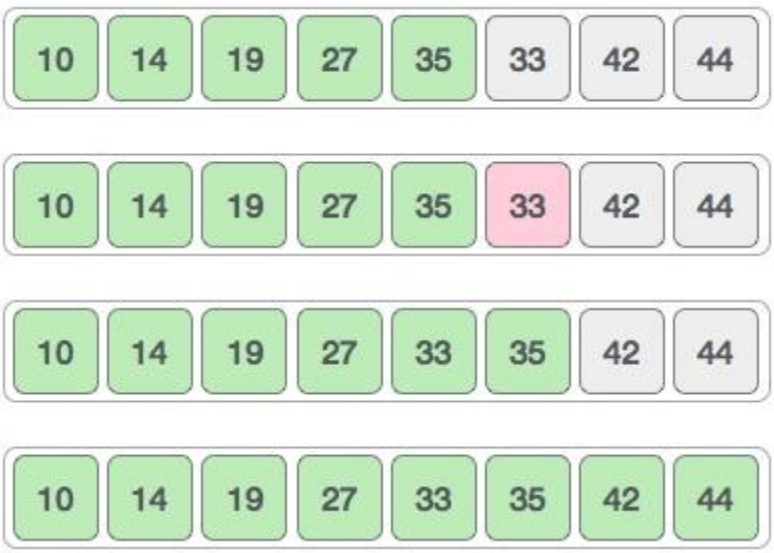
- Then, we take interval of 1 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



- We compare and swap the values, if required, in the original array. After this step, the array should look like this −



- Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.
- Following is the step-by-step depiction –



- We see that it required only four swaps to sort the rest of the array.

SELECTION SORT:

- Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.
- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.
- This algorithm is not suitable for large data sets as its average and worst-case complexities are of O(n2), where n is the number of items.

Algorithm

- Step 1 − Set MIN to location 0
- Step 2 − Search the minimum element in the list
- Step 3 − Swap with value at location MIN
- Step 4 − Increment MIN to point to next element
- Step 5 − Repeat until list is sorted

How Selection Sort Works?

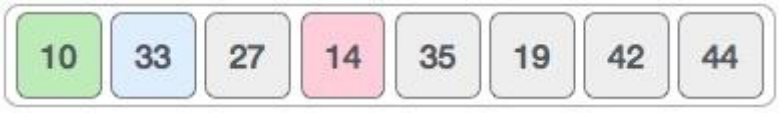
- Consider the following depicted array as an example.

- For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

- So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

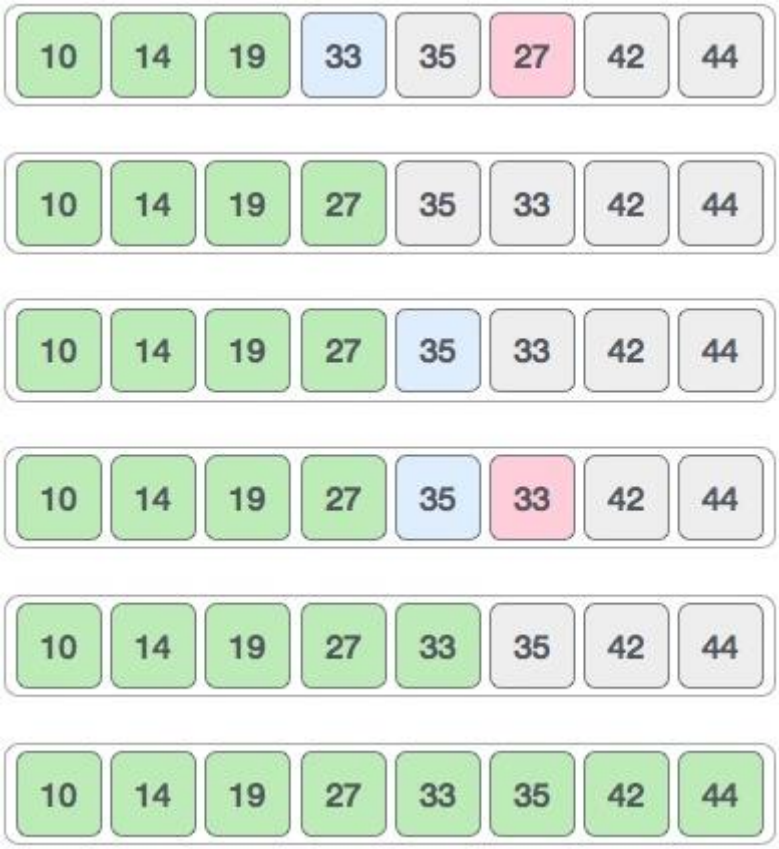
- For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

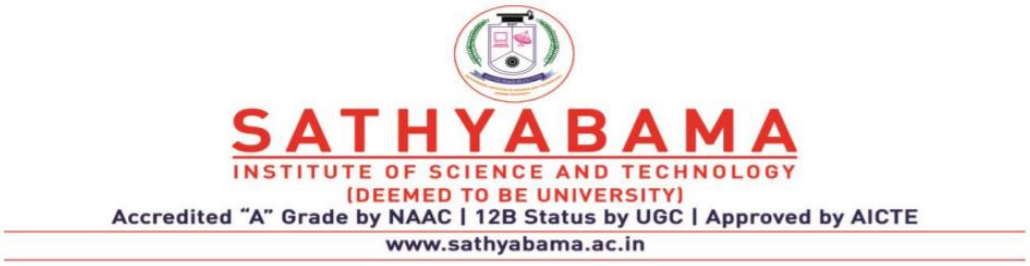
- We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



- After two iterations, two least values are positioned at the beginning in a sorted manner.

- The same process is applied to the rest of the items in the array. Following is a pictorial depiction of the entire sorting process –


10 14 19 33 35 27 42 44
10 14 19 27 35 33 42 44
10 14 19 27 35 33 42 44
10 14 19 27 35 33 42 44
10 14 19 27 33 35 42 44
10 14 19 27 33 35 42 44



SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND INSTRUMENTATION ENGINEERING

UNIT – IV – SCSA1206 – C AND DATA STRUCTURES

UNIT IV

LINEAR DATA STRUCTURES

ABSTRACT DATA TYPES (ADTs)

- The Data Type is basically a type of data that can be used in different computer program. It signifies the type like integer, float etc, the space like integer will take 4-bytes, character will take 1-byte of space etc.
- The abstract datatype is special kind of datatype, whose behaviour is defined by a set of values and set of operations.
- The keyword "Abstract" is used, we can perform different operations. The ADT is made of with primitive datatypes, but operation logics are hidden.
- Some examples of ADT are Stack, Queue, List etc.
- Let us see some operations of those mentioned ADT −
    - Stack −
        - isFull(), This is used to check whether stack is full or not
        - isEmpry(), This is used to check whether stack is empty or not
        - push(x), This is used to push x into the stack
        - pop(), This is used to delete one element from top of the stack
        - peek(), This is used to get the top most element of the stack
        - size(), this function is used to get number of elements present into the stack
    - Queue −
        - isFull(), This is used to check whether queue is full or not
        - isEmpry(), This is used to check whether queue is empty or not
        - insert(x), This is used to add x into the queue at the rear end
        - delete(), This is used to delete one element from the front end of the queue
        - size(), this function is used to get number of elements present into the queue
    - List −
        - size(), this function is used to get number of elements present into the list
        - insert(x), this function is used to insert one element into the list
        - remove(x), this function is used to remove given element from the list
        - get(i), this function is used to get element at position i
        - replace(x, y), this function is used to replace x with y value

LIST ADT

- The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers and address of compare function needed to compare the data in the list.

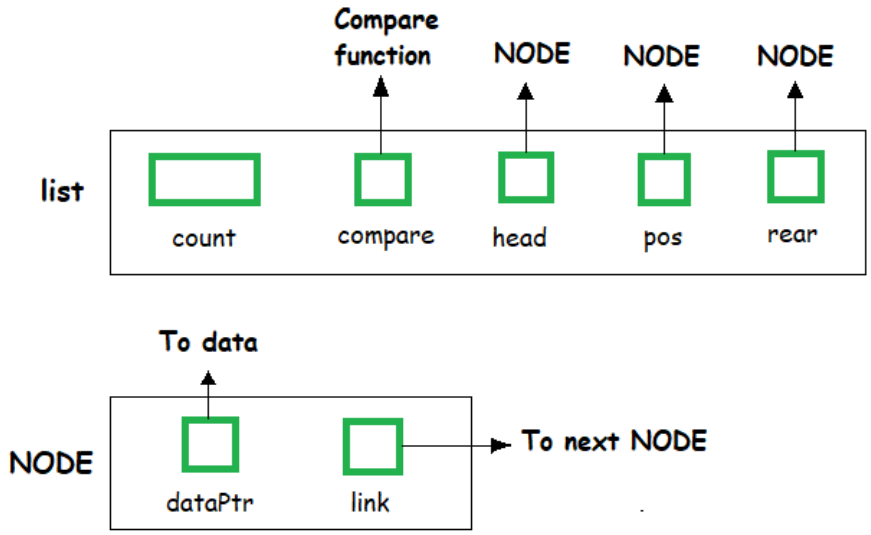


Fig. 4.1 Structure of a List ADT

- The data node contains the pointer to a data structure and a self-referential pointer which points to the next node in the list.

```
//List ADT Type Definitions
typedef struct node
{
 void *DataPtr;
 struct node *link;
} Node;
typedef struct
{
 int count;
 Node *pos;
 Node *head;
 Node *rear;
 int (*compare) (void *argument1, void *argument2)
} LIST;
```

- The List ADT Functions is given below:
    - A list contains elements of the same type arranged in sequential order and following operations can be performed on the list.
        - get() – Return an element from the list at any given position.
        - insert() – Insert an element at any position of the list.
        - remove() – Remove the first occurrence of any element from a non-empty list.
        - removeAt() – Remove the element at a specified location from a non-empty list.
        - replace() – Replace an element at any position by another element.
        - size() – Return the number of elements in the list.
        - isEmpty() – Return true if the list is empty, otherwise return false.
        - isFull() – Return true if the list is full, otherwise return false.

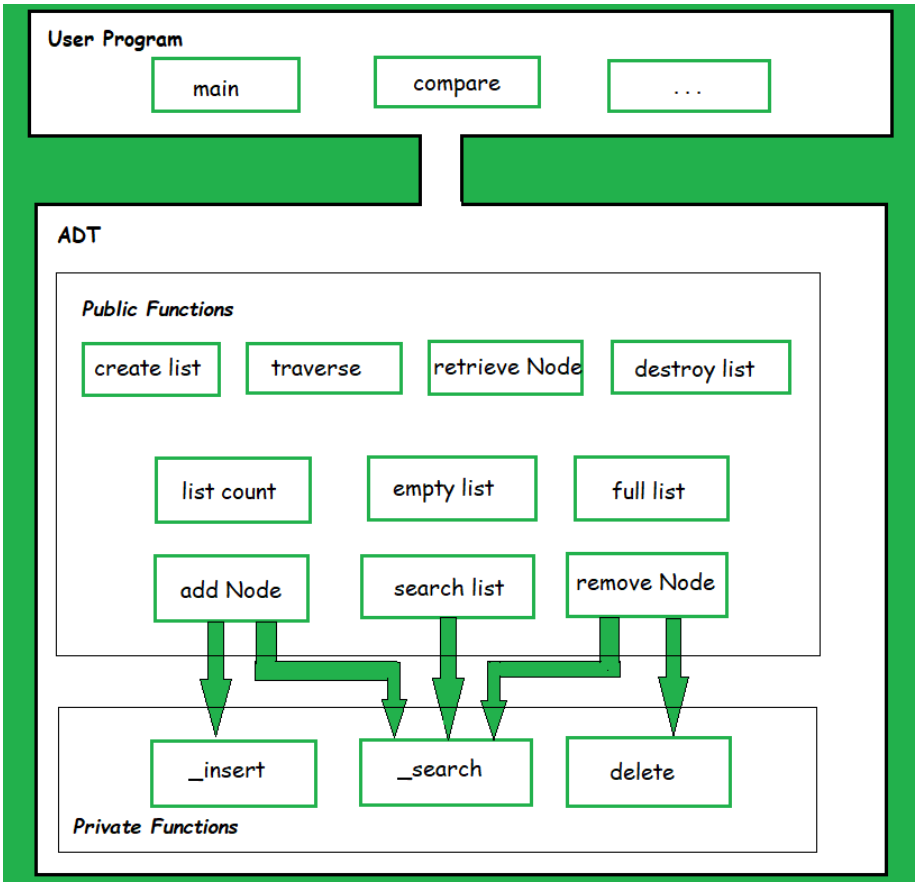


Fig. 4.2 Operations of List ADT

Linked List

- A linked list is a sequence of data structures, which are connected together via links.
- Linked List is a sequence of links which contains items. Each link contains a connection to another link.

- Linked list is the second most-used data structure after array.
- Following are the important terms to understand the concept of Linked List.
    - Link − Each link of a linked list can store a data called an element.
    - Next − Each link of a linked list contains a link to the next link called Next.
    - LinkedList − A Linked List contains the connection link to the first link called First.

Linked List Representation

- Linked list can be visualized as a chain of nodes, where every node point to the next node.

- As per the above illustration, following are the important points to be considered.
    - Linked List contains a link element called first.
    - Each link carries a data field(s) and a link field called next.
    - Each link is linked with its next link using its next link.
    - Last link carries a link as null to mark the end of the list.

Types of Linked List

- Following are the various types of linked list.
    - Simple Linked List − Item navigation is forward only.
    - Doubly Linked List − Items can be navigated forward and backward.
    - Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

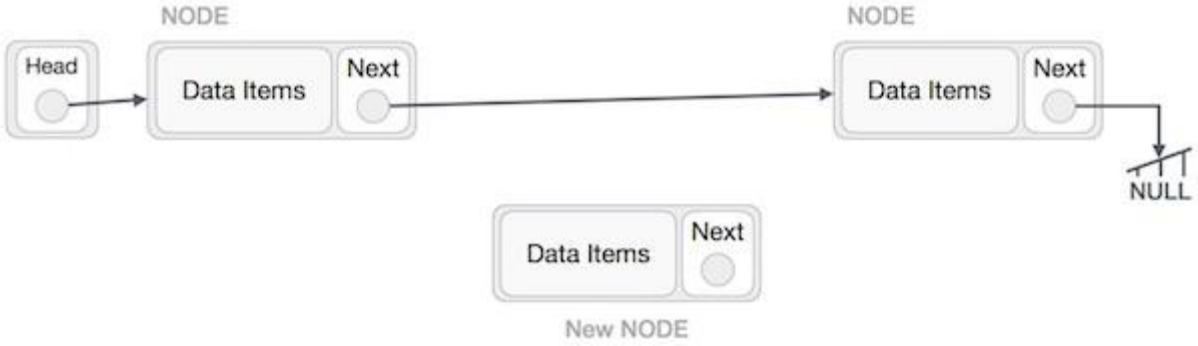
Basic Operations

- Following are the basic operations supported by a list.
    - Insertion − Adds an element at the beginning of the list.
    - Deletion − Deletes an element at the beginning of the list.
    - Display − Displays the complete list.
    - Search − Searches an element using the given key.
    - Delete − Deletes an element using the given key.

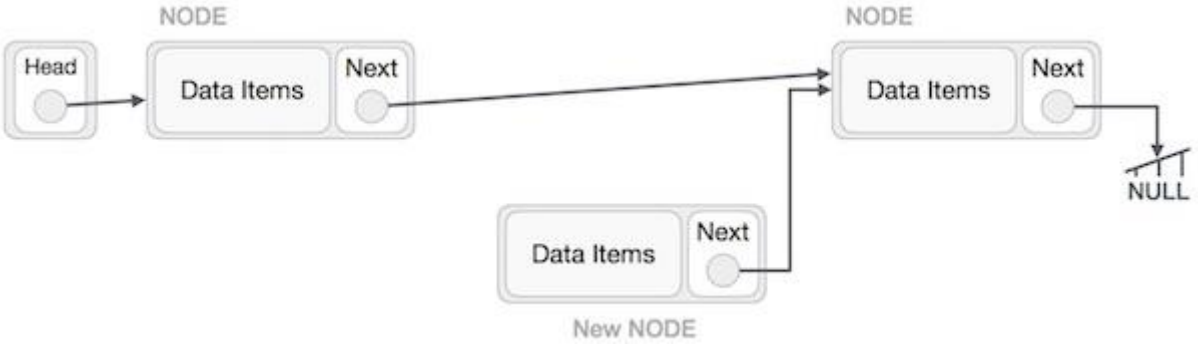
Insertion Operation

- Adding a new node in linked list is a more than one step activity. We shall learn this

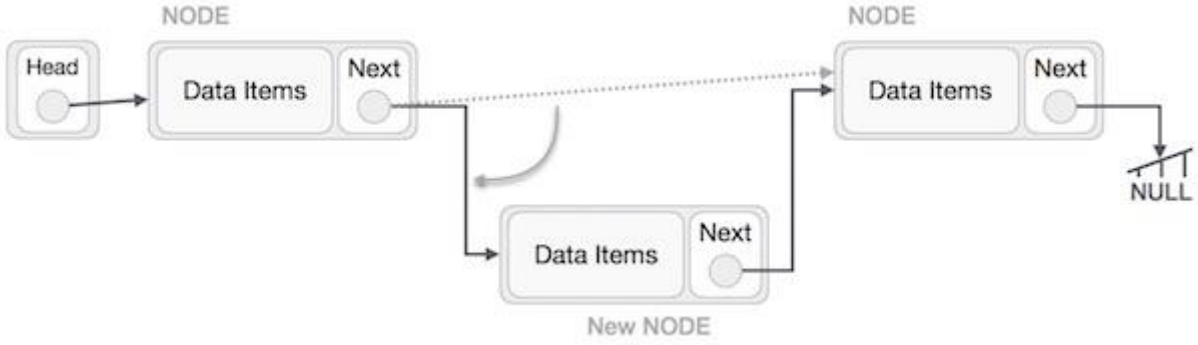
with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



- Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C −
- NewNode.next −> RightNode;
- It should look like this −



- Now, the next node at the left should point to the new node.
- LeftNode.next −> NewNode;



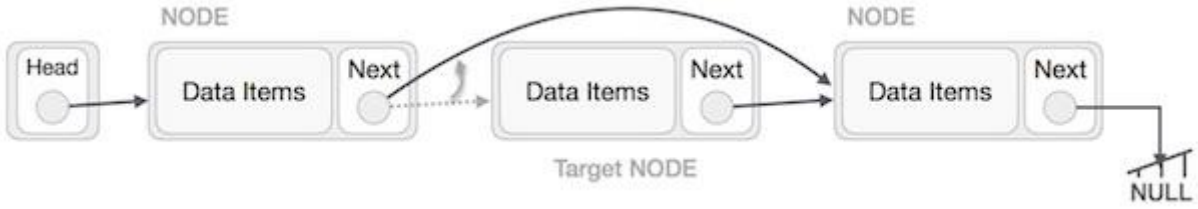
- This will put the new node in the middle of the two. The new list should look like this –

- Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation

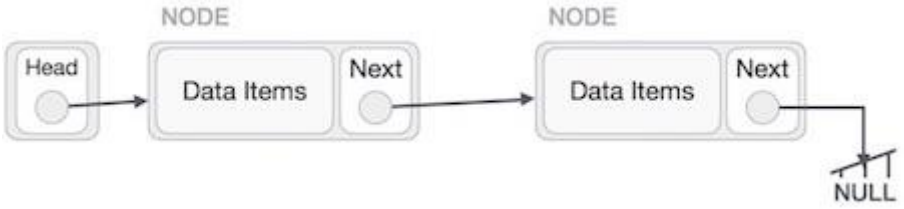
- Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

- The left (previous) node of the target node now should point to the next node of the target node
- LeftNode.next −> TargetNode.next;



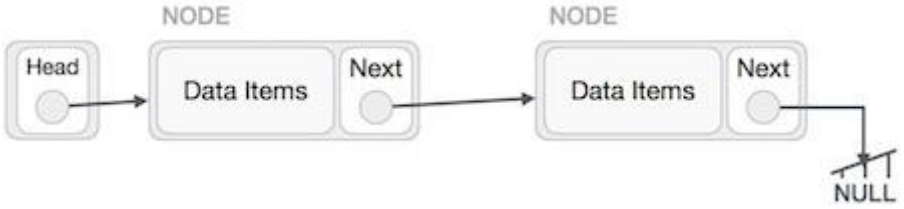
- This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.
- TargetNode.next −> NULL;

- We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

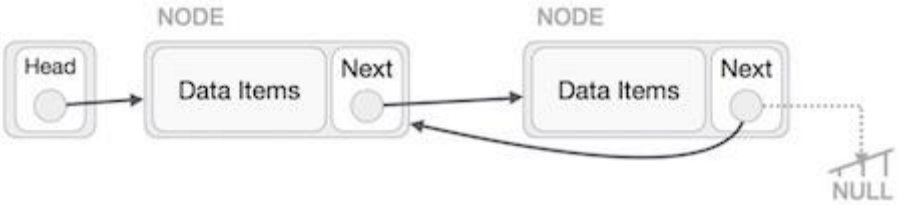


Reverse Operation

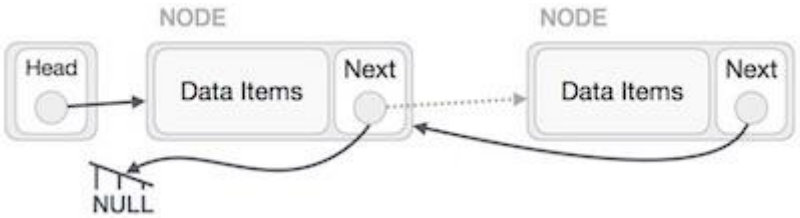
- This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



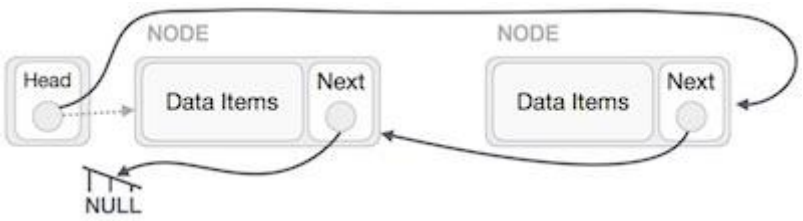
- First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



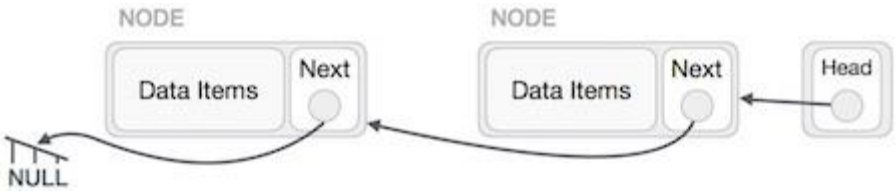
- We have to make sure that the last node is not the last node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



- Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



- We'll make the head node point to the new first node by using the temp node.

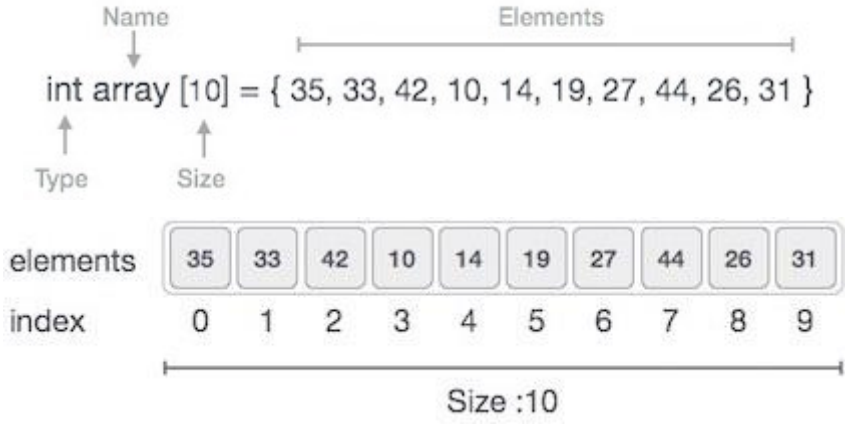


Arrays

- Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms.
- Following are the important terms to understand the concept of Array.
    - Element − Each item stored in an array is called an element.
    - Index − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

- Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



- As per the above illustration, following are the important points to be considered.
    - Index starts with 0.
    - Array length is 10 which means it can store 10 elements.
    - Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- Traverse − print all the array elements one by one.
- Insertion − Adds an element at the given index.

- Deletion − Deletes an element at the given index.
- Search − Searches an element using the given index or by the value.
- Update − Updates an element at the given index.

IMPLEMENTATION OF AN ADT

- The first step in the implementation is the choice of the data structure to represent the ADT's data.
- This choice of the data structure depends mainly on details of the ADT's operations and the context in which the operations will be used.
- The first step in implementing an ADT is choosing a data structure such as arrays, records, etc. to represent the ADT.
- Each operation associated with the ADT is implemented by one or more subroutines.
- Two standard implementations for the list ADT that we will be discussing are
    - Array-based implementation
    - Linked list-based implementation.

ARRAY IMPLEMENTATION OF ADT LIST

- The simplest method to implement a List ADT is to use an array that is a "linear list" or a "contiguous list" where elements are stored in contiguous array positions.
- The implementation specifies an array of a particular maximum length, and all storage is allocated before run-time. It is a sequence of n-elements where the items in the array are stored with the index of the array related to the position of the item in the list.
- In array implementation, elements are stored in contiguous array positions (Figure 4.3). An array is a viable choice for storing list elements when the elements are sequential, because it is a commonly available data type and in general algorithm development is easy.

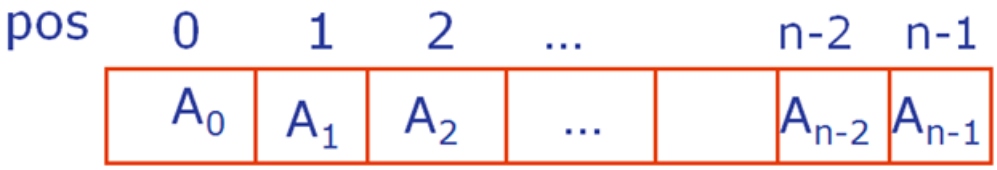


Fig. 4.3 Array Implementation of List ADT

- List is a sequence of zero or more elements of a given type A_0, A_1, ..., A_{n-1}
- n → length of the list
- A_0 → first element of the list
- A_{n-1} → last element of the list

- If n = 0, the list is empty.
- Elements can be linearly ordered according to their position in the list

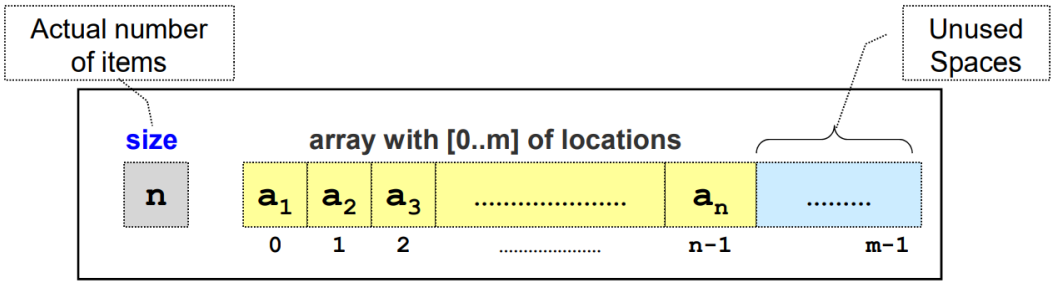


Fig. 4.4 Internal of List ADT - Array Implementation

- We say a_i precedes $a_i + 1$, $a_i + 1$ follows a_i, and a_i is at position i
- Let us assume the following:
- L → list of objects of type element
- x → an object of this type
- p → of type position
- END (L) → a function that returns the position following the last position in the list L

Operations:

1. **Insert (x, p, L)**

   Insert x at position p in list L

   If p = END(L), insert x at the end

   If L does not have position p, result is undefined

- Simplest Case: Insert to the end of array
- Other Insertions:
  - Some items in the list needs to be shifted
  - Worst case: Inserting at the head of array

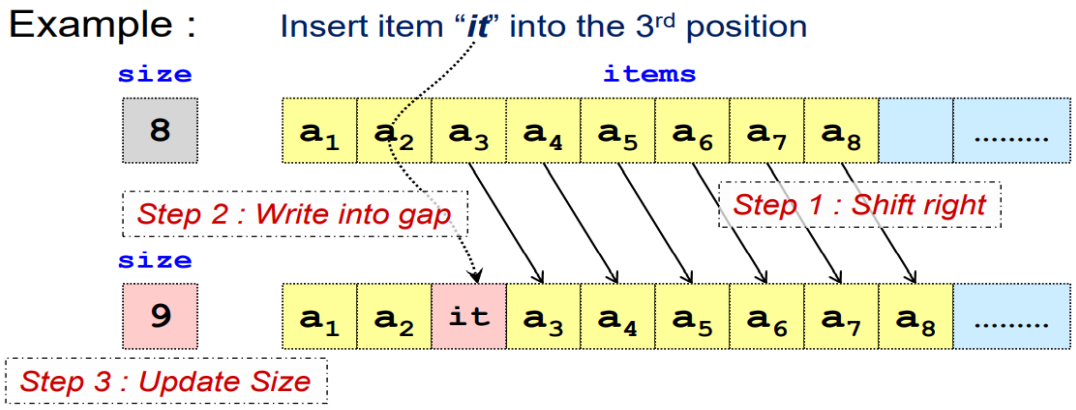


Fig. 4.5 Insertion of an element in List

2. **Locate (x, L)**

   returns position of x on L

   returns END(L) if x does not appear

3. **Retrieve (p, L)**

   returns element at position p on L

   undefined if p does not exist or p = END(L)

4. **Delete (p, L)**

   delete element at position p in L

   undefined if p = END(L) or does not exist

- Simplest Case: Delete item from the end of array
- Other deletions:
  - Items needs to be shifted
  - Worst Case: Deleting at the head of array

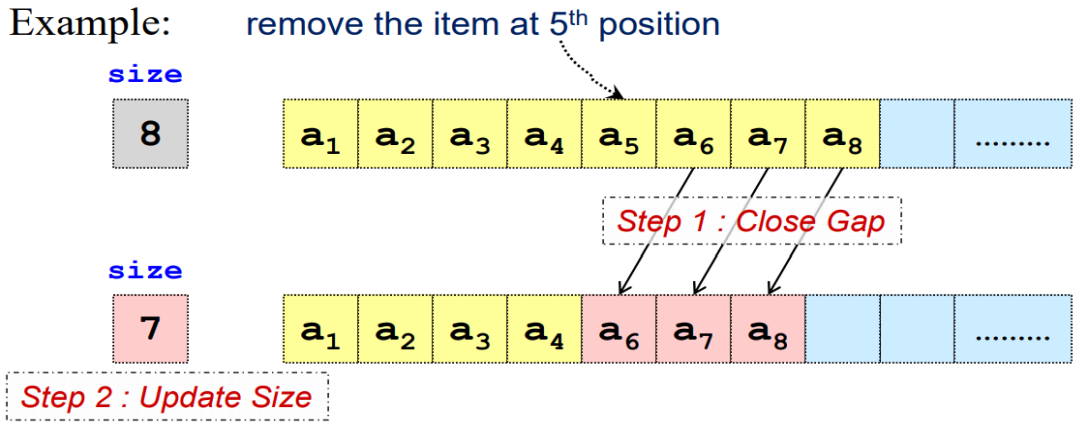


Fig. 4.6 Deletion of an element in List

5. **Next (p, L)**

   returns the position immediately following position p

6. **Prev (p, L)**

   returns the position previous to p

7. **Makenull (L)**

   causes L to become an empty list and returns position END(L)

8. **First (L)**

   returns the first position on L

9. **Printlist (L)**

   print the elements of L in order of occurrence

Advantages of Array-Based Implementation of Lists

Some of the major advantages of using array implementation of lists are:

- Array is a natural way to implement lists
- Arrays allow fast, random access of elements
- Array based implementation is memory efficient since very little memory is required other than that needed to store the actual contents

Some of the disadvantages of using arrays to implement lists are:

- The size of the list must be known when the array is created and is fixed (static)
- Array implementations of lists use a static data structure. Often defined at compile-time. This means the array size or structure cannot be altered while program is running. This requires an accurate estimate of the size of the array.
- This fixing of the size beforehand usually results in overestimation of size which means we tend to usually waste space rather than have program run out.
- The deletion and insertion of elements into the list is slow since it involves shifting of elements. It also means that data must be added to the end of the list for insertion and deletion to be efficient. If insertion and deletion is towards the front of the list, all other elements must shuffle down. This is slow and inefficient. This inefficiency is even more pronounced when the size of the list is large.

C Program to implement List ADT using Arrays.....

```
#include<stdio.h>
#include<conio.h>
#define MAX 10
void create();
void insert();
void deletion();
void search();
void display();
int a,b[20], n, p, e, f, i, pos;
void main()
{
//clrscr();
int ch;
```

```
char g='y';
do
{
printf("\n main Menu");
printf("\n 1.Create \n 2.Delete \n 3.Search \n 4.Insert \n 5.Display\n 6.Exit \n");
printf("\n Enter your Choice");
scanf("%d", &ch);
switch(ch)
{
case 1:
create();
break;
case 2:
deletion();
break;
case 3:
search();
break;
case 4:
insert();
break;
case 5:
display();
break;
case 6:
exit();
break;
default:
printf("\n Enter the correct choice:");
}
printf("\n Do u want to continue:::");
scanf("\n%c", &g);
```

```
}
while(g=='y'||g=='Y');
getch();
}
void create()
{
printf("\n Enter the number of nodes");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n Enter the  Element:",i+1);
scanf("%d", &b[i]);
}
}
void deletion()
{
printf("\n Enter the position u want to delete::");
scanf("%d", &pos);
if(pos>=n)
{
printf("\n Invalid Location::");
}
else
{
for(i=pos+1;i<n;i++)
{
b[i-1]=b[i];
}
n--;
}
printf("\n The Elements after deletion");
for(i=0;i<n;i++)
```

```
{
printf("\t%d", b[i]);
}
}
void search()
{
printf("\n Enter the Element to be searched:");
scanf("%d", &e);
for(i=0;i<n;i++)
{
if(b[i]==e)
{
printf("Value is in the %d Position", i);
}
else
{
printf("Value %d is not in the list::", e);
continue;
}
}
}
void insert()
{
printf("\n Enter the position u need to insert::");
scanf("%d", &pos);

if(pos>=n)
 {
 printf("\n invalid Location::");
 }
 else
 {
```

```
 for(i=MAX-1;i>=pos-1;i--)
 {
 b[i+1]=b[i];
 }
 printf("\n Enter the element to insert::\n");
 scanf("%d",&p);
 b[pos]=p;
 n++;
 }
 printf("\n The list after insertion::\n");
display();
}
void display()
{
printf("\n The Elements of The list ADT are:");
for(i=0;i<n;i++)
{
printf("\n\n%d", b[i]);
}
}
```

IMPLEMENTATION OF LIST ADT USING LINKED LIST

Pointer Based Linked List:

- Allow elements to be non-contiguous in memory
- Order the elements by associating each with its neighbour(s) through pointers

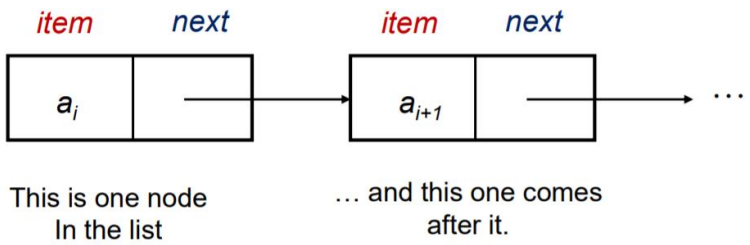


Fig. 4.7 Linked List Implementation of List ADT

A single node in the Linked List

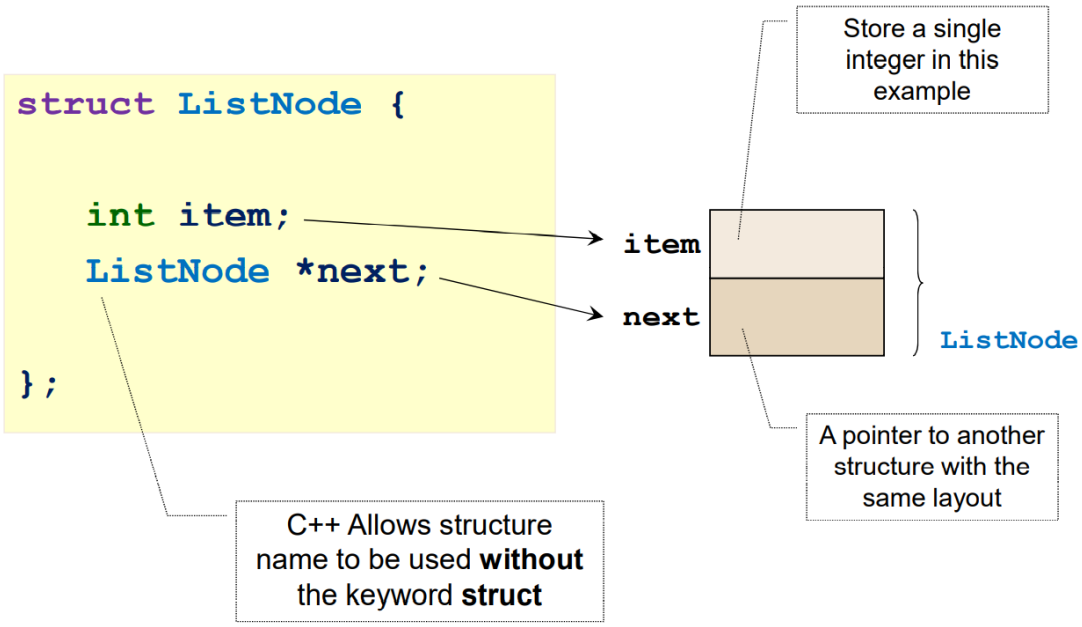


Fig. 4.8 Structure of a Node

- List of four items < a1, a2, a3, a4 >

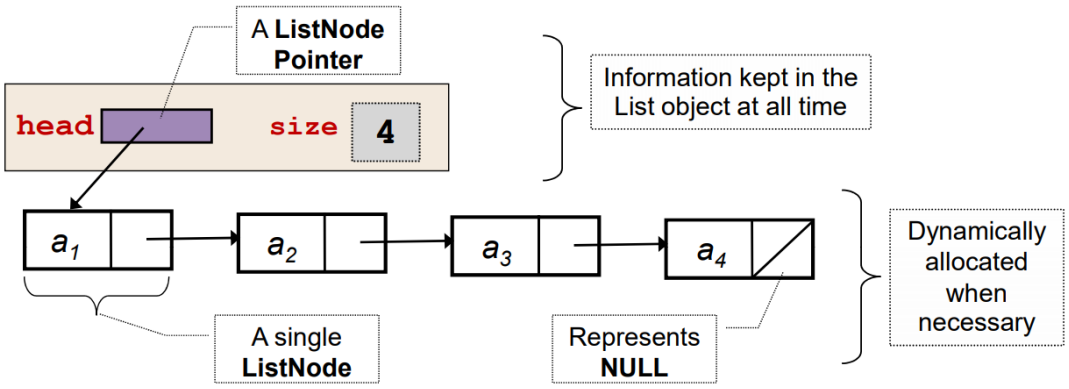


Fig. 4.9 Representation of List Elements in Linked List

- We need:
    - Head pointer to indicate the first node
    - Other nodes are accessed by "hopping" through the next pointer
    - Size for the number of items in the linked list
- Linked list implementation is more complicated: Need to handle a number of scenarios separately.

Linked List Insertion: General

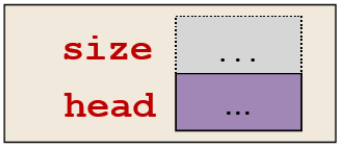
- List ADT provides the insert () method to add an item:
    - The new item itself is given

- The index [1…size+1] of the new item is given

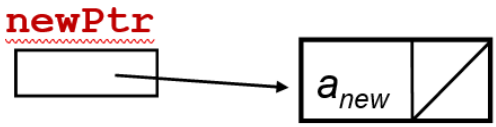
- Due to the nature of linked list, there are several possible scenarios:
  - Item is added to an empty linked list
  - Item is added to the head (first item) of the linked list
  - Item is added to the last position of the linked list
  - Item is added to the other positions of the linked list

Linked List Insertion: Preliminary

- The List object stores: Head pointer and the current size of linked list



- For all valid cases, we need to construct a new linked list node to store the new item



Insertion: Empty Liked List

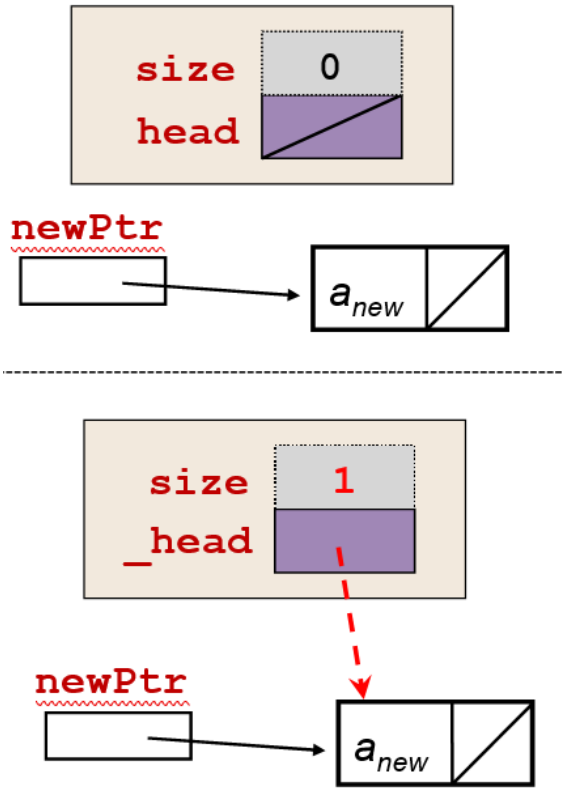


Fig. 4.10 Insertion: Empty Liked List

Insertion: Head of Linked List

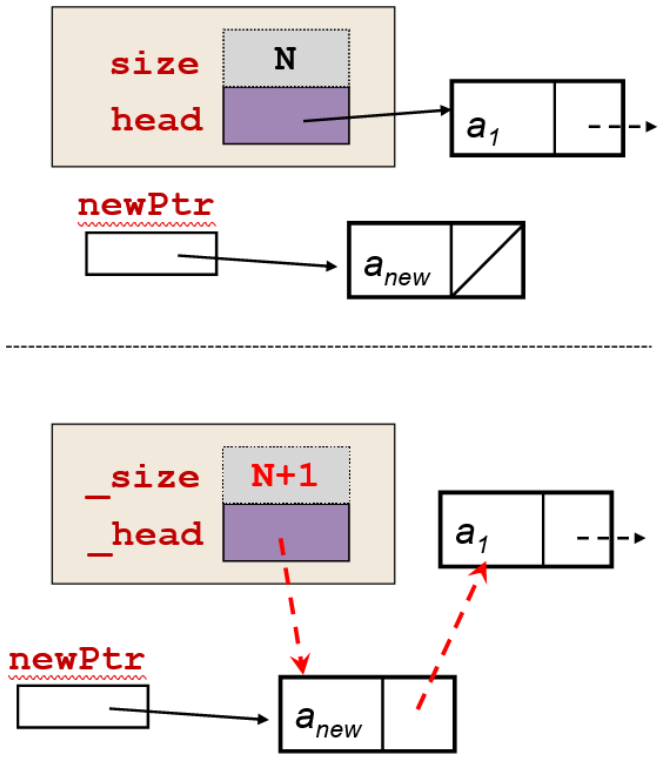


Fig. 4.11 Insertion: Head of Linked List

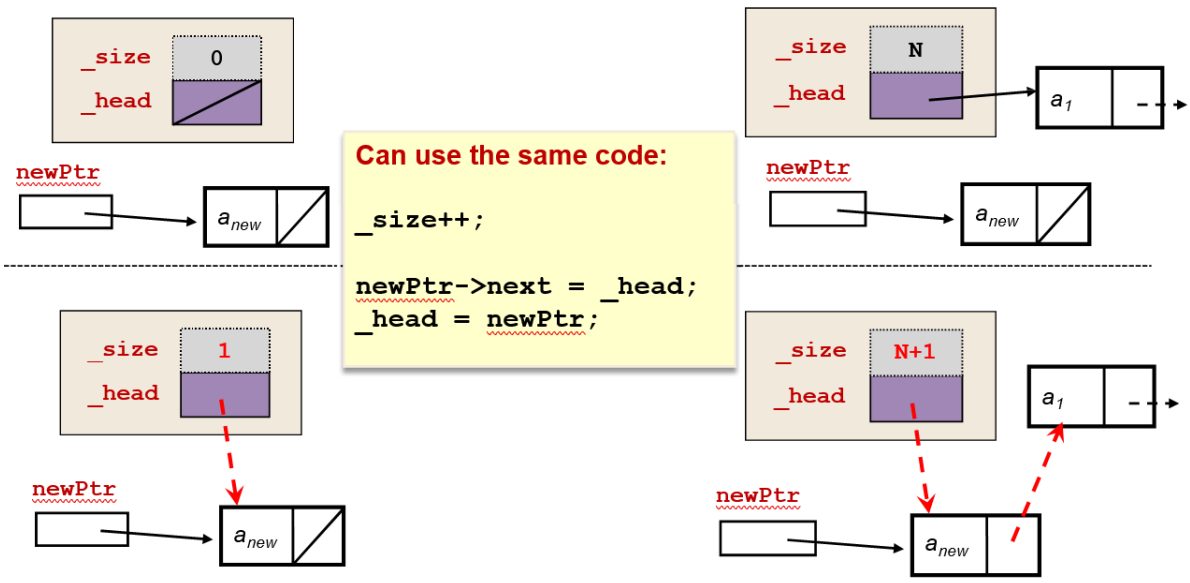


Fig. 4.12 Insert into head of linked list (possibly empty)

- Since we only keep the head pointer, list traversal is needed to reach other positions

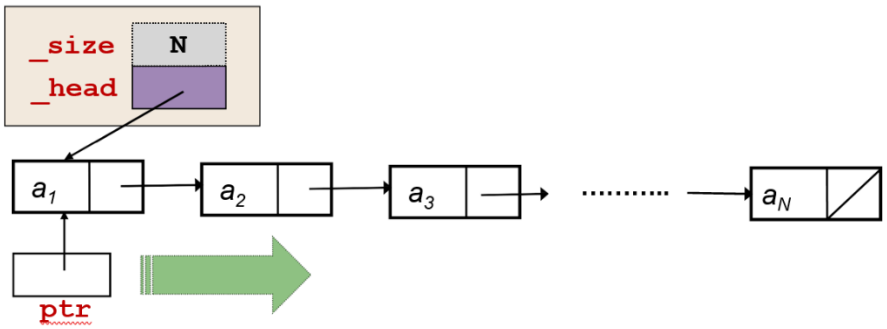


Fig. 4.13 List Traversal

Insertion: End of Linked List

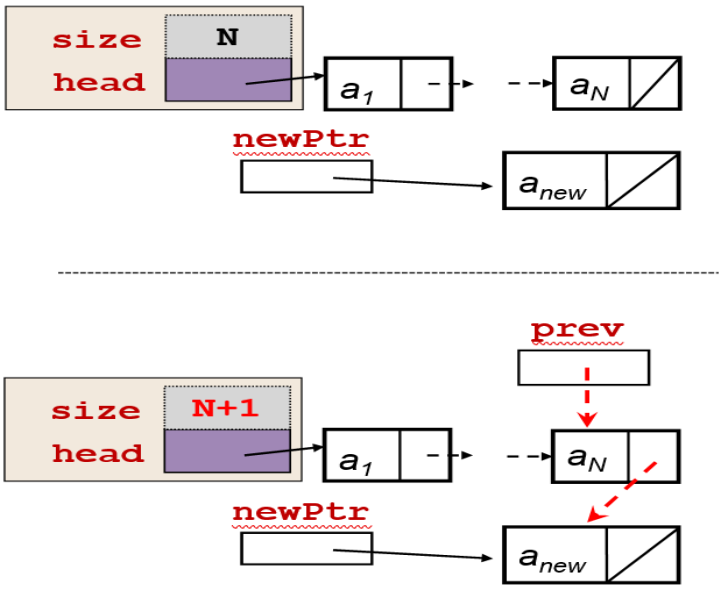


Fig. 4.14 Insertion: End of Linked List

Insertion: Kth Position of Linked List (Middle)

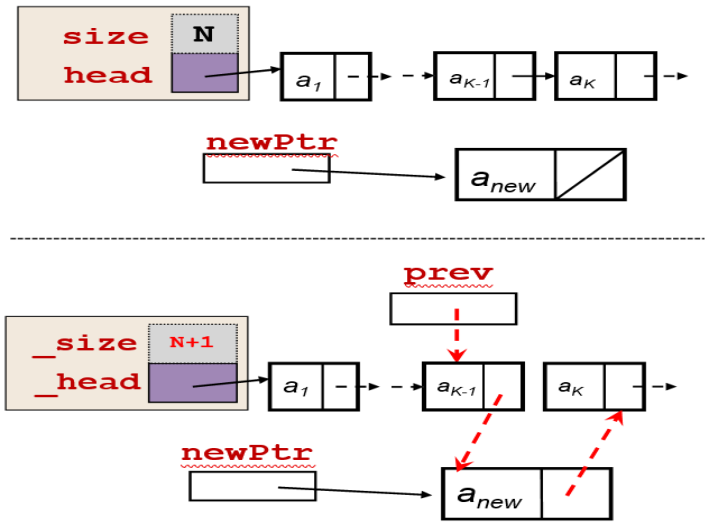


Fig. 4.15 Insertion: Kth of Linked List (Middle)

Linked List Deletion: General

- For Linked List deletion, the cases can be simplified similar to:
    - Deletion of head node (1st Node in list)
    - Deletion of other node (including middle or end of list)

Deletion: Head of Linked List

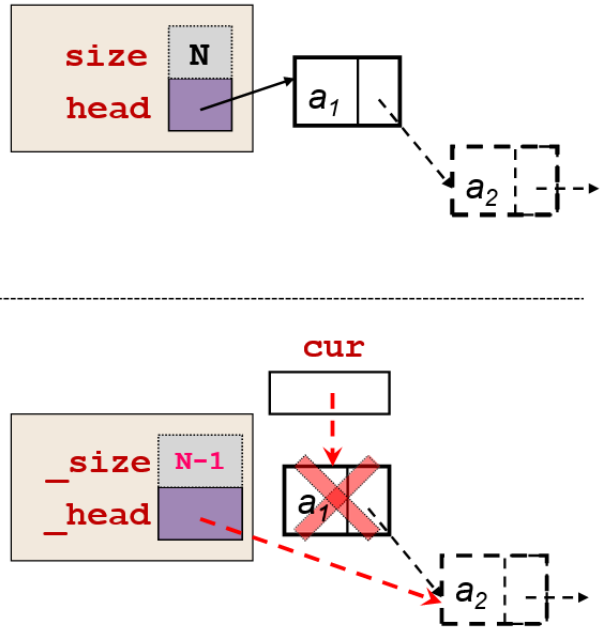


Fig. 4.16 Deletion: Head of Linked List

Deletion: Kth Position of Linked List (Middle)

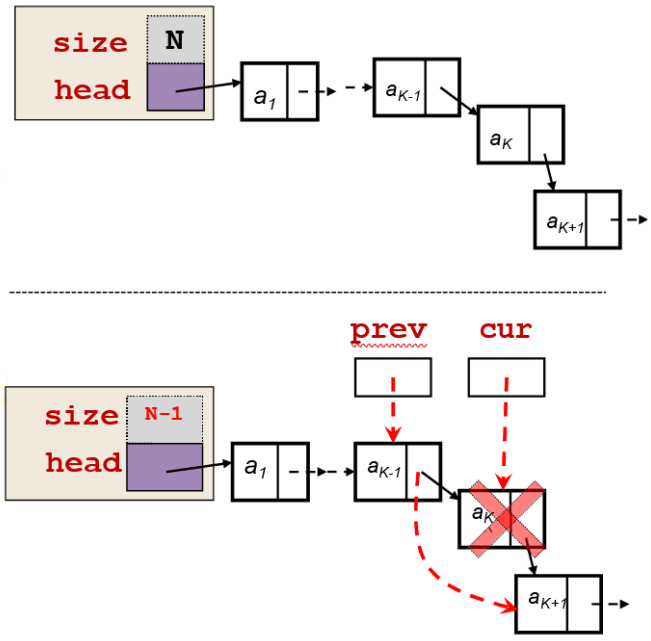


Fig. 4.17 Deletion: Kth Position of Linked List (Middle)

C Program to implement List ADT using Linked Lisgt .....

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
struct node {
   int data;
   int key;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;
//display the list
void printList() {
   struct node *ptr = head;
   printf("\n[ ");
   //start from the beginning
   while(ptr != NULL) {
      printf("(%d,%d) ",ptr->key,ptr->data);
      ptr = ptr->next;
   }
   printf(" ]");
}
//insert link at the first location
void insertFirst(int key, int data) {
   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));
   link->key = key;
   link->data = data;
   //point it to old first node
   link->next = head;
   //point first to new first node
```

```
   head = link;
}
//delete first item
struct node* deleteFirst() {
   //save reference to first link
  struct node *tempLink = head;
   //mark next to first link as first
   head = head->next;
   //return the deleted link
   return tempLink;
}
//is list empty
bool isEmpty() {
   return head == NULL;
}
int length() {
   int length = 0;
   struct node *current;
   for(current = head; current != NULL; current = current->next) {
      length++;
   }
   return length;
}
struct node* find(int key) { //find a link with given key
   struct node* current = head;    //start from the first link
      if(head == NULL) { //if list is empty
      return NULL;
   }
   //navigate through list
   while(current->key != key) {
      //if it is last node
      if(current->next == NULL) {
```

```
        return NULL;
      } else {
        //go to next link
        current = current->next;
      }
   }
   //if data found, return the current Link
   return current;
}
//delete a link with given key
struct node* delete(int key) {
   //start from the first link
   struct node* current = head;
   struct node* previous = NULL;
   //if list is empty
   if(head == NULL) {
      return NULL;
   }
   //navigate through list
   while(current->key != key) {
      //if it is last node
      if(current->next == NULL) {
        return NULL;
      } else {
        //store reference to current link
        previous = current;
        //move to next link
        current = current->next;
      }
   }
   //found a match, update the link
   if(current == head) {
```

```
      //change first to point to next link
      head = head->next;
   } else {
      //bypass the current link
      previous->next = current->next;
   }
   return current;
}
void sort() {
   int i, j, k, tempKey, tempData;
   struct node *current;
   struct node *next;
   int size = length();
   k = size ;
   for ( i = 0 ; i < size - 1 ; i++, k-- ) {
      current = head;
      next = head->next;
      for ( j = 1 ; j < k ; j++ ) {
         if ( current->data > next->data ) {
            tempData = current->data;
            current->data = next->data;
            next->data = tempData;
            tempKey = current->key;
            current->key = next->key;
            next->key = tempKey;
         }
         current = current->next;
         next = next->next;
      }
   }
}
void reverse(struct node** head_ref) {
```

```
   struct node* prev   = NULL;
   struct node* current = *head_ref;
   struct node* next;
   while (current != NULL) {
      next  = current->next;
      current->next = prev;
      prev = current;
      current = next;
   }
   *head_ref = prev;
}
void main() {
   insertFirst(1,10);
   insertFirst(2,20);
   insertFirst(3,30);
   insertFirst(4,1);
   insertFirst(5,40);
   insertFirst(6,56);
   printf("Original List: ");
   //print list
   printList();
   while(!isEmpty()) {
      struct node *temp = deleteFirst();
      printf("\nDeleted value:");
      printf("(%d,%d) ",temp->key,temp->data);
   }
   printf("\nList after deleting all items: ");
   printList();
   insertFirst(1,10);
   insertFirst(2,20);
   insertFirst(3,30);
   insertFirst(4,1);
```

```
insertFirst(5,40);
insertFirst(6,56);
printf("\nRestored List: ");
printList();
printf("\n");
struct node *foundLink = find(4);
if(foundLink != NULL) {
   printf("Element found: ");
   printf("(%d,%d) ",foundLink->key,foundLink->data);
   printf("\n");
} else {
   printf("Element not found.");
}
delete(4);
printf("List after deleting an item: ");
printList();
printf("\n");
foundLink = find(4);
if(foundLink != NULL) {
   printf("Element found: ");
   printf("(%d,%d) ",foundLink->key,foundLink->data);
   printf("\n");
} else {
   printf("Element not found.");
}
printf("\n");
sort();
printf("List after sorting the data: ");
printList();
reverse(&head);
printf("\nList after reversing the data: ");
printList(); }
```

STACK ADT

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages.
- It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

Fig. 4.18 Examples of Stack

- A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only.
- Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.
- This feature makes it LIFO data structure. LIFO stands for Last-in-first-out.
- Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.

Stack Representation

- The following diagram depicts a stack and its operations −

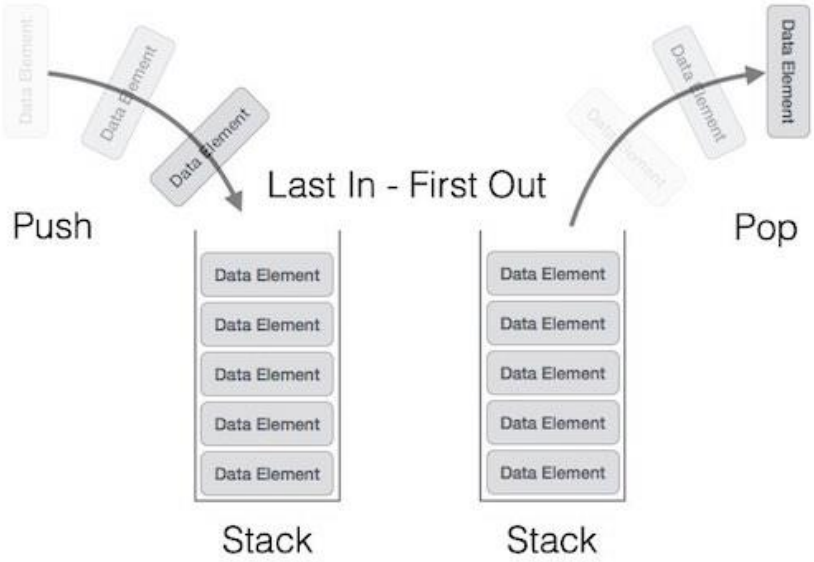


Fig. 4.19 Stack Operations

- A stack can be implemented by means of Array, Structure, Pointer, and Linked List.

- Stack can either be a fixed size one or it may have a sense of dynamic resizing.
- Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

- Stack operations may involve initializing the stack, using it and then de-initializing it.
- Apart from these basic stuffs, a stack is used for the following two primary operations
  - push() − Pushing (storing) an element on the stack.
  - pop() − Removing (accessing) an element from the stack.
- When data is PUSHed onto stack, To use a stack efficiently, we need to check the status of stack as well.
- For the same purpose, the following functionality is added to stacks −
  - peek() − get the top data element of the stack, without removing it.
  - isFull() − check if stack is full.
  - isEmpty() − check if stack is empty.
- At all times, we maintain a pointer to the last PUSHed data on the stack.
- As this pointer always represents the top of the stack, hence named top.
- The top pointer provides top value of the stack without actually removing it.
- First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek
  return stack[top]
end procedure
```

Implementation of peek() function in C programming language −

```
int peek() {
  return stack[top];
}
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull
  if top equals to MAXSIZE
    return true
```

```
    else
      return false
    endif
  end procedure
```

Implementation of isfull() function in C programming language −

```
  bool isfull() {
    if(top == MAXSIZE)
      return true;
    else
      return false;
  }
```

isempty()

Algorithm of isempty() function −

```
  begin procedure isempty
    if top less than 1
      return true
    else
      return false
    endif
  end procedure
```

- Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

```
  bool isempty() {
    if(top == -1)
      return true;
    else
      return false;
  }
```

Push Operation

- The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- Step 1 − Checks if the stack is full.
- Step 2 − If the stack is full, produces an error and exit.
- Step 3 − If the stack is not full, increments top to point next empty space.
- Step 4 − Adds data element to the stack location, where top is pointing.
- Step 5 − Returns success.

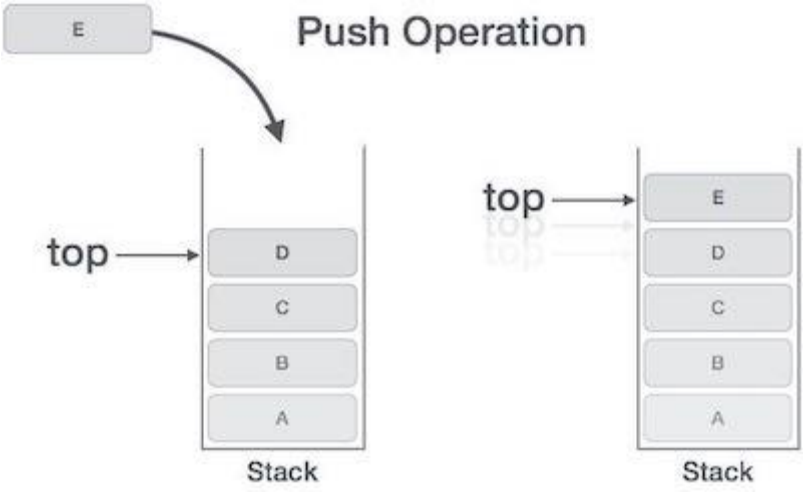


Fig. 4. 20 PUSH Operation in Stack

- If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data
  if stack is full
    return null
  endif
   top ← top + 1
  stack[top] ← data
end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −

```
void push(int data) {
  if(!isFull()) {
    top = top + 1;
    stack[top] = data;
  } else {
```

```
      printf("Could not insert data, Stack is full.\n");
    }
  }
```

Pop Operation

- Accessing the content while removing it from the stack, is known as a Pop Operation.
- In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value.
- But in linked-list implementation, pop() actually removes data element and deallocates memory space.
- A Pop operation may involve the following steps −
  - Step 1 − Checks if the stack is empty.
  - Step 2 − If the stack is empty, produces an error and exit.
  - Step 3 − If the stack is not empty, accesses the data element at which top is pointing.
  - Step 4 − Decreases the value of top by 1.
  - Step 5 − Returns success.

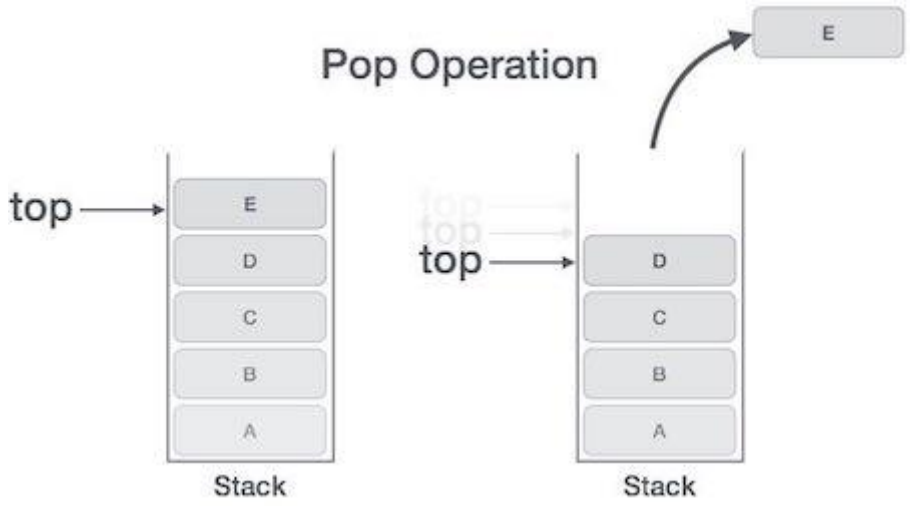


Fig. 4. 21 POP Operation in Stack

Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack
  if stack is empty
    return null
  endif
    data ← stack[top]
```

```
      top ← top - 1
      return data
    end procedure
```

Implementation of this algorithm in C, is as follows −

```
    int pop(int data) {
      if(!isempty()) {
        data = stack[top];
        top = top - 1;
        return data;
      } else {
        printf("Could not retrieve data, Stack is empty.\n");
      }
    }
```

C Program for Stack ADT implementation using Array ……..

```
#include<stdio.h>
#include<conio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
   clrscr();
   top=-1;
   printf("\n Enter the size of STACK[MAX=100]:");
   scanf("%d",&n);
   printf("\n\t STACK OPERATIONS USING ARRAY");
   printf("\n\t--------------------------------");
   printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
   do
   {
      printf("\n Enter the Choice:");
```

```
        scanf("%d",&choice);
        switch(choice)
        {
           case 1:
           {
              push();
              break;
           }
           case 2:
           {
              pop();
              break;
           }
           case 3:
           {
              display();
              break;
           }
           case 4:
           {
              printf("\n\t EXIT POINT ");
              break;
           }
           default:
           {
              printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
           }
           }
    } while(choice!=4);
    getch();
    return 0; }
void push()
```

```
{
   if(top>=n-1)
   {
      printf("\n\tSTACK is over flow");
   }
   else
   {
      printf(" Enter a value to be pushed:");
      scanf("%d",&x);
      top++;
      stack[top]=x;
   }
}
void pop()
{
   if(top<=-1)
   {
      printf("\n\t Stack is under flow");
   }
   else
   {
      printf("\n\t The popped elements is %d",stack[top]);
      top--;
   }
}
void display()
{
   if(top>=0)
   {
      printf("\n The elements in STACK \n");
      for(i=top; i>=0; i--)
         printf("\n%d",stack[i]);
```

```
        printf("\n Press Next Choice");
    }
    else
    {
        printf("\n The STACK is empty");
    }
 }
```

C Program for Stack ADT implementation using Linked List ……..

```
#include <stdio.h>
#include <stdlib.h>
 struct node
{
    int info;
    struct node *ptr;
}*top,*top1,*temp;
int topelement();
void push(int data);
void pop();
void empty();
void display();
void destroy();
void stack_count();
void create();
int count = 0;
void main()
{
    int no, ch, e;
    printf("\n 1 - Push");
    printf("\n 2 - Pop");
    printf("\n 3 - Top");
    printf("\n 4 - Empty");
    printf("\n 5 - Exit");
```

```
    printf("\n 6 - Dipslay");
    printf("\n 7 - Stack Count");
    printf("\n 8 - Destroy stack");
    create();
    while (1)
    {
        printf("\n Enter choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            printf("Enter data : ");
            scanf("%d", &no);
            push(no);
            break;
        case 2:
            pop();
            break;
        case 3:
            if (top == NULL)
                printf("No elements in stack");
            else
            {
                e = topelement();
                printf("\n Top element : %d", e);
            }
            break;
        case 4:
            empty();
            break;
        case 5:
            exit(0);
```

```
        case 6:
            display();
            break;
        case 7:
            stack_count();
            break;
        case 8:
            destroy();
            break;
        default :
            printf(" Wrong choice, Please enter correct choice  ");
            break;
        }
    }
}
 /* Create empty stack */
void create()
{
    top = NULL;
}
 /* Count stack elements */
void stack_count()
{
    printf("\n No. of elements in stack : %d", count);
}
 /* Push data into stack */
void push(int data)
{
    if (top == NULL)
    {
        top =(struct node *)malloc(1*sizeof(struct node));
        top->ptr = NULL;
```

```
        top->info = data;
    }
    else
    {
        temp =(struct node *)malloc(1*sizeof(struct node));
        temp->ptr = top;
        temp->info = data;
        top = temp;
    }
    count++;
}
 /* Display stack elements */
void display()
{
    top1 = top;

    if (top1 == NULL)
    {
        printf("Stack is empty");
        return;
    }
    while (top1 != NULL)
    {
        printf("%d ", top1->info);
        top1 = top1->ptr;
    }
 }
 /* Pop Operation on stack */
void pop()
{
    top1 = top;
     if (top1 == NULL)
```

```
    {
        printf("\n Error : Trying to pop from empty stack");
        return;
    }
    else
        top1 = top1->ptr;
    printf("\n Popped value : %d", top->info);
    free(top);
    top = top1;
    count--;
}
 /* Return top element */
int topelement()
{
    return(top->info);
}
/* Check if stack is empty or not */
void empty()
{
    if (top == NULL)
        printf("\n Stack is empty");
    else
        printf("\n Stack is not empty with %d elements", count);
}
 /* Destroy entire stack */
void destroy()
{
    top1 = top;
    while (top1 != NULL)
    {
        top1 = top->ptr;
        free(top);
```

```
        top = top1;
        top1 = top1->ptr;
    }
    free(top1);
    top = NULL;
    printf("\n All stack elements destroyed");
    count = 0;
}
```

QUEUE ADT

- Queue is an abstract data structure, somewhat similar to Stacks.
- Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).
- Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

Fig. 4.22 Queue Example

- A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

- As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −

Fig. 4.23 Representation of Queue

- As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

- Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory.
- Here we shall try to understand the basic operations associated with queues −
  - enqueue() − add (store) an item to the queue.
  - dequeue() − remove (access) an item from the queue.
- Few more functions are required to make the above-mentioned queue operation efficient. These are −
  - peek() − Gets the element at the front of the queue without removing it.
  - isfull() − Checks if the queue is full.
  - isempty() − Checks if the queue is empty.
- In queue, we always dequeue (or access) data, pointed by front pointer and while enqueuing (or storing) data in the queue we take help of rear pointer.

The following are the supportive functions of a queue −

peek()

- This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows −

```
begin procedure peek
  return queue[front]
end procedure
```

Implementation of peek() function in C programming language −

```
int peek()
{
  return queue[front];
}
```

isfull()

- As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full.
- In case we maintain the queue in a circular linked-list, the algorithm will differ.

Algorithm of isfull() function −

```
begin procedure isfull
   if rear equals to MAXSIZE
      return true
   else
      return false
   endif
end procedure
```

Implementation of isfull() function in C programming language −

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty
   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif
   end procedure
```

- If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty. Here's the C programming code −

```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Enqueue Operation

- Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.
- The following steps should be taken to enqueue (insert) data into a queue −
  - Step 1 − Check if the queue is full.
  - Step 2 − If the queue is full, produce overflow error and exit.
  - Step 3 − If the queue is not full, increment rear pointer to point the next empty space.
  - Step 4 − Add data element to the queue location, where the rear is pointing.
  - Step 5 − return success.

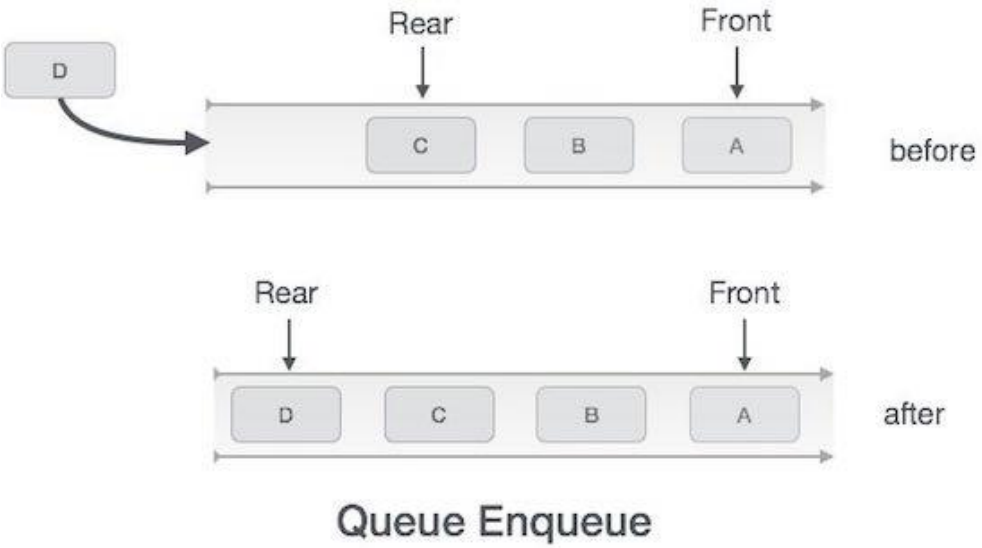


Fig. 4.24 Enqueue Operation

- Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)
    if queue is full
    return overflow
  endif
    rear ← rear + 1
  queue[rear] ← data
  return true
end procedure
```

Implementation of enqueue () in C programming language −

```
int enqueue (int data)
  if(isfull())
     return 0;
     rear = rear + 1;
  queue[rear] = data;
     return 1;
end procedure
```

Dequeue Operation

- Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access.
- The following steps are taken to perform dequeue operation −
  - Step 1 − Check if the queue is empty.
  - Step 2 − If the queue is empty, produce underflow error and exit.
  - Step 3 − If the queue is not empty, access the data where front is pointing.
  - Step 4 − Increment front pointer to point to the next available data element.
  - Step 5 − Return success.

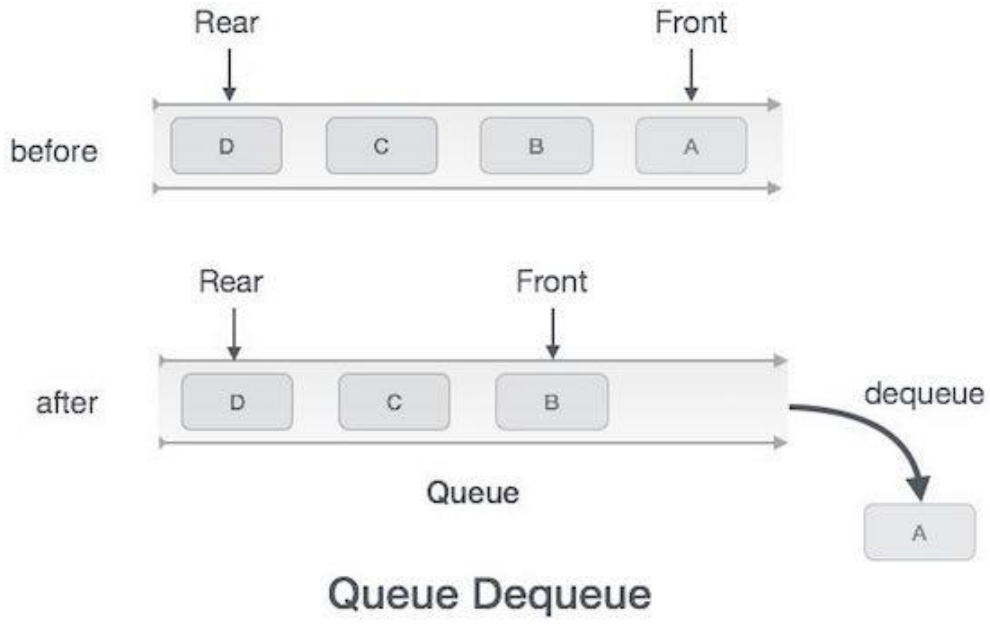


Fig. 4.25 Dequeue Operation

Algorithm for dequeue operation

```
procedure dequeue
     if queue is empty
```

```
    return underflow
  end if
  data = queue[front]
  front ← front + 1
  return true
end procedure
```

Implementation of dequeue () in C programming language −

```
int dequeue () {
  if(isempty())
    return 0;
  int data = queue[front];
  front = front + 1;
  return data;
}
```

C Program for the implementation of Queue ADT using Array ……

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
int peek() {
  return intArray[front];
}
int isEmpty() {
  return itemCount == 0;
}
int isFull() {
  return itemCount == MAX;
```

```
}
int size() {
   return itemCount;
}
void insert(int data) {
   if(!isFull()) {
      if(rear == MAX-1) {
         rear = -1;
      }
      intArray[++rear] = data;
      itemCount++;
   }
}
int removeData() {
   int data = intArray[front++];
   if(front == MAX) {
      front = 0;
   }
   itemCount--;
   return data;
}
int main() {
   int num;
   /* insert 5 items */
   insert(3);
   insert(5);
   insert(9);
   insert(1);
   insert(12);
   // front : 0
   // rear  : 4
   // ------------------
```

```
// index : 0 1 2 3 4
// ------------------
// queue : 3 5 9 1 12
insert(15);
// front : 0
// rear  : 5
// ----------------------
// index : 0 1 2 3 4  5
// ----------------------
// queue : 3 5 9 1 12 15
if(isFull()) {
   printf("Queue is full!\n");
}
// remove one item
num = removeData();
printf("Element removed: %d\n",num);
// front : 1
// rear  : 5
// -------------------
// index : 1 2 3 4  5
// -------------------
// queue : 5 9 1 12 15
// insert more items
insert(16);
// front : 1
// rear  : -1
// ----------------------
// index : 0  1 2 3 4  5
// ----------------------
// queue :  5 9 1 12 15 16
// As queue is full, elements will not be inserted.
insert(17);
```

```
  insert(18);
  // ----------------------
  // index : 0  1 2 3 4  5
  // ----------------------
  // queue : 5 9 1 12 15 16
  printf("Element at front: %d\n",peek());
  printf("----------------------\n");
  printf("index : 5 4 3 2  1  0\n");
  printf("----------------------\n");
  printf("Queue:  ");
  while(!isEmpty()) {
     int n = removeData();
     printf("%d ",n);
}
getch();
return(0);
}
```

C Program for the implementation of Queue ADT using Linked List ……

```
#include <stdio.h>
#include <stdlib.h>
 struct node
{
   int info;
   struct node *ptr;
}*front,*rear,*temp,*front1;
 int frontelement();
void enq(int data);
void deq();
void empty();
void display();
void create();
void queuesize();
```

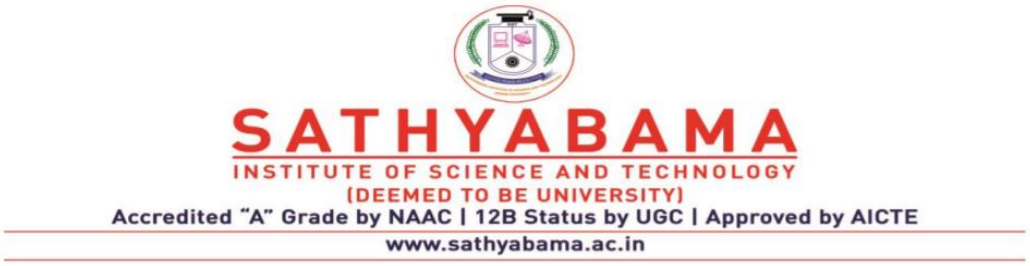
```
 int count = 0;
 void main()
{
    int no, ch, e;
     printf("\n 1 - Enque");
    printf("\n 2 - Deque");
    printf("\n 3 - Front element");
    printf("\n 4 - Empty");
    printf("\n 5 - Exit");
    printf("\n 6 - Display");
    printf("\n 7 - Queue size");
    create();
    while (1)
    {
        printf("\n Enter choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            printf("Enter data : ");
            scanf("%d", &no);
            enq(no);
            break;
        case 2:
            deq();
            break;
        case 3:
            e = frontelement();
            if (e != 0)
                printf("Front element : %d", e);
            else
                printf("\n No front element in Queue as queue is empty");
```

```
            break;
        case 4:
            empty();
            break;
        case 5:
            exit(0);
        case 6:
            display();
            break;
        case 7:
            queuesize();
            break;
        default:
            printf("Wrong choice, Please enter correct choice  ");
            break;
        }
    }
}
 /* Create an empty queue */
void create()
{
    front = rear = NULL;
}
 /* Returns queue size */
void queuesize()
{
    printf("\n Queue size : %d", count);
}
 /* Enqueing the queue */
void enq(int data)
{
    if (rear == NULL)
```

```
    {
        rear = (struct node *)malloc(1*sizeof(struct node));
        rear->ptr = NULL;
        rear->info = data;
        front = rear;
    }
    else
    {
        temp=(struct node *)malloc(1*sizeof(struct node));
        rear->ptr = temp;
        temp->info = data;
        temp->ptr = NULL;
         rear = temp;
    }
    count++;
}
 /* Displaying the queue elements */
void display()
{
    front1 = front;
     if ((front1 == NULL) && (rear == NULL))
    {
        printf("Queue is empty");
        return;
    }
    while (front1 != rear)
    {
        printf("%d ", front1->info);
        front1 = front1->ptr;
    }
    if (front1 == rear)
        printf("%d", front1->info);
```

```
}
 /* Dequeing the queue */
void deq()
{
   front1 = front;
    if (front1 == NULL)
   {
      printf("\n Error: Trying to display elements from empty queue");
      return;
   }
   else
      if (front1->ptr != NULL)
      {
         front1 = front1->ptr;
         printf("\n Dequed value : %d", front->info);
         free(front);
         front = front1;
      }
      else
      {
         printf("\n Dequed value : %d", front->info);
         free(front);
         front = NULL;
         rear = NULL;
      }
      count--;
}
 /* Returns the front element of queue */
int frontelement()
{
   if ((front != NULL) && (rear != NULL))
      return(front->info);
```

```
    else
        return 0;
}
 /* Display if queue is empty or not */
void empty()
{
     if ((front == NULL) && (rear == NULL))
        printf("\n Queue empty");
    else
       printf("Queue not empty");
}
```



SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND INSTRUMENTATION ENGINEERING

UNIT – V – SCSA1206 – C AND DATA STRUCTURES

UNIT V

NON-LINEAR DATA STRUCTURES

LINEAR DATA STRUCTURE:

- Data structure where data elements are arranged sequentially or linearly where the elements are attached to its previous and next adjacent in what is called a linear data structure.
- In linear data structure, single level is involved. Therefore, we can traverse all the elements in single run only.
- Linear data structures are easy to implement because computer memory is arranged in a linear way. Its examples are array, stack, queue, linked list, etc.

NON-LINEAR DATA STRUCTURE:

- Data structures where data elements are not arranged sequentially or linearly are called non-linear data structures.
- In a non-linear data structure, single level is not involved. Therefore, we can't traverse all the elements in single run only.
- Non-linear data structures are not easy to implement in comparison to linear data structure.
- It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are trees and graphs.

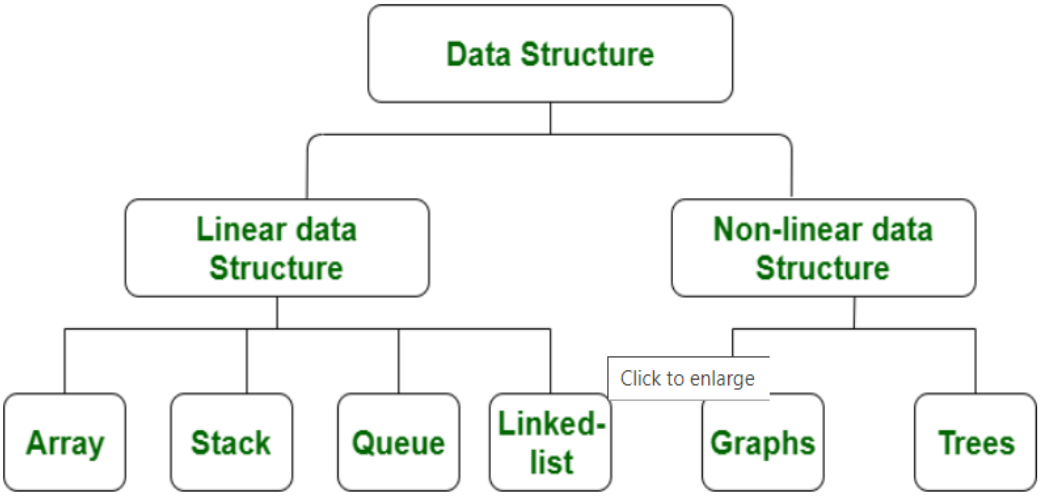


Fig. 5.1 Categorizes of Data Structures

Difference between Linear and Non-linear Data Structures:

Table 5.1 Difference between Linear and Non-Linear Data Structures

Sr. No.	Key	Linear Data Structures	Non-linear Data Structures
1	Data Element Arrangement	In linear data structure, data elements are sequentially connected and each element is traversable through a single run.	In non-linear data structure, data elements are hierarchically connected and are present at various levels.
2	Levels	In linear data structure, all data elements are present at a single level.	In non-linear data structure, data elements are present at multiple levels.
3	Implementation complexity	Linear data structures are easier to implement.	Non-linear data structures are difficult to understand and implement as compared to linear data structures.
4	Traversal	Linear data structures can be traversed completely in a single run.	Non-linear data structures are not easy to traverse and needs multiple runs to be traversed completely.
5	Memory utilization	Linear data structures are not very memory friendly and are not utilizing memory efficiently.	Non-linear data structures uses memory very efficiently.

Sr. No.	Key	Linear Data Structures	Non-linear Data Structures
6	Time Complexity	Time complexity of linear data structure often increases with increase in size.	Time complexity of non-linear data structure often remain with increase in size.
7	Examples	Array, List, Queue, Stack.	Graph, Map, Tree.

TREES

- A tree is a non-linear data structure that is used to represents hierarchical relationships between individual data items.

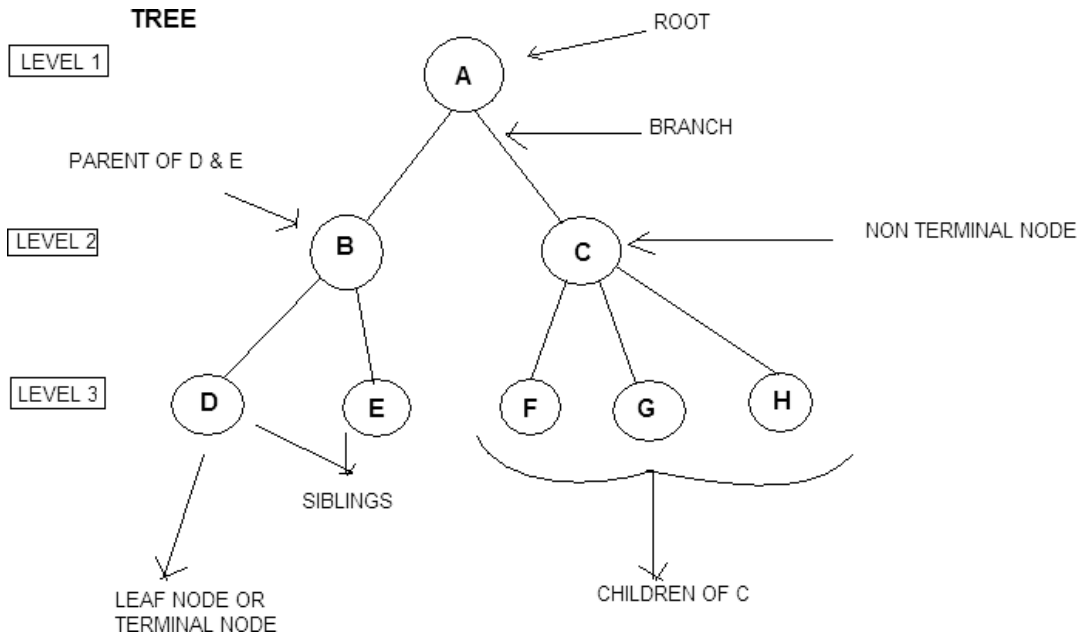


Fig. 5.2 Structure of Tree ADT

- **Tree:** A tree is a finite set of one or more nodes such that, there is a specially designated node called root. The remaining nodes are partitioned into n>=0 disjoint sets T1, T2, …..Tn, where each of these set is a tree T1, ……Tn are called the subtrees of the root.
- **Branch:** Branch is the link between the parent and its child.
- **Leaf:** A node with no children is called a leaf.
- **Subtree:** A Subtree is a subset of a tree that is itself a tree.

- **Degree:** The number of subtrees of a node is called the degree of the node. Hence nodes that have degree zero are called leaf or terminal nodes. The other nodes are referred as non-terminal nodes.
- **Children:** The nodes branching from a particular node X are called children of X and X is called its parent.
- **Siblings:** Children of the same parent are said to be siblings.
- **Degree of tree:** Degree of the tree is the maximum of the degree of the nodes in the tree.
- **Ancestors:** Ancestors of a node are all the nodes along the path from root to that node. Hence root is ancestor of all the nodes in the tree.
- **Level:** Level of a node is defined by letting root at level one. If a node is at level L, then its children are at level L + 1.
- **Height or depth:** The height or depth of a tree is defined to be the maximum level of any node in the tree.
- **Climbing:** The process of traversing the tree from the leaf to the root is called climbing the tree.
- **Descending:** The process of traversing the tree from the root to the leaf is called descending the tree.

TREE TRAVERSAL

- Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node.
- We cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −
    - In-order Traversal
    - Pre-order Traversal
    - Post-order Traversal
- Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

- In this traversal method, the left subtree is visited first, then the root and later the right

sub-tree. We should always remember that every node may represent a subtree itself.

- If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.

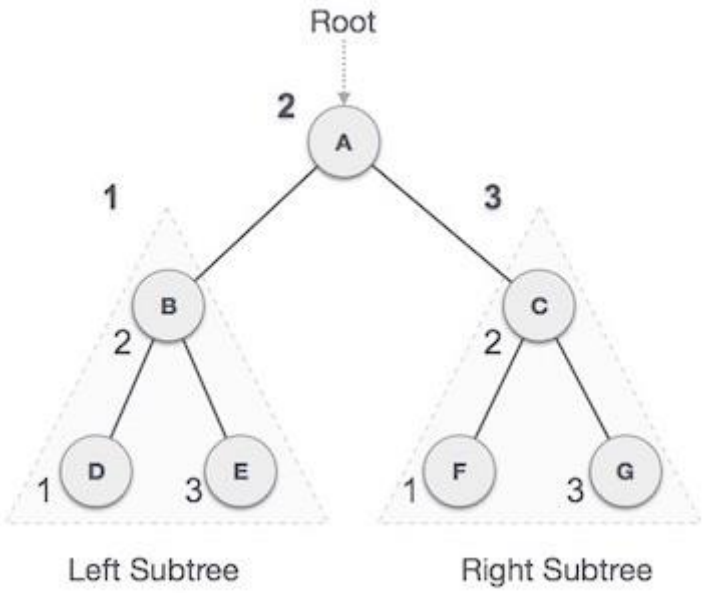


Fig. 5.3 In-order Traversal

- We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited.
- The output of inorder traversal of this tree will be −

$$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$$

Algorithm

Until all nodes are traversed −

- Step 1 − Recursively traverse left subtree.
- Step 2 − Visit root node.
- Step 3 − Recursively traverse right subtree.

Pre-order Traversal

- In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

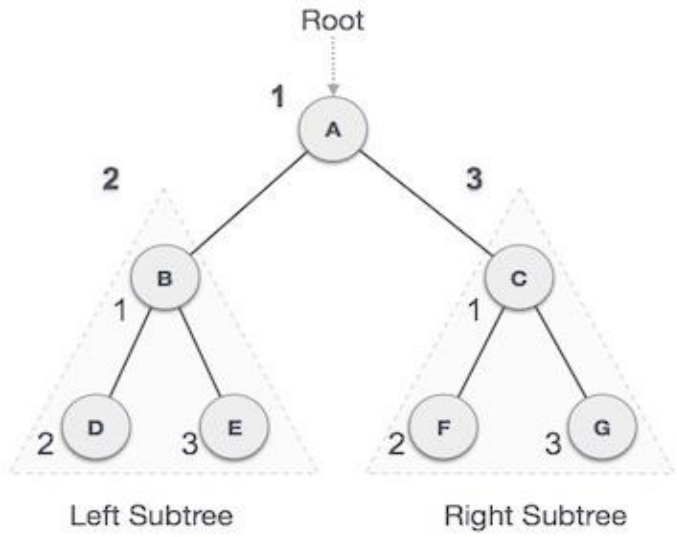


Fig. 5.4 Pre-order Traversal

- We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited.
- The output of pre-order traversal of this tree will be −

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$$

Algorithm

Until all nodes are traversed −

- Step 1 − Visit root node.
- Step 2 − Recursively traverse left subtree.
- Step 3 − Recursively traverse right subtree.

Post-order Traversal

- In this traversal method, the root node is visited last, hence the name. First, we traverse the left subtree, then the right subtree and finally the root node.
- We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited.
- The output of post-order traversal of this tree will be −

$$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$$

Algorithm

Until all nodes are traversed −

- Step 1 − Recursively traverse left subtree.
- Step 2 − Recursively traverse right subtree.

- Step 3 − Visit root node.

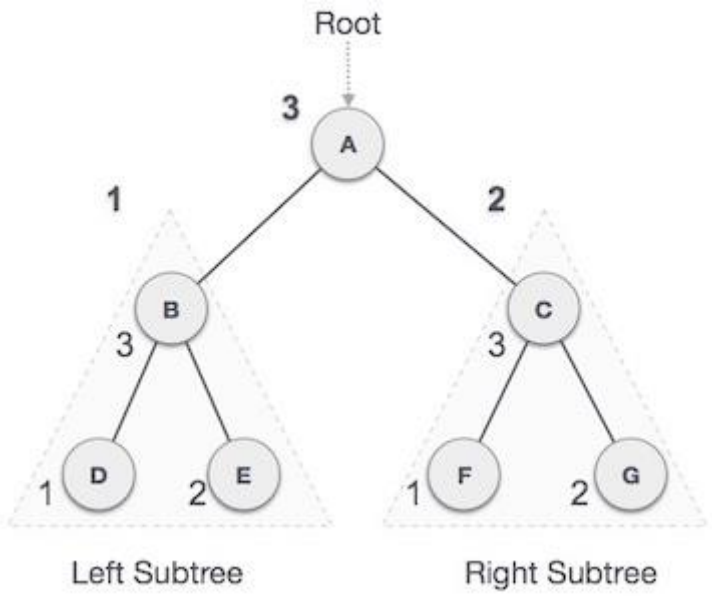


Fig. 5.5 Post-order Traversal

BINARY TREE

- Binary tree has nodes each of which has no more than two child nodes.
- Binary tree: A binary tree is a finite set of nodes that either is empty or consists of a root and two disjoint binary trees called the left subtree and right subtree.
- Left child: The node present to the left of the parent node is called the left child.
- Right child: The node present to the right of the parent node is called the right child.

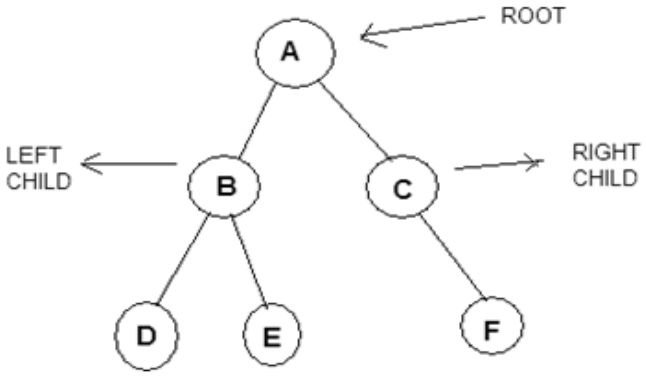


Fig. 5.6 Binary Tree

- **Skewed Binary tree:** If the new nodes in the tree are added only to one side of the binary tree then it is a skewed binary tree.

Fig. 5.7 Left and Right Skewed Binary Tree

- **Strictly binary tree:** If the binary tree has each node consisting of either two nodes or no nodes at all, then it is called a strictly binary tree.

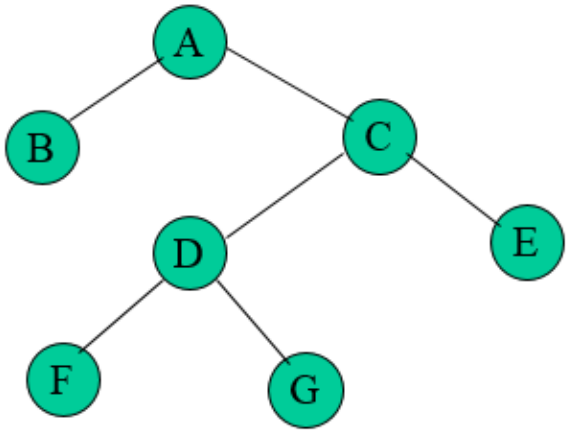


Fig. 5.8 Strictly Binary Tree

- **Complete binary tree:** If all the nodes of a binary tree consist of two nodes each and the nodes at the last level does not consist any nodes, then that type of binary tree is called a complete binary tree.

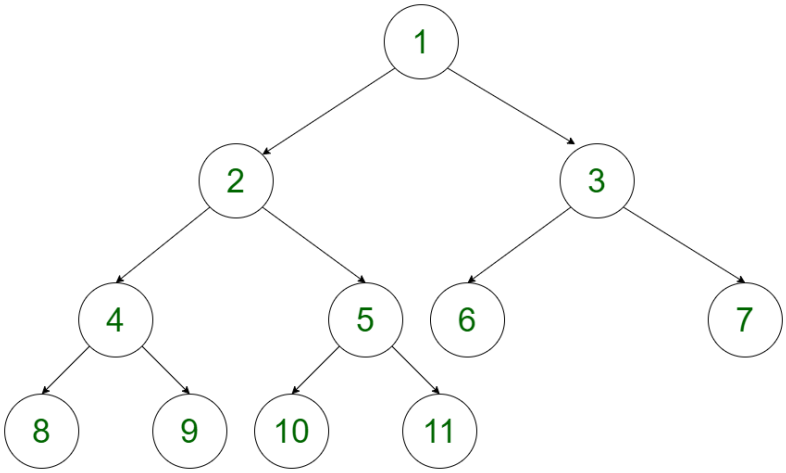


Fig. 5.9 Complete Binary Tree

- It can be observed that the maximum number of nodes on level i of a binary tree is 2i - 1, where i >= 1. The maximum number of nodes in a binary tree of depth k is 2k – 1, where k >= 1.

Representation of binary trees

- There are two ways in which a binary tree can be represented. They are:
    - Array representation of binary trees.
    - Linked representation of binary trees.

Array representation of binary trees

- When arrays are used to represent the binary trees, then an array of size 2k is declared where, k is the depth of the tree. For example, if the depth of the binary tree is 3, then maximum 23 - 1 = 7 elements will be present in the node and hence the array size will be 8.
- This is because the elements are stored from position one leaving the position 0 vacant. But generally, an array of bigger size is declared so that later new nodes can be added to the existing tree.
- The root element is always stored in position 1. The left child of node i is stored in position 2i and right child of node is stored in position 2i + 1. Hence the following formulae can be used to identify the parent, left child and right child of a particular node.
- Parent (i) = i / 2, if i 1 1. If i = 1 then i is the root node and root does not have parent.
- Left child (i) = 2i, if 2i 2 n, where n is the maximum number of elements in the tree. If 2i > n, then i has no left child.
- Right child (i) = 2i + 1, if 2i + 1 2 n. If 2i + 1 > n, then i has no right child.
- The following binary tree can be represented using arrays as shown.

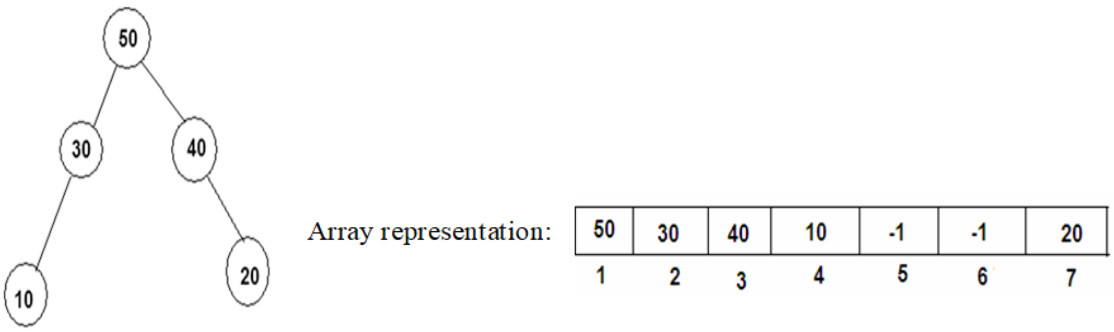


Fig. 5.10 Array Representation of Binary Tree

- The empty positions in the tree where no node is connected are represented in the array using -1, indicating absence of a node.
- Using the formula, we can see that for a node 3, the parent is 3/ 2 →1. Referring to the array locations, we find that 50 is the parent of 40. The left child of node 3 is 2*3 → 6. But the position 6 consists of -1 indicating that the left child does not exist for the node 3. Hence 50 does not have a left child. The right child of node 3 is 2*3 + 1 → 7. The position 7 in the array consists of 20. Hence, 20 is the right child of 40.

Linked representation of binary trees

- In linked representation of binary trees, instead of arrays, pointers are used to connect the various nodes of the tree.
- Hence each node of the binary tree consists of three parts namely, the info, left and right. The info part stores the data, left part stores the address of the left child and the right part stores the address of the right child.
- Logically the binary tree in linked form can be represented as shown.

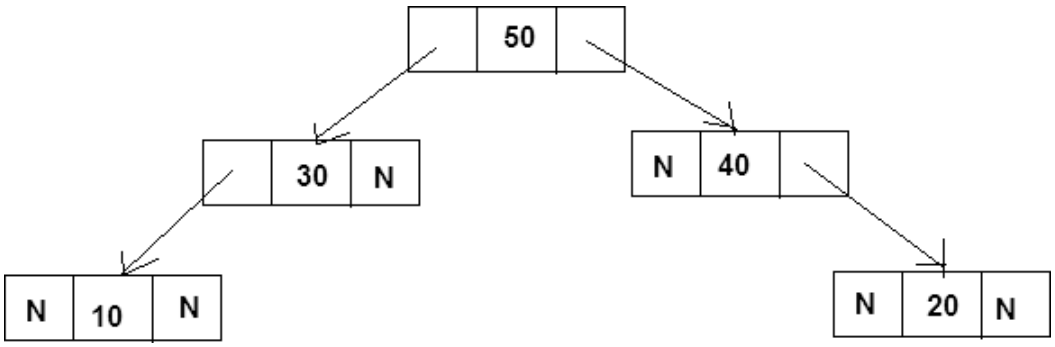


Fig. 5.11 Linked List Representation of Binary Tree

- The pointers storing NULL value indicates that there is no node attached to it. Traversing through this type of representation is very easy.
- The left child of a particular node can be accessed by following the left link of that node and the right child of a particular node can be accessed by following the right link of that node.

BINARY TREE TRAVERSALS

- There are three standard ways of traversing a binary tree T with root R. They are:
    - Preorder Traversal
    - Inorder Traversal

- Postorder Traversal

Preorder Traversal:

1. Process the root R.
2. Traverse the left subtree of R in preorder.
3. Traverse the right subtree of R in preorder.

Inorder Traversal:

1. Traverse the left subtree of R in inorder.
2. Process the root R.
3. Traverse the right subtree of R in inorder.

Postorder Traversal:

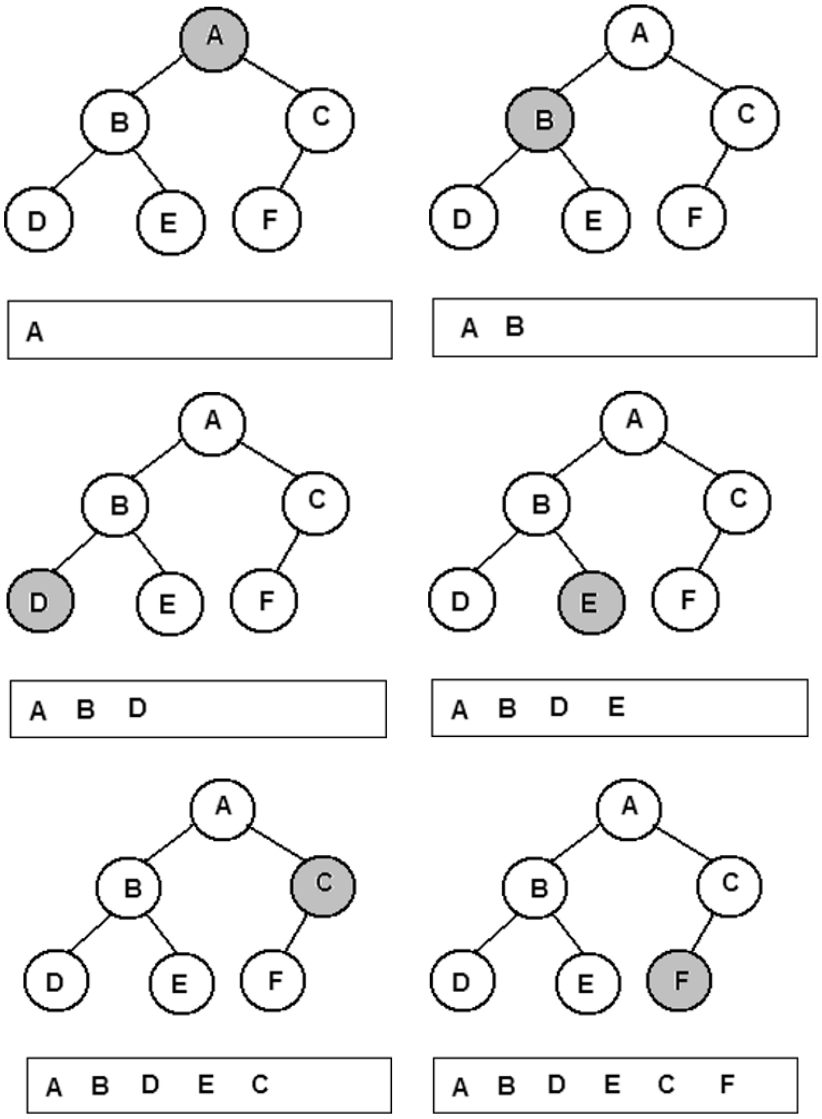
1. Traverse the left subtree of R in postorder.
2. Traverse the right subtree of R in postorder.
3. Process the root R.

- Observe that each algorithm contains the same three steps, and that the left subtree of R is always traversed before the right subtree.
- The difference between the algorithms is the time at which the root R is processed.
- The three algorithms are sometimes called, respectively, the node-left-right (NLR) traversal, the left-node-right (LNR) traversal and the left-right-node (LRN) traversal.

Traversal algorithms using recursive approach

Preorder Traversal

- In the preorder traversal the node element is visited first and then the right subtree of the node and then the right subtree of the node is visited.
- Consider the following case where we have 6 nodes in the tree A, B, C, D, E, F. The traversal always starts from the root of the tree. The node A is the root and hence it is visited first. The value at this node is processed. The processing can be doing some computation over it or just printing its value. Now we check if there exists any left child for this node if so, apply the preorder procedure on the left subtree. Now check if there is any right subtree for the node A, the preorder procedure is applied on the right subtree.
- Since there exists a left subtree for node A, B is now considered as the root of the left subtree of A and preorder procedure is applied. Hence, we find that B is processed next and then it is checked if B has a left subtree.
- This recursive method is continued until all the nodes are visited.



- The algorithm for the above method is presented in the pseudo-code given below:

```
PREORDER (ROOT)
Temp =ROOT
If temp = NULL
Return
End if
Print info(temp)
If left(temp) != NULL
PREORDER (left(temp))
End if
If right(temp) != NULL
```

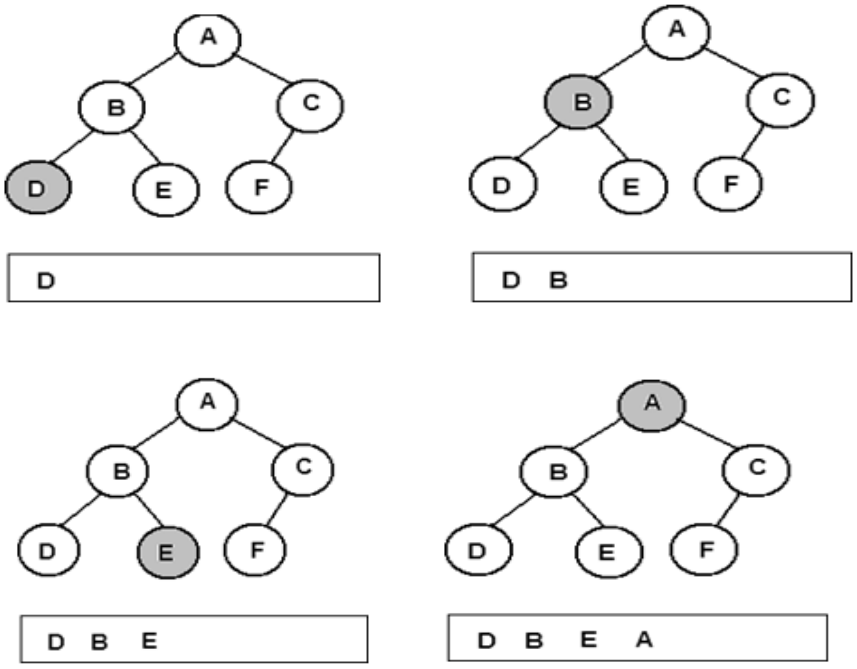
PREORDER (right(temp))

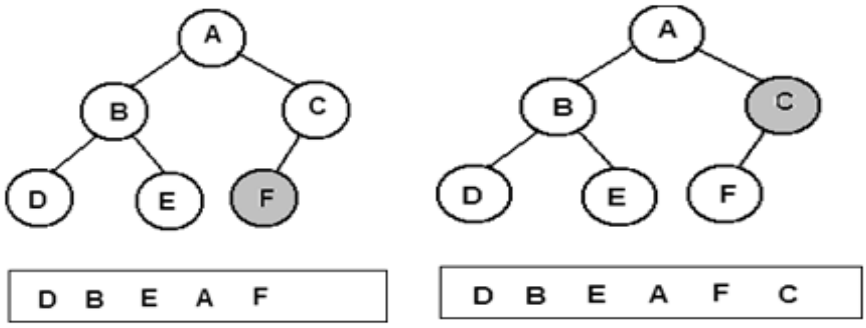
End if

End PREORDER

Inorder Traversal

- In the Inorder traversal method, the left subtree of the current node is visited first and then the current node is processed and at last the right subtree of the current node is visited.
- In the following example, the traversal starts with the root of the binary tree.
- The node A is the root and it is checked if it has the left subtree. Then the inorder traversal procedure is applied on the left subtree of the node A.
- Now we find that node D does not have left subtree. Hence the node D is processed and then it is checked if there is a right subtree for node D.
- Since there is no right subtree, the control returns back to the previous function which was applied on B. Since left of B is already visited, now B is processed. It is checked if B has the right subtree. If so, apply the inorder traversal method on the right subtree of the node B.
- This recursive procedure is followed till all the nodes are visited.

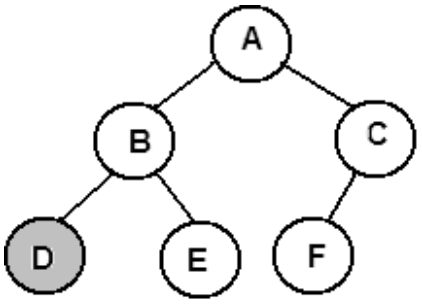




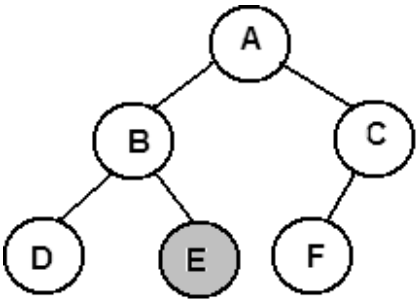

```
INORDER (ROOT)
Temp = ROOT
If temp = NULL
Return
End if
If left(temp) != NULL
INORDER (left(temp))
End if
Print info(temp)
If right(temp) != NULL
INORDER (right(temp))
End if
End INORDER
```

Postorder Traversal

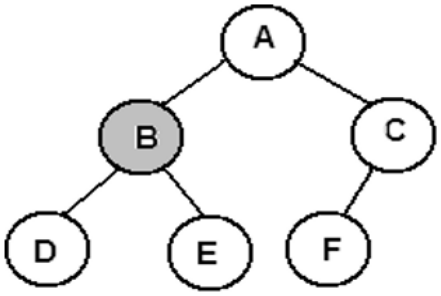
- In the postorder traversal method the left subtree is visited first, then the right subtree and at last the current node is processed.
- In the following example, A is the root node. Since A has the left subtree the postorder traversal method is applied recursively on the left subtree of A.
- Then when left subtree of A is completely is processed, the postorder traversal method is recursively applied on the right subtree of the node A.
- If right subtree is completely processed, then the current node A is processed.



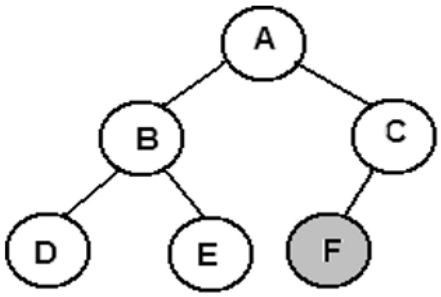
D



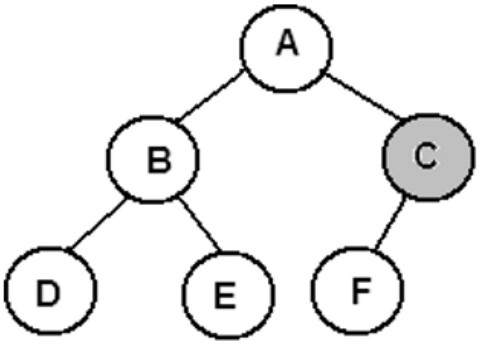
D E



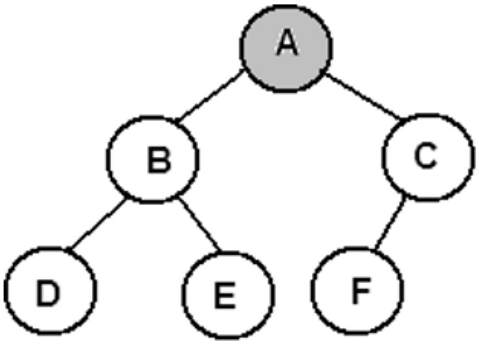
D E B



D E B F



D E B F C



D E B F C A

POSTORDER (ROOT)

Temp = ROOT

If temp = NULL

Return

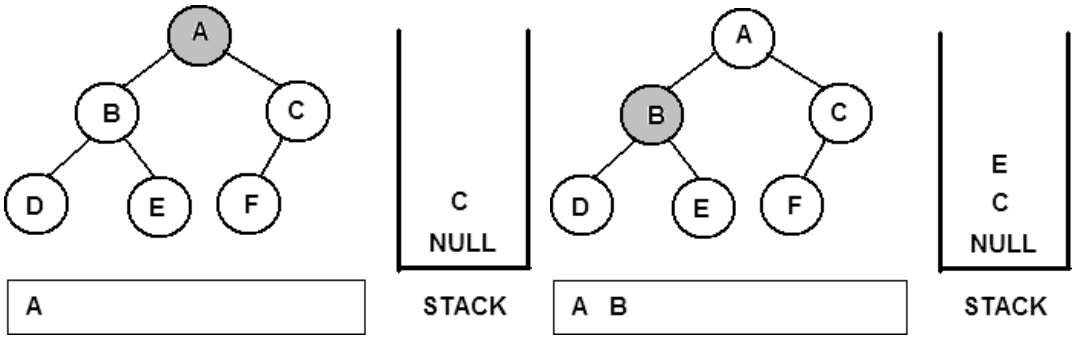
End if

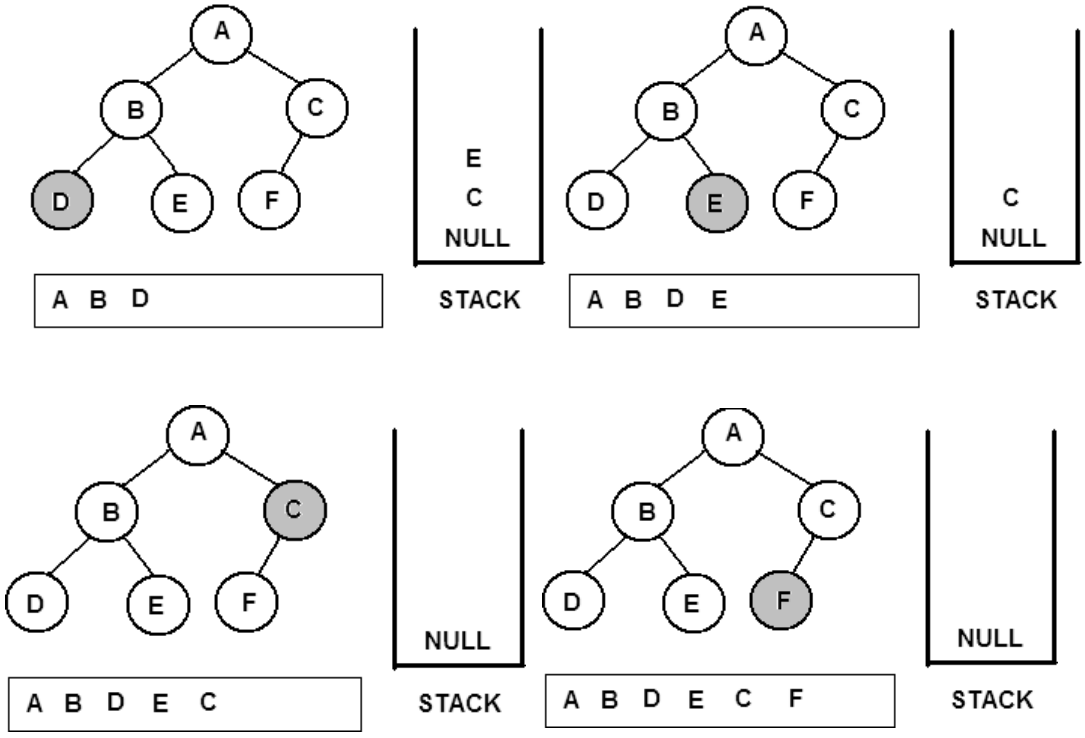
```
If left(temp) != NULL
POSTORDER (left(temp))
End if
If right(temp) != NULL
POSTORDER (right(temp))
End if
Print info(temp)
End POSTORDER
```

Binary Tree Traversal Using Iterative Approach

Preorder Traversal

- In the iterative method a stack is used to implement the traversal methods. Initially the stack is stored with a NULL value.
- The root node is taken for processing first. A pointer temp is made to point to this root node.
- If there exists a right node for the current node, then push that node into the stack. If there exists a left subtree for the current node then temp is made to the left child of the current node.
- If the left child does not exist, then a value is popped from the stack and temp is made to point to that node which is popped and the same process is repeated. This is done till the NULL value is popped from the stack.





PREORDER (ROOT)

Temp = ROOT, push (NULL) While temp 1 NULL

Print info(temp)

If right(temp) != NULL

Push(right(temp))

End if

If left(temp) != NULL

Temp = left(temp)

Else

Temp = pop ()

End if

End while

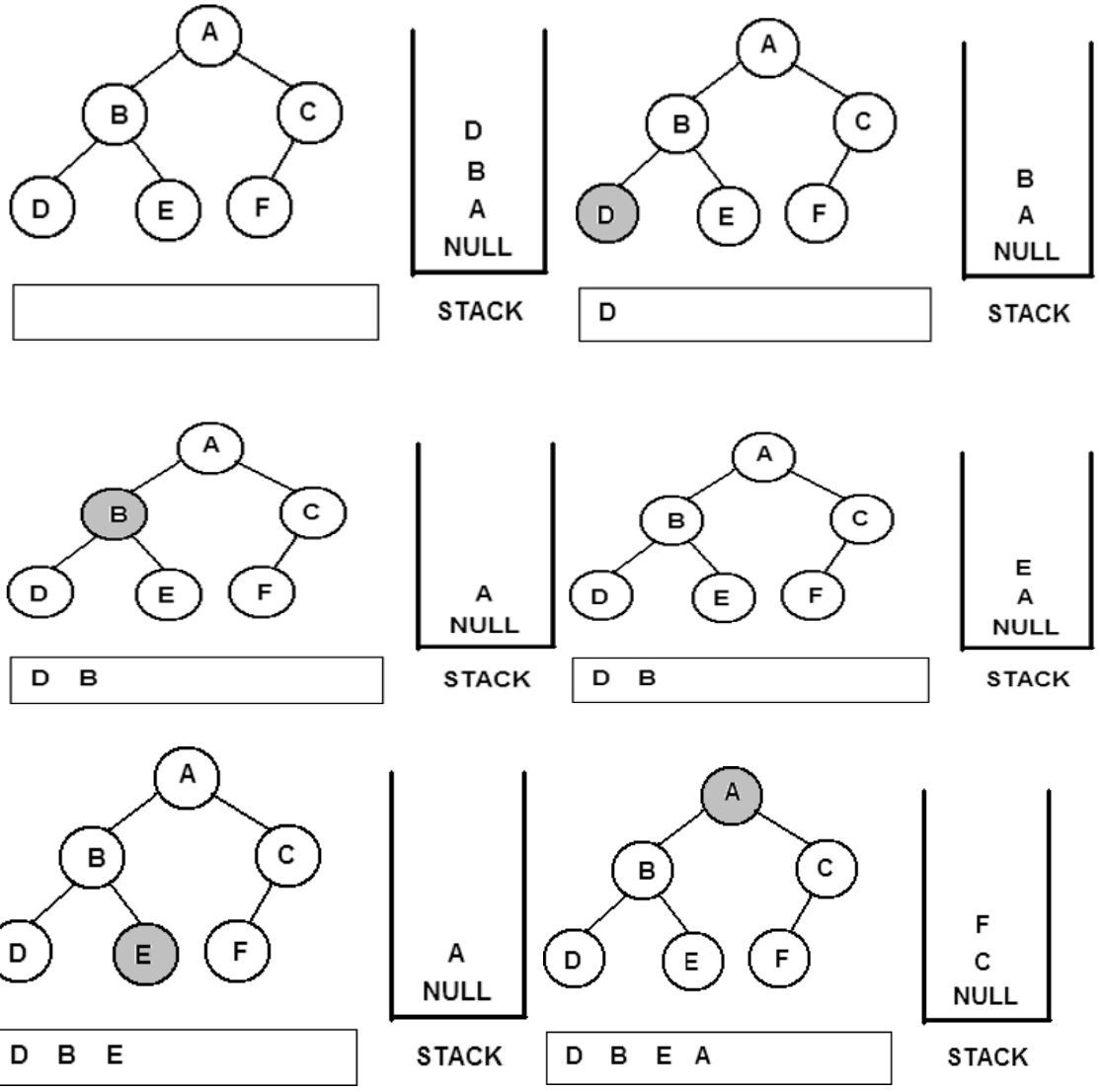
End PREORDER

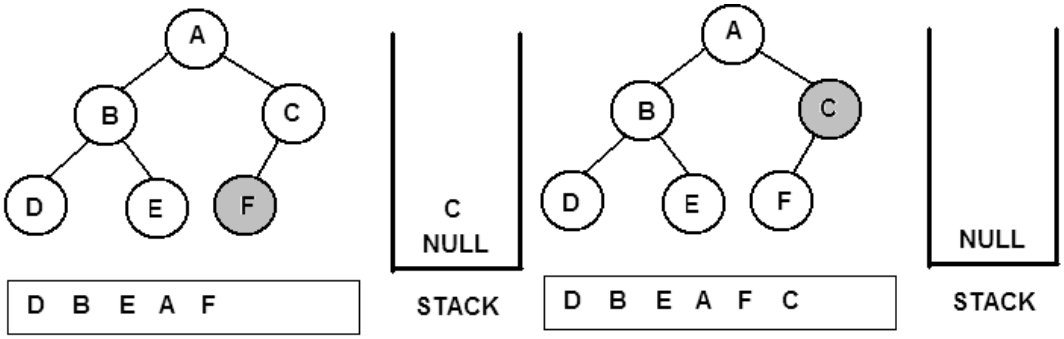
Inorder Traversal

- In the Inorder traversal method, the traversal starts at the root node. A pointer Temp is made to point to root node.
- Initially, the stack is stored with a NULL value and a flag RIGHTEXISTS is made equal to 1. Now for the current node, if the flag RIGHTEXISTS = 1, then immediately

it is made 0, and the node pointed by temp is pushed to the stack.

- The temp pointer is moved to the left child of the node if the left child exists. Every time the temp is moved to a new node, the node is pushed into the stack and temp is moved to its left child. This is continued till temp reaches a NULL value.
- After this one by one, the nodes in the stack are popped and are pointed by temp. The node is processed and if the node has right child, then the flag RIGHTEXISTS is set to 1 and the process describe above starts from the beginning. Thus, the process stops when the NULL value from the stack is popped.





INORDER (ROOT)

Temp = ROOT, push (NULL), RIGHTEXISTS = 1

While RIGHTEXISTS = 1

RIGHTEXISTS = 0

While temp != NULL

Push(temp)

Temp = left(temp)

End while

While (TRUE)

Temp = pop ()

If temp = NULL

Break

End if

Print info(temp)

If right(temp) != NULL

Temp = right(temp)

RIGHTEXISTS = 1

Break

End if

End while

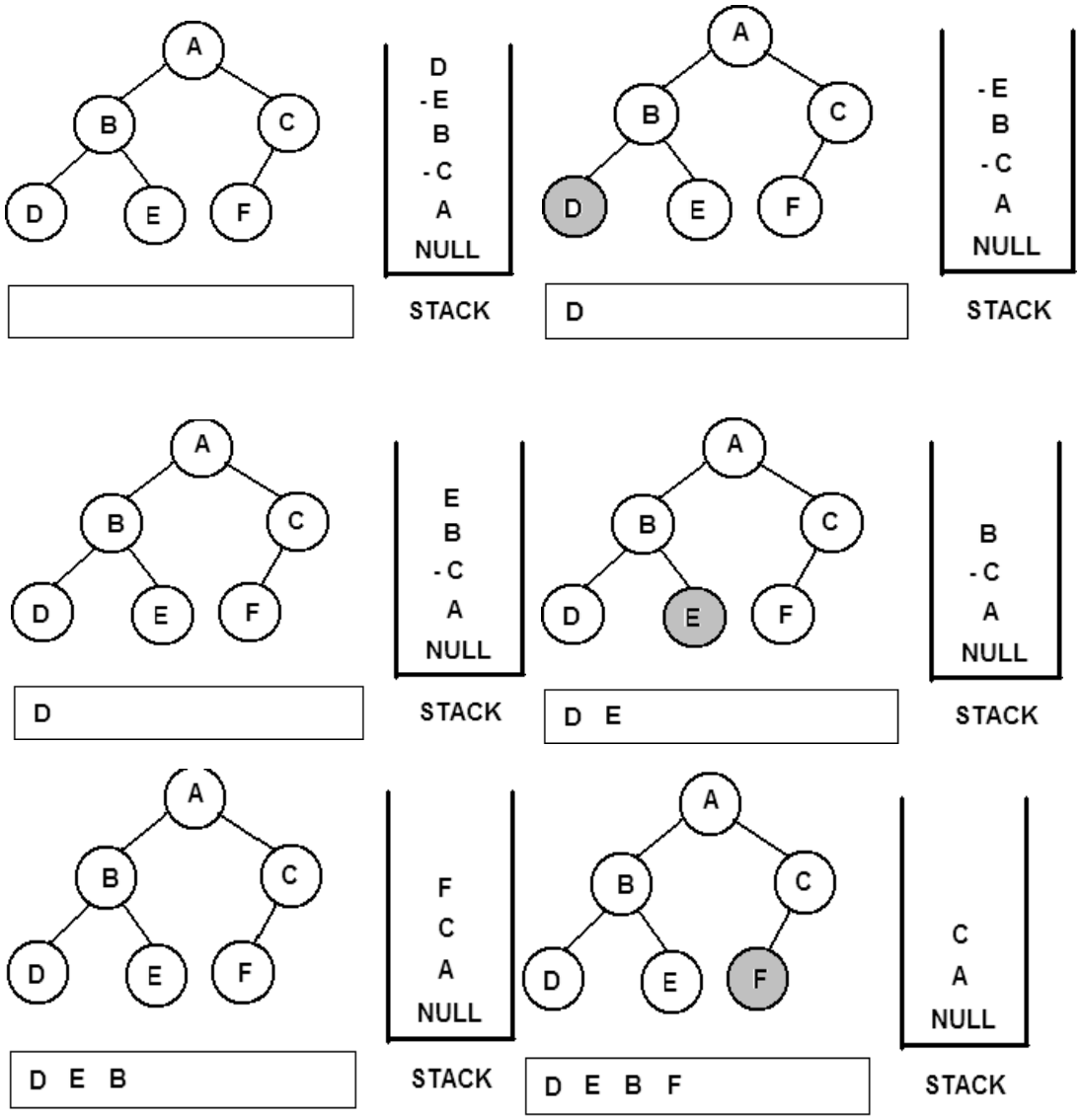
End while

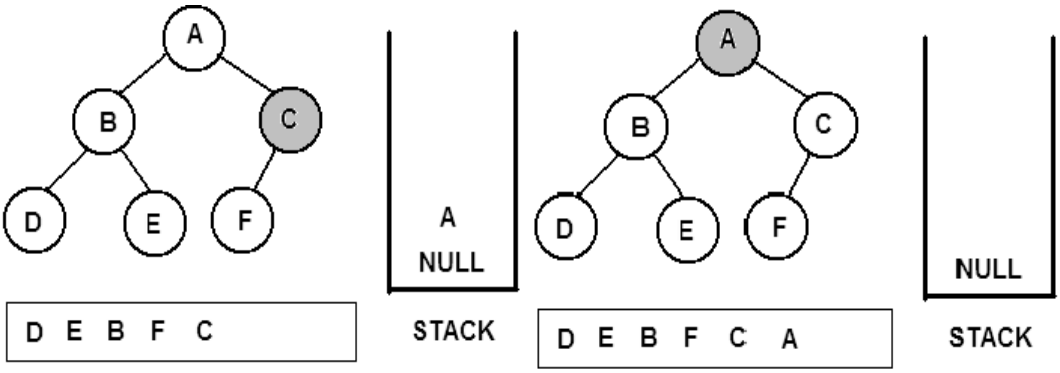
End INORDER

Postorder Traversal

- In the postorder traversal method, a stack is initially stored with a NULL value.

- A pointer temp is made to point to the root node. A flag RIGHTEXISTS is set to 1. A loop is started and continued until this flag is 1.
- The current node is pushed into the stack and it is checked if it has a right child. If so, the negative of value of that node is pushed into the stack and the temp is moved to its left child if it exists.
- This process is repeated till the temp reached a NULL value.
- Now the values in the stack are popped one by one and are pointed by temp. If the value popped is positive then that node is processed. If the value popped is negative, then the value is negated and pointed by temp.
- The flag RIGHTEXISTS is set to 1 and the same above process repeats. This continues till the NULL value from the stack is popped.





POSTORDER (ROOT)

Temp = ROOT, push (NULL), RIGHTEXISTS = 1

While RIGHTEXISTS = 1

RIGHTEXISTS = 0

While temp != NULL

Push (temp)

If right(temp) != NULL

Push (- right(temp))

End if

Temp =left(temp)

End while

Do

Temp = pop ()

If temp > 0

Print info(temp)

End if

If temp < 0

Temp = -temp

RIGHTEXISTS = 1

Break

End if

While temp != NULL

End while

End POSTORDER

EXPRESSION TREE

- The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for 3 + ((5+9) * 2) would be:

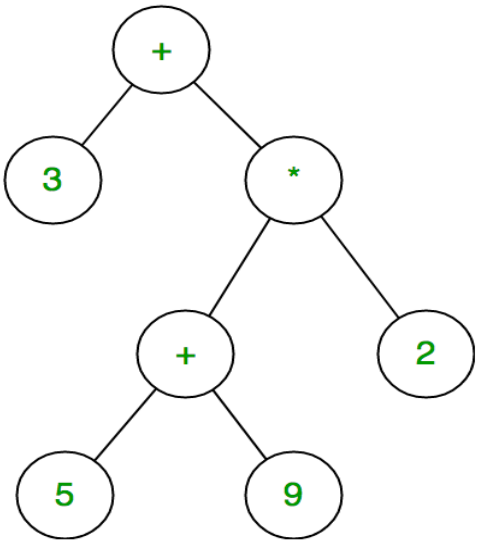


Fig. 5.12 Expression Tree

- Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)
- **Evaluating the expression represented by** an **expression tree:**

```
Let t be the expression tree
If t is not null then
    If t.value is operand then
        Return t.value
    A = solve(t.left)
    B = solve(t.right)
     // calculate applies operator 't.value'
    // on A and B, and returns value
    Return calculate (A, B, t.value)
```

- **Construction of Expression Tree:**
- Now for constructing an expression tree we use a stack. We loop through input expression and do the following for every character.
    - If a character is an operand push that into the stack

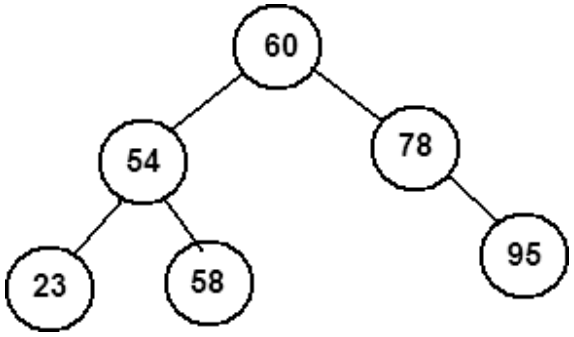
  - If a character is an operator pop two values from the stack make them its child and push the current node again.
- In the end, the only element of the stack will be the root of an expression tree.

APPLICATION OF BINARY TREES

- Manipulation of arithmetic expression
- Construction of symbol table
- Analysis of Syntax
- Writing Grammar
- Creation of Expression Tree

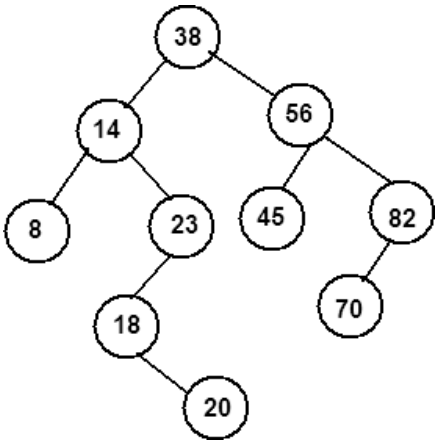
BINARY SEARCH TREES

- Binary Search Tree: A Binary tree T is a Binary Search Tree (BST), if each node N of T has the following property: The value at N is greater than every value in the left subtree of N and is less than every value in the right subtree of N.
- Consider the following tree. The root node 60 is greater than all the elements (54, 23, 58) in its left subtree and is less than all elements in its right subtree (78, 95). Similarly, 54 is greater than its left child 23 and lesser than its right child 58. Hence each and every node in a binary search tree satisfies this property.
- The reason why we go for a Binary Search tree is to improve the searching efficiency. The average case time complexity of the search operation in a binary search tree is O (log n).



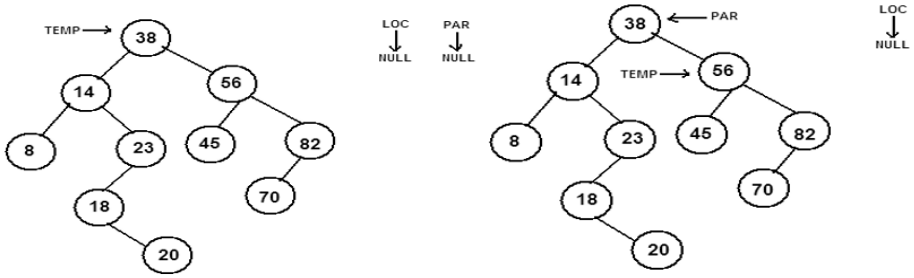
- Consider the following list of numbers. A binary tree can be constructed using this list of numbers, as shown.
- 38 14 8 23 18 20 56 45 82 70
- Initially 38 is taken and placed as the root node. The next number 14 is taken and compared with 38. As 14 is lesser than 38, it is placed as the left child of 38. Now the third number 8 is taken and compared starting from the root node 38. Since is 8 is less

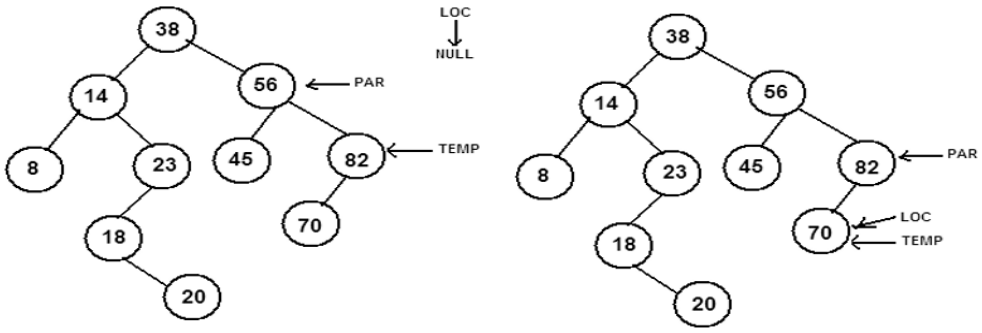
than 38 moves towards left of 38. Now 8 is compared with 14, and as it is less that 14 and also 14 does not have any child, 8 is attached as the left child of 14. This process is repeated until all the numbers are inserted into the tree. Remember that if a number to be inserted is greater than a particular node element, then we move towards the right of the node and start comparing again.



Search Operation in a Binary Search Tree

- The search operation on a BST returns the address of the node where the element is found. The pointer LOC is used to store the address of the node where the element is found. The pointer PAR is used to point to the parent of LOC. Initially the pointer TEMP is made to point to the root node. Let us search for a value 70 in the following BST. Let k = 70. The k value is compared with 38. As k is greater than 38, move to the right child of 38, i.e., 56. k is greater than 56 and hence we move to the right child of 56, which is 82. Now since k is lesser than 82, temp is moved to the left child of 82. The k value matches here and hence the address of this node is stored in the pointer LOC.
- Every time the temp pointer is moved to the next node, the current node is made pointed by PAR. Hence, we get the address of that node where the k value is found, and also the address of its parent node though PAR.





SEARCH (ROOT, k)

Temp = ROOT, par = NULL, loc = NULL

While temp != NULL

If k = info(temp)

Loc = temp Break

End if

If k < info(temp)

Par = temp

Temp = left(temp)

Else

Par = temp

Temp = right(temp)

End if

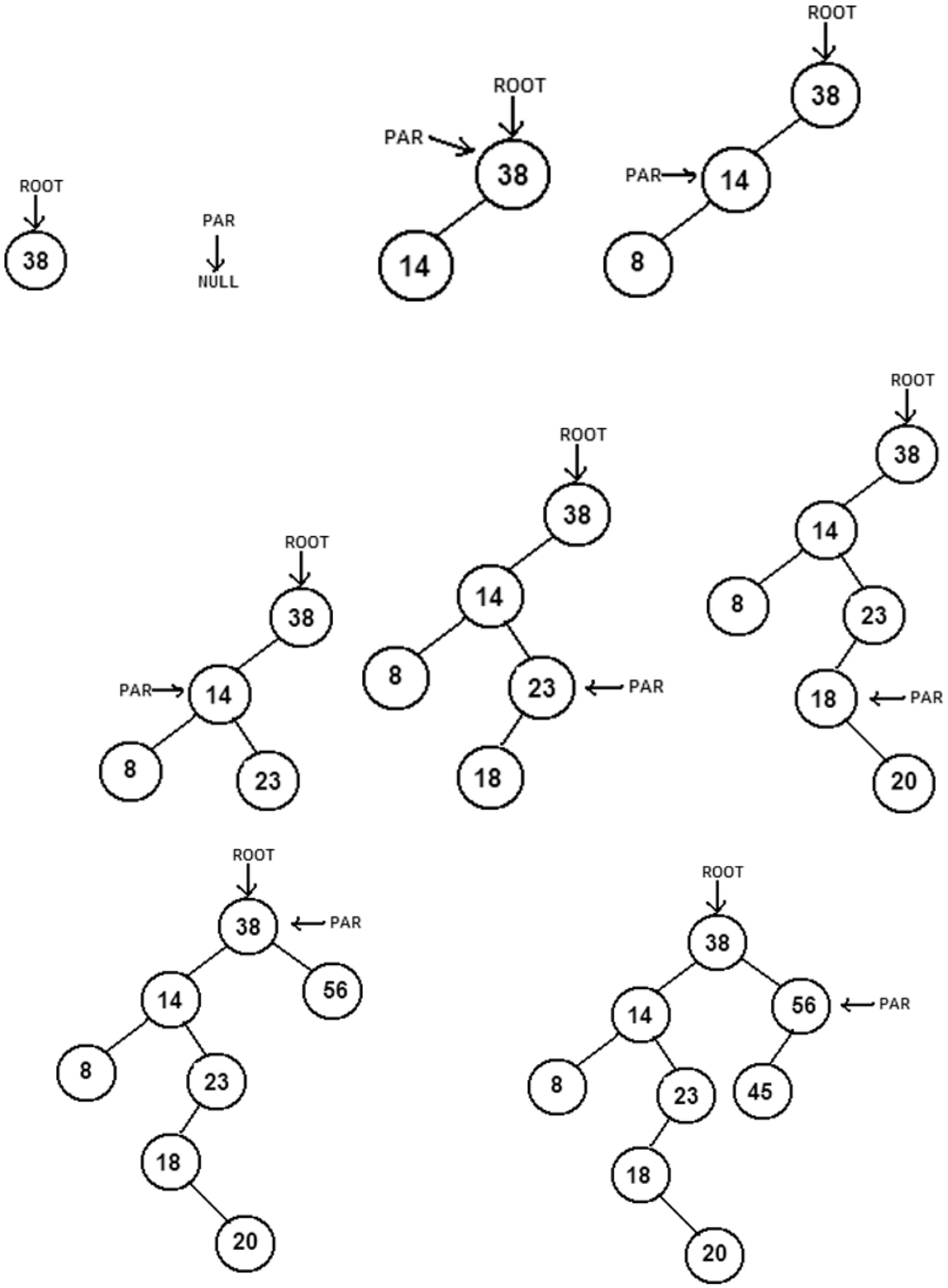
End while

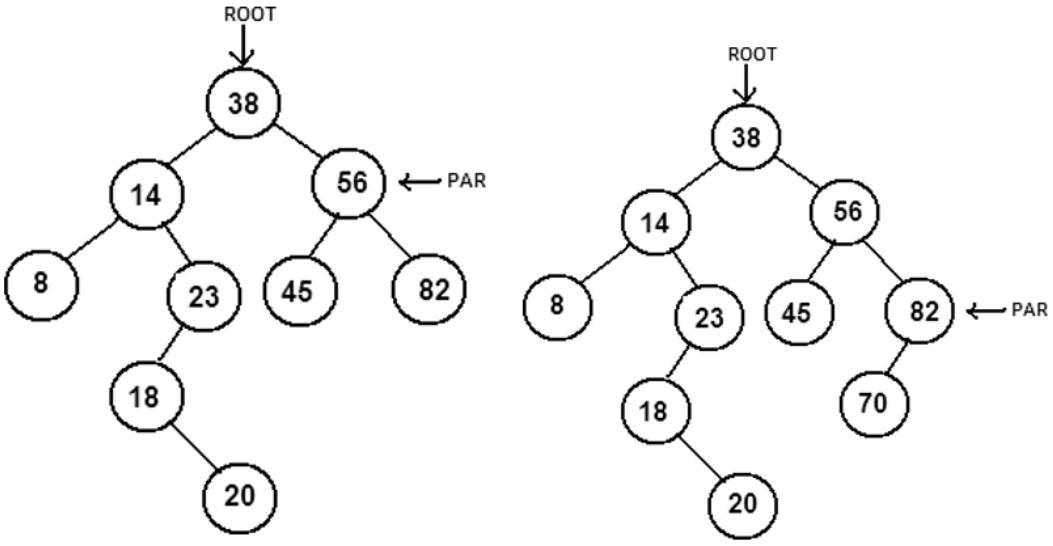
End SEARCH

Insert Operation in a Binary Search Tree

- The BST itself is constructed using the insert operation described below. Consider the following list of numbers. A binary tree can be constructed using this list of numbers using the insert operation, as shown.
- 39 14 8 23 18 20 56 45 82 70
- Initially 38 is taken and placed as the root node. The next number 14 is taken and compared with 38. As 14 is lesser than 38, it is placed as the left child of 38. Now the third number 8 is taken and compared starting from the root node 38. Since is 8 is less than 38 moves towards left of 38. Now 8 is compared with 14, and as it is less than 14 and also 14 does not have any child, 8 is attached as the left child of 14. This process is repeated until all the numbers are inserted into the tree.

- Remember that if a number to be inserted is greater than a particular node element, then we move towards the right of the node and start comparing again.





INSERT (ROOT, k)

Temp = ROOT, par = NULL

While temp != NULL

If k = info(temp)

Print “Item already exists!” Return

End if

If k < info(temp)

Par = temp

Temp = left(temp)

Else

Par = temp

Temp = right(temp)

End if

End while

Info(R) = k, left(R) = NULL, right(R) = NULL

If par = NULL

ROOT = R

End if

If k < info(par)

Left(par) = R

Else

Right(par) = R

End if

End INSERT

Delete Operation in a Binary Search Tree

- The delete operation in a Binary search tree follows two cases. In case A, the node to be deleted has no children or it has only one child. In case B, the node to be deleted has both left child and the right child. It is taken care that, even after deletion the binary search tree property holds for all the nodes in the BST.

DELETE (ROOT, k)

SEARCH (ROOT, k)

If Loc = NULL

Print "Item not found" Return

End if

If right (Loc) != NULL and left (Loc) != NULL

CASEA (Loc, par)

Else

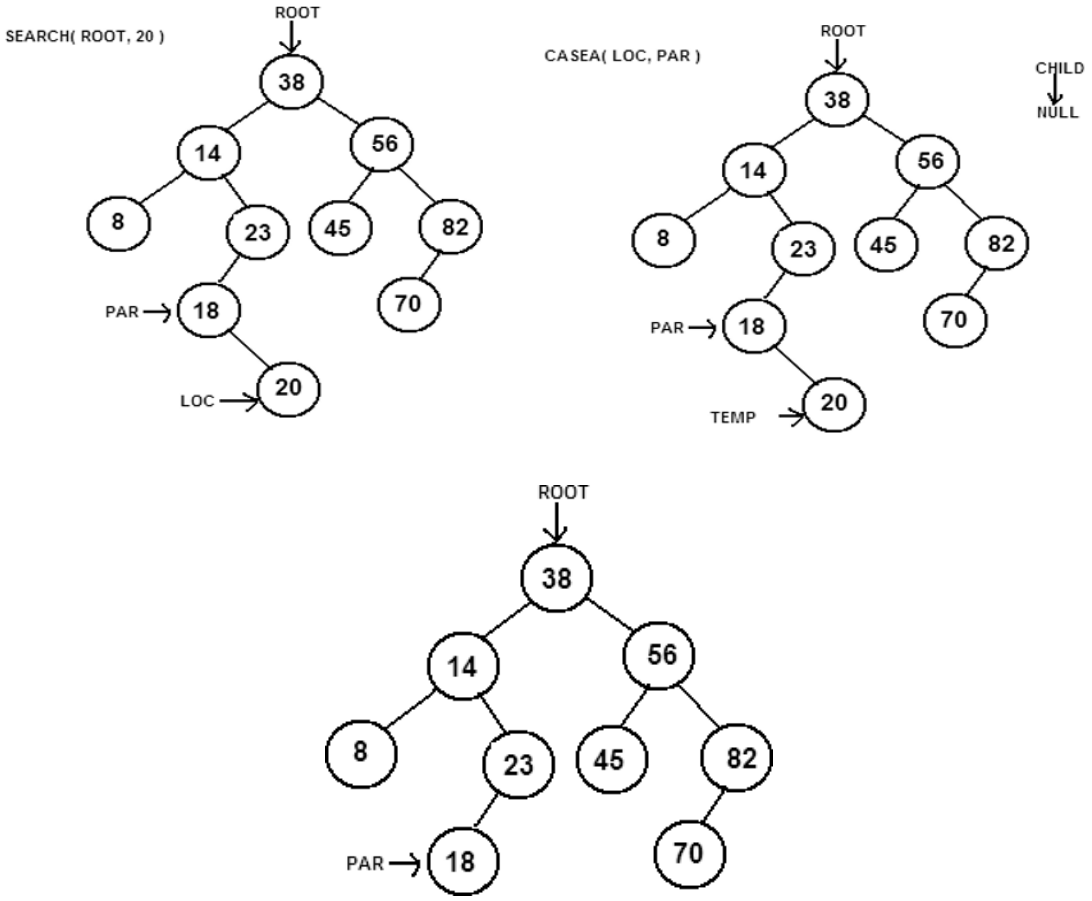
CASEB (Loc, par)

End if

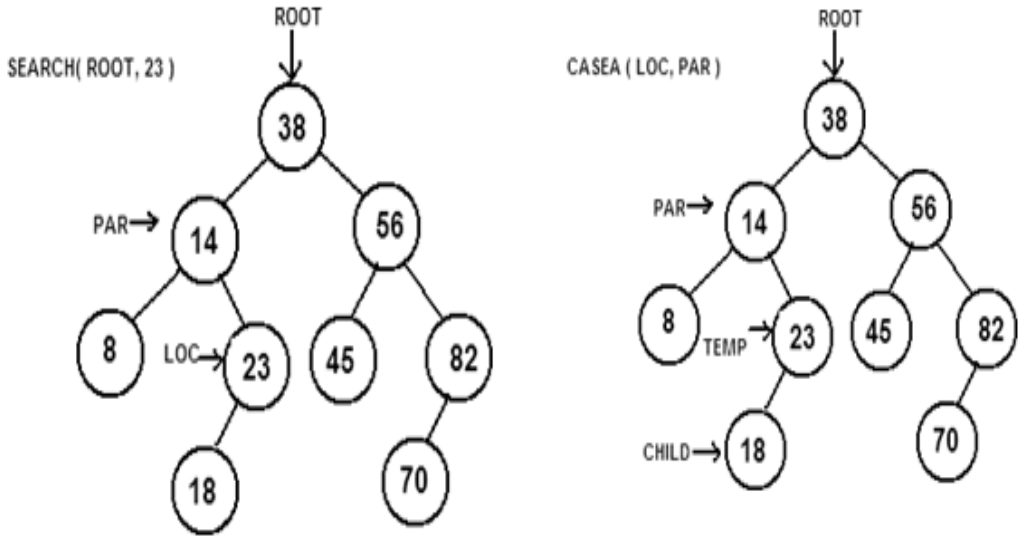
End DELETE

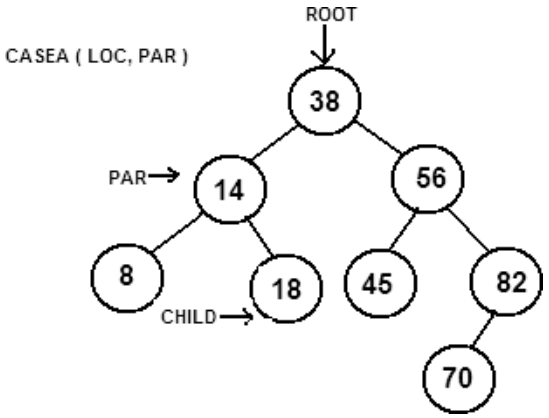
Case A:

- The search is operation is performed for the key value that has to be deleted.
- The search operation, returns the address of the node to be deleted in the pointer LOC and its parents' address is returned in a pointer PAR.
- If the node to be deleted has no children then, it is checked whether the node pointed by LOC is left child of PAR or is it the right child of PAR.
- If it is the left child of PAR, then left of PAR is made NULL else right of PAR is made NULL.
- The sequence of steps followed for deleting 20 from the tree is as shown.



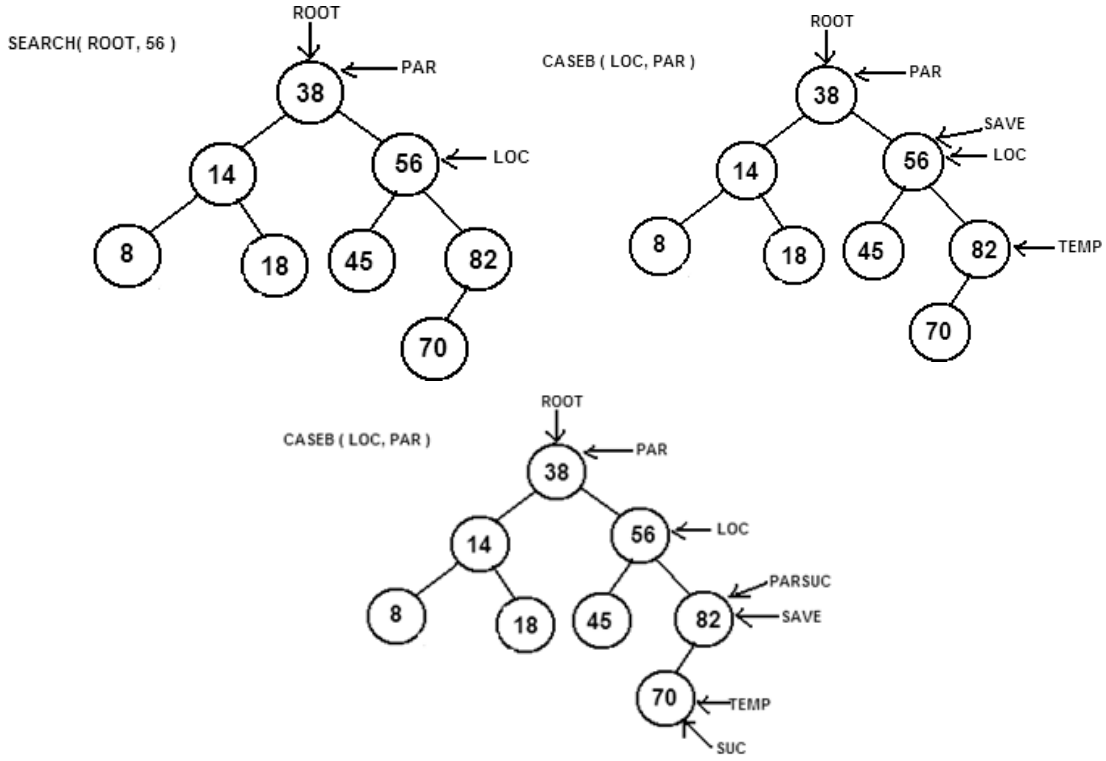
- If the node to be deleted has one child, then a new pointer CHILD is made to point to the child of LOC. If LOC is left child of PAR then left of PAR is pointed to CHILD. If LOC is right child of PAR then right of PAR is pointed to CHILD.
- The sequence of steps for deleting the node 23 is shown.

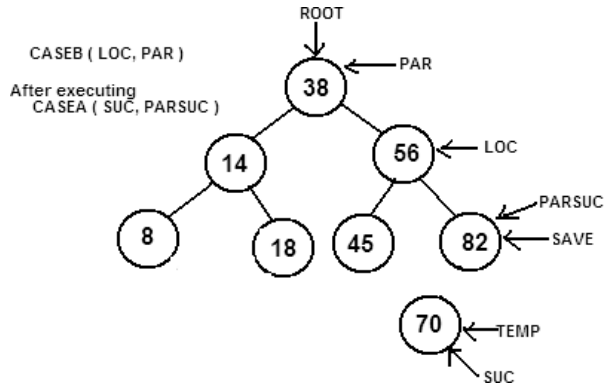


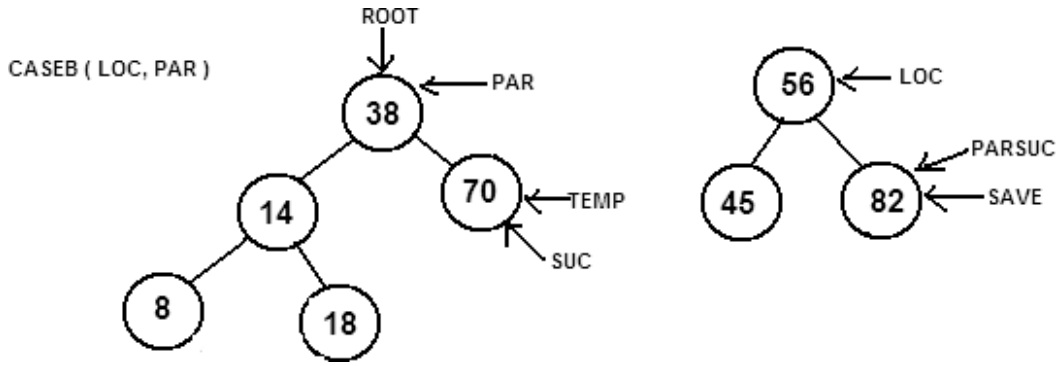


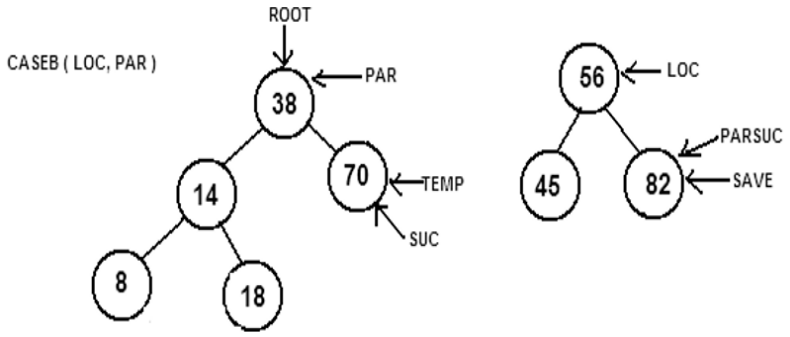
Case B:

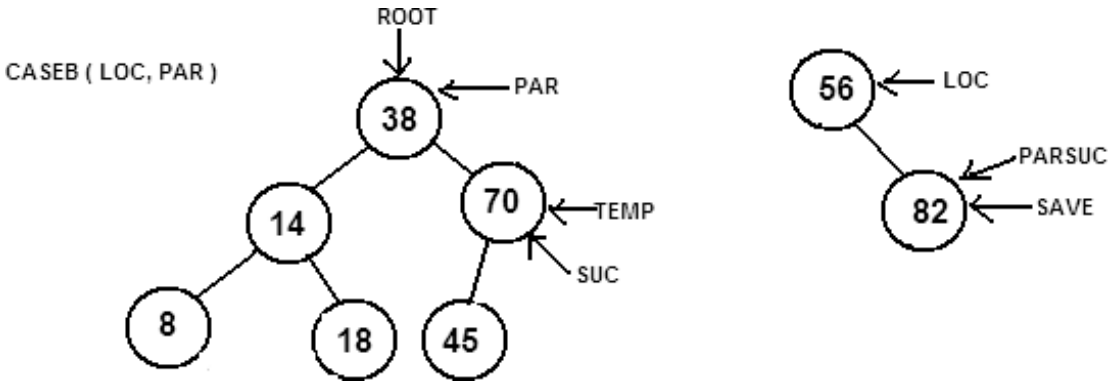
- In this case, the node to be deleted has both the left child and the right child. Here we introduce two new pointers SUC and PARSUC. The inorder successor of the node to be deleted is found out and is pointed by SUC and its parent node is pointed by the PARSUC. In the following example the node to be deleted is 56 which has both the left child and the right child. The inorder successor of 56 is 70 and hence it is pointed by SUC. Now the SUC replaces 56 as shown in the following sequence of steps.

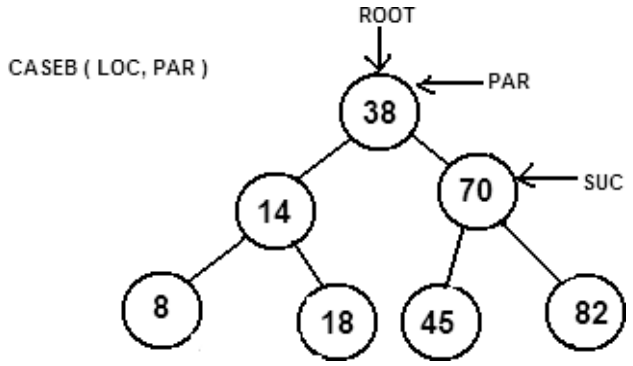



ROOT
CASEB (LOC, PAR)
After executing
CASEA (SUC, PARSUC)
PAR
38
14
56
LOC
8
18
45
82
PARSUC
SAVE
70
TEMP
SUC


ROOT
CASEB (LOC, PAR)
PAR
38
14
70
TEMP
SUC
8
18
56
LOC
PARSUC
45
82
SAVE


ROOT
CASEB (LOC, PAR)
PAR
38
14
70
TEMP
SUC
8
18
56
LOC
PARSUC
45
82
SAVE


ROOT
CASEB (LOC, PAR)
PAR
38
14
70
TEMP
SUC
8
18
45
56
LOC
PARSUC
82
SAVE



CASEA (Loc, par)

Temp = Loc

If left(temp) = NULL and right(temp) = NULL Child = NULL

Else

If left(temp) != NULL

Child = left(temp)

Else

Child = right(temp)

End if

End if

If par != NULL

If temp = left(temp)

Left(par) = child

Else

Right(par) = child

End if

Else

ROOT = child

End if

End CASEA

CASEB (Loc, par)

Temp = right (Loc)

Save = Loc

While left(temp) != NULL

```
Save = temp
Temp = left(temp)
End while
Suc = temp
Parsuc = save
CASEA (suc, parsuc)
If par != NULL
If Loc = left(par)
Left(par) = suc
Else
Right(par) = suc
End if
Else
ROOT = suc
End if
Left(suc) = left (Loc)
Right(suc) = right (Loc)
End CASEB
```

GRAPHS

- **Graph:** A graph G is a defined as a set of objects called nodes and edges.
- G = (V, E)
- A graph G consists of two things:
    - A set V of elements called nodes (or points or vertices)
    - A set E of edges such that each edge e in E is identified with a unique pair [u,v] of nodes in V, denoted by e = [u,v]
- **Node:** A node is a data element of the graph.
- **Edge:** An edge is a path between two nodes.
- There are two types of graph. They are
    - Undirected graph
    - Directed graph
- **Undirected graph:** An undirected graph is a graph in which the edges are directionally oriented towards a node.

- **Directed graph:** A Directed graph or a Digraph is a graph in which the edges are not directionally oriented towards any node.

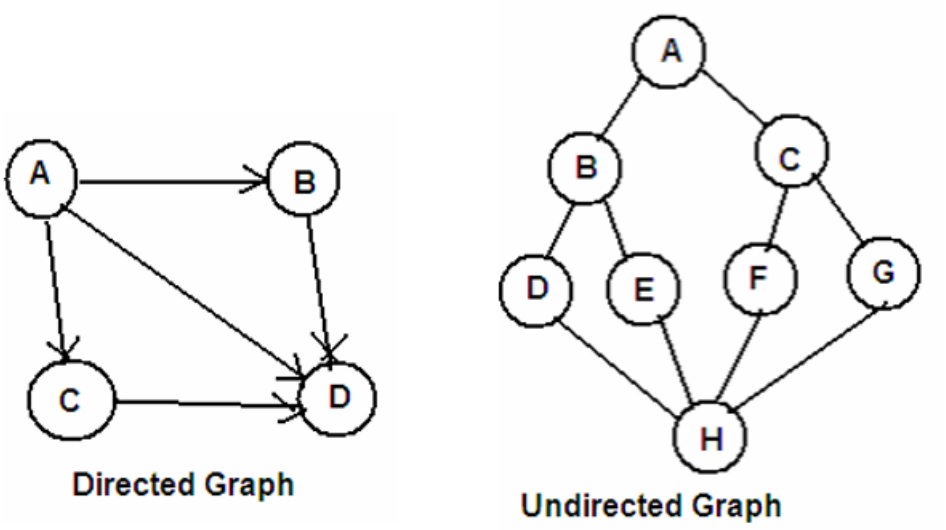


Fig. 5.13 Types of Graph

- **Arc:** The directed edge in a directed graph is called an arc.
- **Strongly connected graph:** A Directed graph is called a strongly connected graph if for any two nodes I and J, there is a directed path from I to J and also from J to I.
- **Weakly connected graph:** A Directed graph is called a weakly connected graph if for any two nodes I and J, there is a directed path from I to J or from J to I.
- **Outdegree:** The number of arcs exiting from the node is called outdegree of that node.
- **Indegree:** The number of arcs entering the node is called indegree of that node.
- **Source node:** A node where the indegree is 0 but has a positive value for outdegree is called a source node. That is there are only outgoing arcs to the node and no incoming arcs to the node.
- **Sink node:** A node where the outdegree is 0 and has a positive value for indegree is called the sink node. That is there is only incoming arcs to the node and no outgoing arcs the node.
- **Cycle:** A cycle in a directed graph is a directed path that originates and terminates at the same node.
- **Length of the path:** The length of the path between node I and K is the number of edges between them in a path from I to K.
- **Degree of a node:** In an undirected graph, the degree of a node is the number of edges connected directly to the node.

For example,

- In the directed graph shown above, the outdegree of A is 3. The indegree of B is 1. The node A is the source node. The node D is the Sink node. The length of the path between A to D is 2.
- **Degree:** The degree of the node B in the undirected graph shown above is 3.

Representation of Graphs

- The graphs can be represented using Adjacency matrix or otherwise called the Incidence matrix.
- The adjacency matrix is a N X N matrix where N is the number of nodes in the graph. Each entry (I, J) in the matrix has either 1 or 0. An entry 1 indicates that there is a direct connection from I to J. An entry 0 indicates that there is no direct connection from I to J.
- If an adjacency matrix is written for the above directed graph as shown:

	A	B	C	D
A	0	1	1	1
B	0	0	0	1
C	0	0	0	1
D	0	0	0	0

Graph Traversals

- There are two methods for traversing through the nodes of the graph. They are:
    - Breadth First Search Traversal (BFS)
    - Depth First Search Traversal (DFS)

Breadth First Search Traversal (BFS)

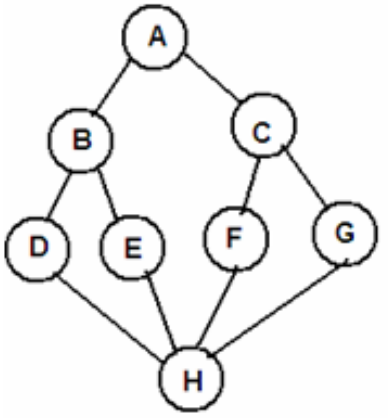
- As the name implies, this method traverses the nodes of the graph by searching through the nodes breadth-wise. Initially let the first node of the graph be visited. This node is now considered as node u. Now find out all the nodes which are adjacent to this node. Let all the adjacent nodes be called as w. Add the node u to a queue. Now every time an adjacent node w is visited, it is added to the queue. One by one all the adjacent nodes w are visited and added to the queue. When all the unvisited adjacent nodes are visited, then the node u is deleted from the queue and hence the next element in the queue now becomes the new node u. The process is repeated on this new node u. This is continued till all the nodes are visited.

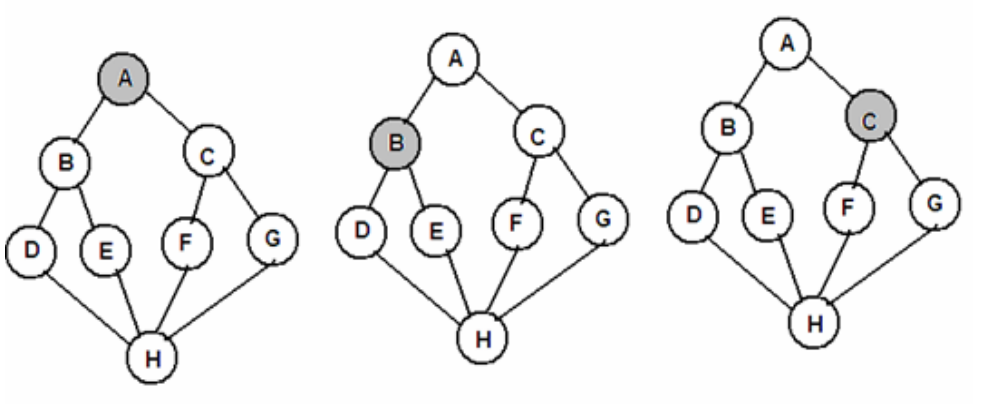
- The Breadth First Traversal (BFT) algorithm calls the BFS algorithm on all the nodes.
- **Algorithm1**

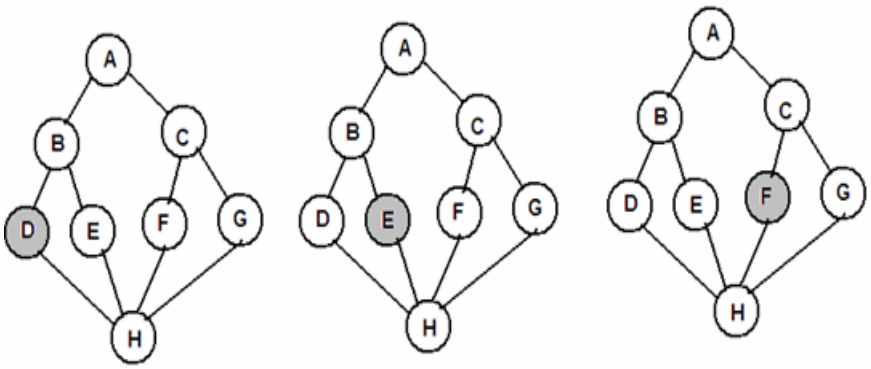
```
BFT (G, n)
Repeat for i = 1 to n
Visited[i] = 0
End Repeat
Repeat for i = 1 to n
If visited[i] = 0
BFS(i)
End if
End Repeat
```

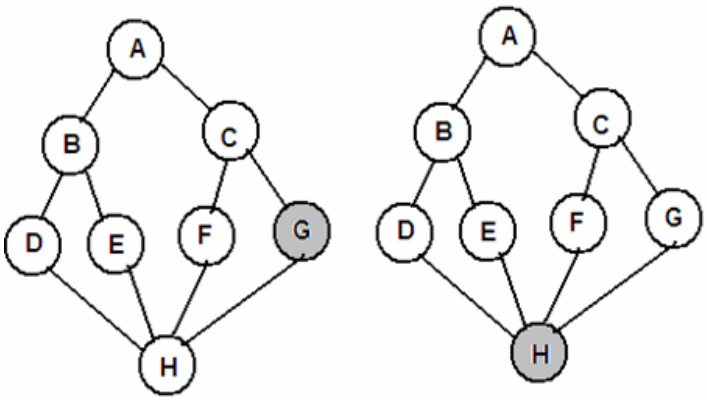
- **Algorithm2**

```
BFS (v)
u = v
visited[v] = 1
Repeat while(true)
Repeat for all vertices w adjacent from u If visited[w] = 0
Add w to queue
Visited[w] = 1
End if
End Repeat
If queue is empty
Return
End if
Delete u from queue
End while
End BFS
```

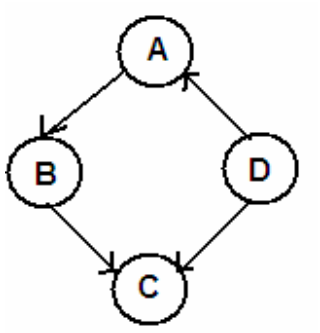

A
B
C
D
E
F
G
H

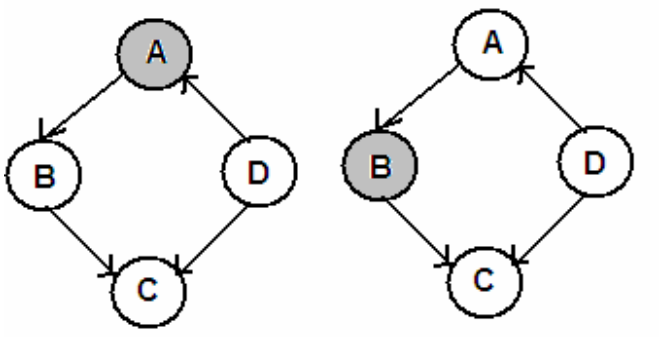

A
B
C
D
E
F
G
H
A
B
C
D
E
F
G
H
A
B
C
D
E
F
G
H

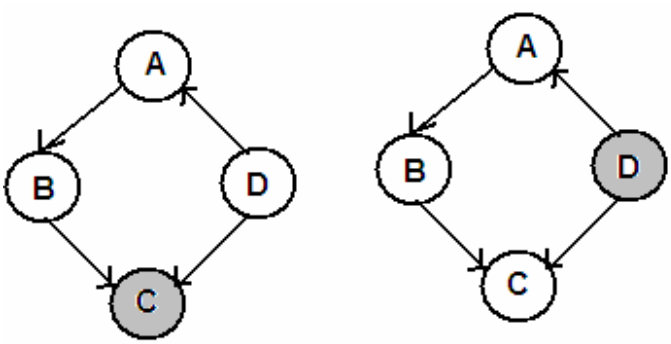

A
B
C
D
E
F
G
H
A
B
C
D
E
F
G
H
A
B
C
D
E
F
G
H


A
B
C
D
E
F
G
H
A
B
C
D
E
F
G
H

- Now the following diagrams illustrates the BFS on a directed graph.







Depth First Search Traversal (DFS)

- In the Depth First Search Traversal, as the name implies the nodes of the graph are traversed by searching through all the nodes by first going to the depth of the graph.
- The first node is visited first. Let this be node u. Find out all the adjacent nodes of u. Let that be w.
- Apply the DFS on the first adjacent node recursively. Since a recursive approach is followed, the nodes are traversed by going to the depth of the graph first.
- The DFT algorithm calls the DFS algorithm repeatedly for all the nodes in the graph.
- **Algorithm1**

  DFT (G, n)

Repeat for i = 1 to n

Visited[i] = 0

End Repeat

Repeat for i = 1 to n

If visited[i] = 0

DFS(i)

End if

End Repeat

- **Algorithm2**

DFS(v)

Visited[v] = 1

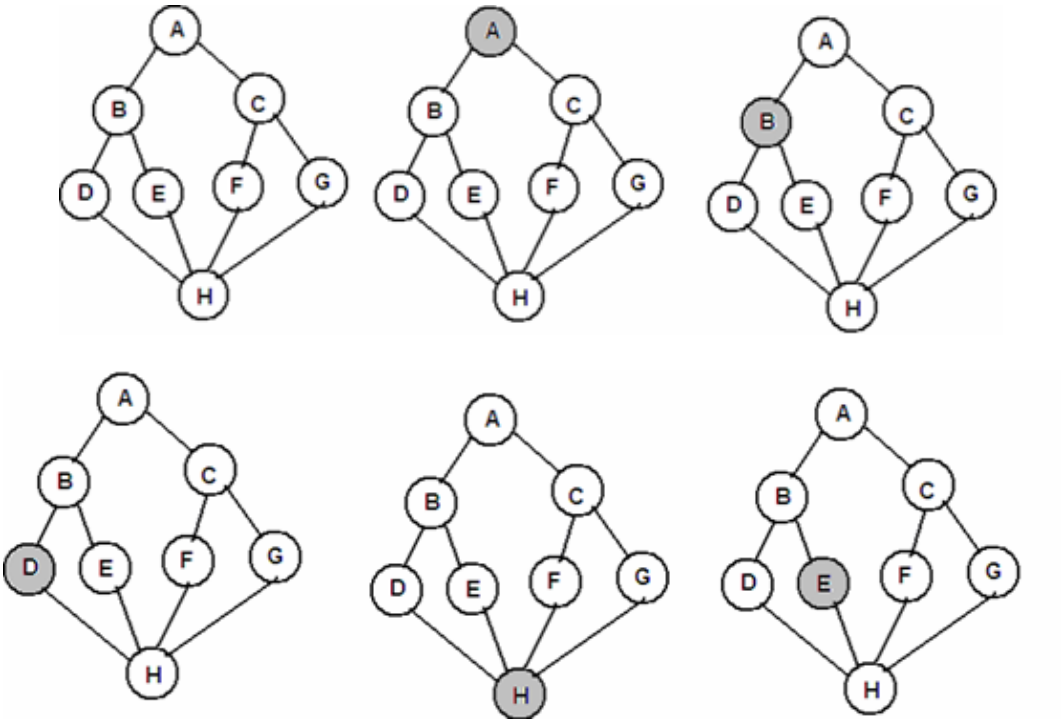
Repeat for each vertex w adjacent from v

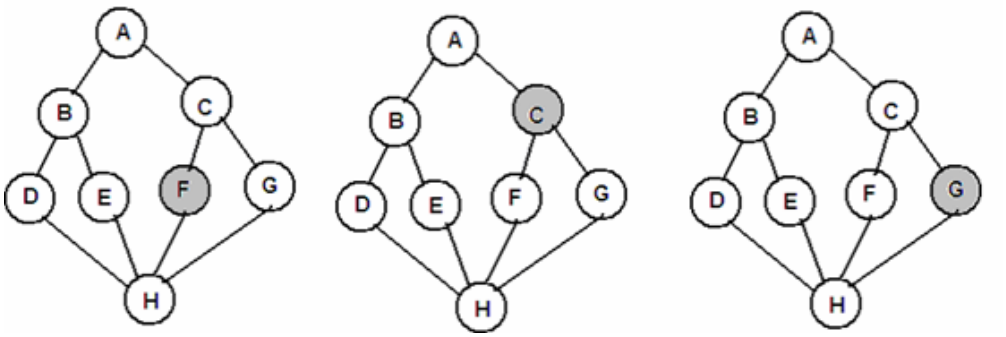
If visited[w] = 0

DFS(w)

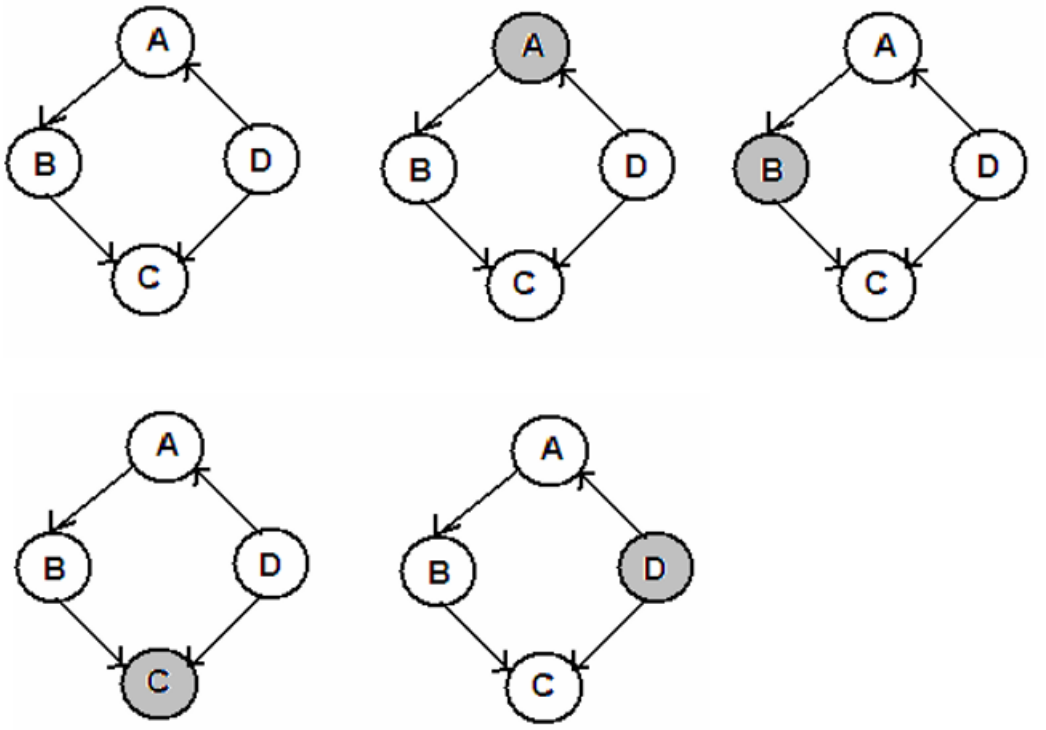
End if

End for



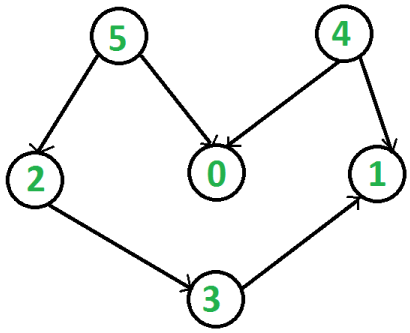


- Now the following diagrams illustrates the DFS on a directed graph.



TOPOLOGICAL SORT

- Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.
- For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”.
- The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



Topological Sorting vs Depth First Traversal (DFS):

- In DFS, we print a vertex and then recursively call DFS for its adjacent vertices.
- In topological sorting, we need to print a vertex before its adjacent vertices.
- For example, in the given graph, the vertex '5' should be printed before vertex '0', but unlike DFS, the vertex '4' should also be printed before vertex '0'.
- So Topological sorting is different from DFS.
- For example, a DFS of the shown graph is "5 2 3 1 0 4", but it is not a topological sorting.

Algorithm to find Topological Sorting:

- We can modify DFS to find Topological Sorting of a graph.
- In DFS, we start from a vertex, we first print it and then recursively call DFS for its adjacent vertices.
- In topological sorting, we use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of the stack.
- Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in the stack.
- **Steps in Topological Sorting**

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge
while S is not empty do
   remove a node n from S
   add n to L
   for each node m with an edge e from n to m do
      remove edge e from the graph
```

```
        if m has no other incoming edges then
            insert m into S
    if graph has edges then
        return error (graph has at least one cycle)
    else
        return L (a topologically sorted order)
```

- **Algorithm**

  topoSort (u, visited, stack)

  Input − The start vertex u, an array to keep track of which node is visited or not. A stack to store nodes.

  Output − Sorting the vertices in topological sequence in the stack.

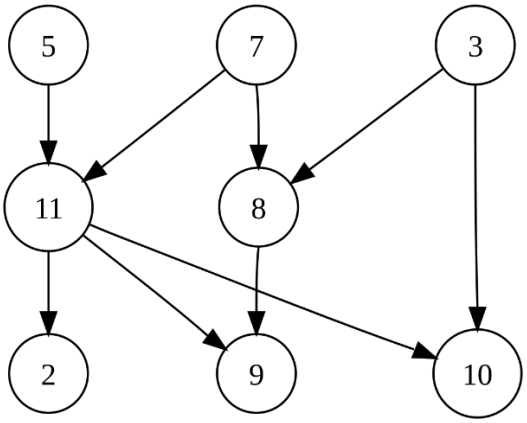
```
Begin
  mark u as visited
  for all vertices v which is adjacent with u, do
    if v is not visited, then
      topoSort(c, visited, stack)
  done
  push u into a stack
End
```

  performTopologicalSorting (Graph)

  Input − The given directed acyclic graph.

  Output − Sequence of nodes.

```
Begin
  initially mark all nodes as unvisited
  for all nodes v of the graph, do
    if v is not visited, then
      topoSort(i, visited, stack)
  done
  pop and print all elements from the stack
End.
```



- The graph shown above has many valid topological sorts, including:
    - 5, 7, 3, 11, 8, 2, 9, 10 (visual top-to-bottom, left-to-right)
    - 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
    - 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
    - 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
    - 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
    - 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)
- **Applications:**
    - Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs.
    - In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linkers