UNIT-I: INTRODUCTION TO DS

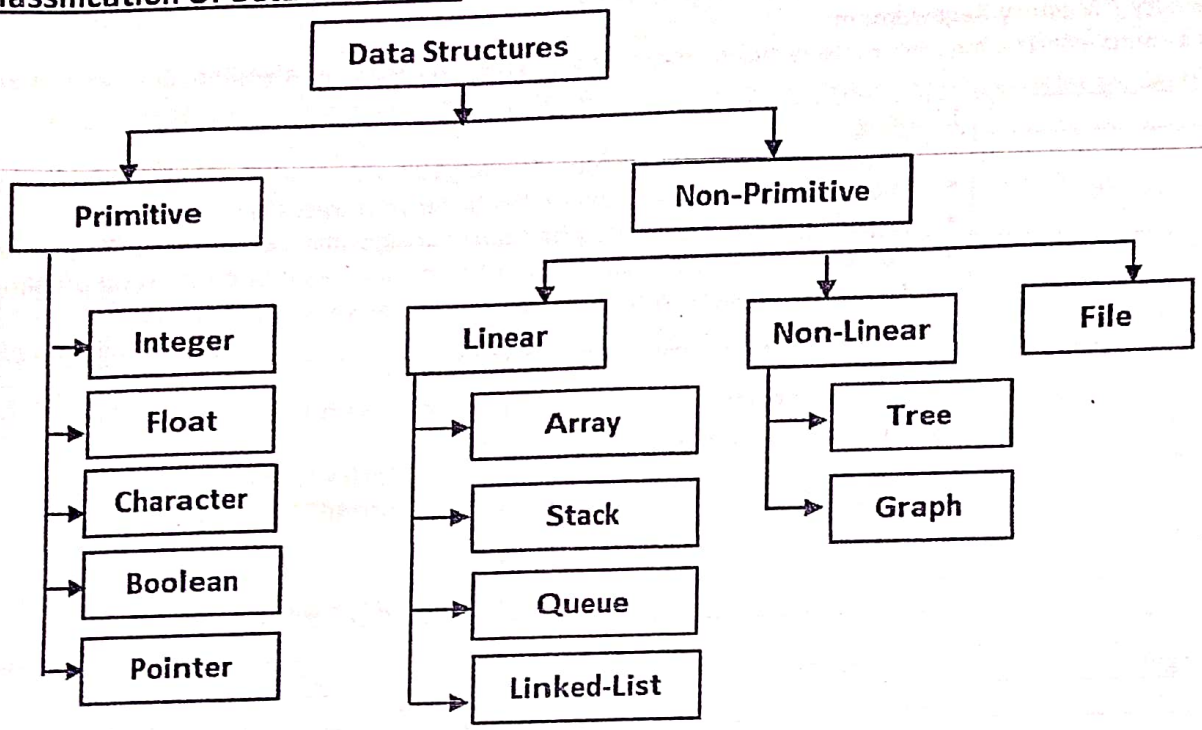
Data:

- Data means set of values or information

Data Structures:

- Data Structure is a format or container or shape or mold to store or arrange or organize or represent the data.
- We have different data structures like Array, Stack, Queue, Tree, Graph, Linked-List, etc.
- The data must be arranged/ stored/ represented in such a format so that it can be accessed and modified efficiently.

Classification of Data structures



Primitive Data Type:

- A primitive data type is a data type that is built into a programming language.

Non-Primitive Data Type:

- Non-primitive data types are created by the programmer and they are not predefined.

Linear Data Structures:

- A Linear data structure have data elements arranged in sequential manner (i.e. not necessary to be consecutive) and each member element is connected to its exactly one previous and one next element.

Non-linear Data Structure:

- Data structures where data elements are not arranged sequentially or linearly are called non-linear data structures.

Algorithm:

- An algorithm is a well-defined list of steps for solving a particular problem.
- The efficiency of an algorithm is calculated on the basis of time required for processing and memory required to store and process.
- The best Algorithm is that which requires less memory space and takes less time to generate the output.
- But, many times it is not possible to achieve both the above conditions at the same time.

Complexity:

- The complexity is the criteria to compare algorithms.
- The complexity of an algorithm is the function which gives the running time and/ or space in term of input size.
- In simple words, the complexity of an algorithm refers to how fast or slow a particular algorithm performs.
- The efficiency of an algorithm depends on two parameters:
  1. Time Complexity
  2. Space Complexity

Time Complexity / Performance Requirement:

- Time complexity is a measure of: How long time an algorithm or program takes to produce output.

Space Complexity / Memory Requirement:

- The Space complexity is a measure of: How much total memory space required by an algorithm or program to comple the task.

Notation	Meaning
O (Big Oh)	▪ It gives worst case complexity of an algorithm. ▪ The smaller your Big-O notation the faster your algorithm ▪ It specifically describes: How bad can your algorithm run as the inputs gets large. ▪ Suppose, an array contains 10 or 20 elements then it will not take more time to ru however the bigger your array gets the slower your algorithm gets. ▪ O(n) Read as Order of n
Ω (Big Omega)	It gives best case complexity of an algorithm
Θ (Theta)	It gives average case complexity of an algorithm

Example:

```
for(i = 1; i <= n; i++)
printf("%d", i);  // O(1)

n * O(1) = O(n)
```

Example:

```
for(i = 1; i <= n; i++)
{
        for(j = 1; j <= n; j++)
        printf("%d", i * j);
}

n * n * O(1) = O(n^2)
```

Time-Space Trade-Off in Algorithms:

- A trade-off is a situation where one thing increases and another thing decreases.
- Sometimes to save time we will use more memory space.
- Sometimes to save memory space we will spend more time.

Operations on Data Structures:

1. **Traversing:**
   - Traversing means visiting every element in a data-structure to apply some process.
   - Traversing is also called as iterating over the data structure.

2. **Searching:**
   - Searching is the process of finding the location or information of a given search key (i.e. value) in a data structure
   - Searching indicates whether the search key is present in the data structure or not.

3. **Insertion:**
   - Insert operation adds (i.e. inserts) a new element into a data structure.

4. **Deletion:**
   - Delete operation removes an existing element from the data structure.

5. **Sorting:**
   - Sorting is a process or technique of arranging the elements in a data structure in ascending or in descending order.

Abstract Data Type (ADT):

- Abstract Data Type (ADT) is a data type, where only behavior is defined but not implementation.
- Opposite of ADT is Concrete Data Type (CDT), where it contains an implementation of ADT.
- Stack, Queue, List, Tree, table etc. are ADT.
- Each of these ADT has many implementations i.e. CDT.

Example:

- List is an abstract data type that is implemented using an array and linked list.
- A queue is implemented using linked-list-based queue, array-based queue, etc.

Program:
Implement C program for performing following operations on Array: Creation, Insertion, Deletion, Search, and Display.

```
#include<stdio.h>
#include<stdlib.h>
#define MAXSIZE 100
int DATA[MAXSIZE], N;

int search(int VALUE)
{
        int LOC = -1, i;

        if(N == 0)
        printf("\nCANNOT SEARCH. ARRAY IS EMPTY..\n");
        else
        {
                for(i = 0; i < N; i++)
                {
                        if(VALUE == DATA[i])
                        {
                                LOC = i;
                                break;
                        }
                }
        }

        return(LOC);
}

main()
{
        int i, OPTION, INDEX, VALUE, LOC;
        char c;

        printf("HOW MANY NUMBERS YOU WANT TO STORE?\n(MAXSIZE = 100): ");
        scanf("%d", &N);

        if(N > 0 && N <= 100)
        {
                printf("\nENTER %d INTEGERS: ", N);
```

```
for(i = 0; i < N; i++)
scanf("%d", &DATA[i]);

do
{
        //fflush(stdin);
        //printf("PRESS ANY KEY TO CONTINUE..");
        //scanf("%c", &c);
        //system("CLS");

        printf("\n1: INSERT\n");
        printf("2: DELETE\n");
        printf("3: SEARCH\n");
        printf("4: DISPLAY / TRAVERSE\n");
        printf("5: EXIT\n");
        printf("ENTER VALID OPTION: ");
        scanf("%d", &OPTION);

        switch(OPTION)
        {
                case 1: // INSERT
                        if(N == MAXSIZE)
                        printf("\nOVERFLOW  ARRAY IS FULL. CANNOT INSERT\n");
                        else
                        {
                                printf("ENTER INDEX TO INSERT\nTHE INDEX MUST BE IN THE RANGE 0
TO %d: ", N);
                                scanf("%d", &INDEX);

                                if(INDEX >= 0 && INDEX <= N)
                                {
                                        printf("ENTER A VALUE TO INSERT: ");
                                        scanf("%d", &VALUE);

                                        if(INDEX == N)
                                        {
                                                DATA[INDEX] = VALUE;
                                                N++;
                                                printf("SUCCESS! VALUE INSERTED..\n");
                                        }
                                        else
                                        {
                                                // shifting
                                                for(i = N - 1; i >= INDEX; i--)
                                                {
                                                        DATA[i+1] = DATA[i];
                                                }

                                                DATA[INDEX] = VALUE;
                                                N++;
                                                printf("\nSUCCESS! VALUE INSERTED..\n");
                                        }
                                }
                                else
```



```
                                        printf("\nINVALID INDEX..\n");
                                }
                                break;

                case 2: // DELETE
                        if(N == 0)
                        printf("\nARRAY IS EMPTY. CANNOT DELETE..\n");
                        else
                        {
                                printf("\nENTER A VALUE TO DELETE: ");
                                scanf("%d", &VALUE);
                                LOC = search(VALUE);

                                if(LOC == -1)
                                printf("VALUE NOT FOUND. CANNOT DELETE\n");
                                else
                                {
                                        if(LOC == N-1)
                                        N--;
                                        else
                                        {
                                                for(i = LOC; i < N; i++)
                                                DATA[i] = DATA[i + 1];

                                                N--;
                                        }
                                        printf("SUCCESS! VALUE DELETED..\n");
                                }
                        }
                        break;

                case 3: // SEARCH
                        if(N == 0)
                        printf("\nARRAY IS EMPTY. CANNOT SEARCH..\n");
                        else
                        {
                                printf("\nENTER A VALUE TO SEARCH: ");
                                scanf("%d", &VALUE);

                                LOC = search(VALUE);
                                if(LOC == -1)
                                printf("VALUE NOT FOUND..\n");
                                else
                                printf("VALUE FOUND AT INDEX %d\n", LOC);
                        }
                        break;

                case 4: // DISPLAY
                        printf("\nARRAY SIZE: %d\n", N);
                        if(N == 0)
                        printf("ARRAY IS EMPTY..\n");
                        else
                        {
                                printf("ARRAY: ");
```

```
                    for(i = 0; i < N; i++)
                    printf("%d ", DATA[i]);
                    printf("\n");

                }
                break;

            case 5: // EXIT
                printf("\nPRESS ANY KEY TO EXIT..\n");
                break;

            default: printf("\nINVALID OPTION..\n");

        }
    }
    while(OPTION != 5);
}
else
printf("ERROR: INVALID VALUE..\n");
}
```

UNIT- II: SEARCHING AND SORTING

SEARCHING:

- Searching is the process of finding the location or index or information of a given search key (i.e. value) in a structure.
- Searching indicates whether the search key (i.e. value) is present in the data structure or not.
- **Types of searching:**
  1. Linear Search
  2. Binary Search

LINEAR SEARCH:

- It is suitable for unsorted list.
- In this technique we will compare the given key or data item (i.e. search value) with each list element, the comparison will begin from the first list element to last list element.
- If the required data item or key or value is found then index of its first occurrence will be displayed otherwise "Not Found" message will be displayed.

Note:

- In algorithm the array indexes begins with 1.
- In program array indexes begins with 0.

Algorithm:

Variable	Description
DATA	It is an array with N elements
ITEM	It is a variable which contains the data item (i.e. value) to search
LOC	It is a variable which contains the location (i.e. index) of required search value.


```
1. Set DATA[N + 1] := ITEM
2. Set LOC := 1

3. Repeat while DATA[LOC] != ITEM:
4.       Set LOC := LOC + 1
   [End of Step-3 loop]

5. If LOC = N+1, Then:
6.       Set LOC := 0
7.       Write: Not Found
8. Else:
9.       Write: Found at location, LOC
   [End of Step-5 If]
10. Exit
```

Program:

```
#include<stdio.h>
#define MAXSIZE 100
main()
{
    int DATA[MAXSIZE], N, i;
    int ITEM, LOC;

    printf("HOW MANY NUMBERS YOU WANT TO STORE? (MAXSIZE = 99): ");
    scanf("%d", &N);

    if(N > 0 && N <= 99)
    {
        printf("ENTER %d INTEGERS: ", N);

        for(i = 0; i < N; i++)
        scanf("%d", &DATA[i]);

        printf("ENTER A VALUE TO SEARCH: ");
        scanf("%d", &ITEM);

        // LINEAR SEARCH
        DATA[N] = ITEM;
        LOC = 0;

        while(DATA[LOC] != ITEM)
        LOC = LOC + 1;

        if(LOC == N)
        {
            LOC = -1;
            printf("NOT FOUND..\n");
        }
        else
        printf("FOUND AT INDEX %d\n", LOC);
    }
    else
    printf("\nINVALID ARRAY SIZE..\n");
}
```

Output:
HOW MANY NUMBERS YOU WANT TO STORE? (MAXSIZE = 99): 5 <ENTER>
ENTER 5 INTEGERS: 78 85 79 91 77 <ENTER>
ENTER A VALUE TO SEARCH: 79 <ENTER>
FOUND AT INDEX 2

BINARY SEARCH:

- The basic requirement of binary search is that the list of elements (i.e. values) must be in ascending sorted order.
- Binary search is suitable for sorted list.
- The binary search is not suitable for unsorted list because, first we need to sort the list and then binary search algorithm will work.
- **Steps:**

1. In this technique the search item is compared with the center (i.e. MID) element of the list.
   If the item is found at MID location, then the location (i.e. index) is written and exit operation is takes place. If the item is not found at MID then we will check the item is less or greater than MID.
2. If the item is less than the MID element then the search criteria is limited from the first list element to MID-1 list element and again steps 1, 2, 3 repeated.
3. If the item is greater than the MID element then the search criteria is limited from MID+1 list element to last element in the array and again steps 1, 2, 3 repeated.
4. If the item is not found then "Not Found" message is written.

Algorithm for binary search:

Variable	Description
DATA	It is an array with N elements in ascending sorted order
ITEM	It is a variable which contains the data-item (i.e. value) to search.
LOC	It is a variable which contains the location (i.e. index) of the data-item (i.e. value to search).
LB	(Lower Bound) It is a variable which contains the start index of the array.
UB	(Upper Bound) It is a variable which contains the end index of the array.
BEG	It is a variable which contains the start index of the search criteria
MID	It is a variable which contains the index of the mid element in the search criteria
END	It is a variable which contains the end index of the search criteria

1. Set BEG := LB, END := UB and MID := INT((BEG + END)/2)

2. Repeat Steps 3 to 7 while BEG <= END and DATA[MID] != ITEM

3. If ITEM < DATA[MID], Then:
4. Set END := MID -1
5. Else
6. Set BEG := MID + 1
[End of Step-3 If]

7. Set MID := INT((BEG + END)/2)
[End of Step-2 loop]

8. If DATA[MID] = ITEM, Then:
9. Set LOC := MID
10. Else:
Set LOC := NULL
[End of Step-8 If]
11. Exit

Patil Sir

Program:

```
#include<stdio.h>
#define MAXSIZE 100
main()
{
        int DATA[MAXSIZE], N, ITEM, LOC, i;
        int LB, UB, BEG, MID, END;

        printf("How many numbers you want to store?: ");
        scanf("%d", &N);

        printf("Enter %d integers: ", N);
        for(i = 0; i < N; i++)
        scanf("%d", &DATA[i]);

        printf("Enter a number to search: ");
        scanf("%d", &ITEM);

        LB = 0;
        UB = N - 1;
        BEG = LB;
        END = UB;
        MID = (BEG + END) / 2;

        while(BEG <= END && DATA[MID] != ITEM)
        {
                if(ITEM < DATA[MID])
                END = MID - 1;
                else
                BEG = MID + 1;

                MID = (BEG + END) / 2;
        }

        if(DATA[MID] == ITEM)
        {
                LOC = MID;
                printf("Found at index %d\n", LOC);
        }
        else
        {
                LOC = -1;
                printf("Not Found..\n");
        }
}
```

Output:
How many numbers you want to store?: 5 <Enter>
Enter 5 integers: 56 89 23 10 11 <Enter>
Enter a number to search: 23 <Enter>
Found at index 2

OR

Binary Search:

Patil Sir

1. Set BEG := LB, END := UB, LOC := 0

2. Repeat while BEG <= END:

3. MID := INT((BEG+END)/2)

4. If ITEM = DATA[MID], Then:
5. Set LOC := MID
6. Break
7. Else
8. If ITEM < DATA[MID], Then:
9. Set END := MID-1
10. Else
11. Set BEG := MID+1
[End of step 8 If]
[End of step 4 If]
[End of Step 2 loop]

12. If LOC = 0, Then:
13. Write : 'Item not found.'
14. Else
15. Write : 'Item found at index ', LOC
[End of step 12 If]
16. Exit

Program:

```
#include<stdio.h>
#define MAXSIZE 100
main()
{
        int DATA[MAXSIZE], N, i, ITEM;
        int LB, UB, BEG, MID, END, LOC;

        printf("HOW MANY NUMBERS YOU WANT TO STORE?(MAXSIZE = 100): ");
        scanf("%d", &N);

        if(N > 0 && N <= 100)
        {
                printf("ENTER %d INTEGERS IN ASCENDING SORTED ORDER: ", N);

                for(i = 0; i < N; i++)
                scanf("%d", &DATA[i]);

                printf("ENTER A VALUE TO SEARCH: ");
                scanf("%d", &ITEM);

                LB = 0;
                UB = N - 1;
                BEG = LB;
                END = UB;
                LOC = -1;

                while(BEG <= END)
                {
```



```
                MID = (BEG + END) / 2;

                if(ITEM == DATA[MID])
                {
                        LOC = MID;
                        break;
                }
                else
                        if(ITEM < DATA[MID])
                        END = MID - 1;
                        else
                        BEG = MID + 1;
        }

        if(LOC == -1)
        printf("NOT FOUND..\n");
        else
        printf("FOUND AT INDEX %d\n", LOC);
    }
    else
    printf("INVALID ARRAY SIZE..\n");
}
```

Output:

```
HOW MANY NUMBERS YOU WANT TO STORE?(MAXSIZE = 100): 5 <ENTER>
ENTER 5 INTEGERS IN ASCENDING SORTED ORDER: 88 91 100 120 150 <ENTER>
ENTER A VALUE TO SEARCH: 88 <ENTER>
FOUND AT INDEX 0
```

SORTING

- Sorting is a process or technique of arranging the elements in a data structure in ascending or in descending order.
- Different sorting techniques are as follows:
  Bubble Sort, Selection Sort, Insertion Sort, Radix Sort or Bucket Sort, Quick Sort etc.

BUBBLE SORT

- In this sorting technique the 1st element is compared with the 2nd element in the list.
  If the 1st element is greater than the 2nd element then exchange of 1st and 2nd elements takes place otherwise n exchange.
- Then the 2nd element is compared with 3rd element again if the 2nd element is greater than the 3rd element the exchange of 2nd and 3rd element takes place otherwise no exchange.
- Same process is repeated up to the last element.
- After comparing up to the last element, the greatest element in the list will be moved at the last position of the list.
- In 2nd pass the above process is repeated up to the last but one element in the list, as the greatest element is alread moved at last position so no need to compare.
- If there are N elements are present in the list then, above process is repeated for N-1 times means the list will be sorte in N-1 passes.

Example: Consider the following list of numbers and sort them in ascending order by using bubble sort technique.

96	87	56	75	43	34	16
DATA[1]	DATA[2]	DATA[3]	DATA[4]	DATA[5]	DATA[6]	DATA[7]

Solution :

Original Array	96	87	56	75	43	34	16
Pass 1	87	96	56	75	43	34	16
	87	56	96	75	43	34	16
	87	56	75	96	43	34	16
	87	56	75	43	96	34	16
	87	56	75	43	34	96	16
	87	56	75	43	34	16	96
Pass 2	56	87	75	43	34	16	
	56	75	87	43	34	16	
	56	75	43	87	34	16	
	56	75	43	34	87	16	
	56	75	43	34	16	87	
Pass 3	56	75	43	34	16		
	56	43	75	34	16		
	56	43	34	75	16		
	56	43	34	16	75		
Pass 4	43	56	34	16			
	43	34	56	16			
	43	34	16	56			
Pass 5	34	43	16				
	34	16	43				
Pass 6	16	34					
Sorted Array	16	34	43	56	75	87	96

Algorithm:

Variable	Description
DATA	It is an array with N elements
PTR	It is a regular variable to store index

1. Repeat for k = 1 to N - 1 by 1: [Passes]
2. Set PTR := 1
3. Repeat while PTR <= (N - k):
4. If DATA[PTR] > DATA[PTR + 1], Then:
5. Exchange DATA[PTR] and DATA[PTR + 1]
   [End of Step-4 If]
6. Set PTR := PTR + 1
   [End of Step-3 loop]

[End of Step-1 loop]
7. Exit

Patil Sir

Reference: Data Structures With C By Seymour Lipschutz
FOR EDUCATIONAL PURPOSE ONLY, NOT FOR SALE.

Program:

```
#include<stdio.h>
#define MAXSIZE 100
main()
{
        int DATA[MAXSIZE];
        int N, i, k, PTR, t;

        printf("HOW MANY NUMBERS YOU WANT TO STORE? (MAXSIZE = 100): ");
        scanf("%d", &N);

        if(N > 0 && N <= 100)
        {
                printf("ENTER %d INTEGERS: ", N);

                for(i = 0; i < N; i++)
                scanf("%d", &DATA[i]);

                // BUBBLE SORT

                for(k = 1; k <= (N-1); k++)
                {
                        PTR = 0;

                        while(PTR < (N-k))
                        {
                                if(DATA[PTR] > DATA[PTR + 1])
                                {
                                        t = DATA[PTR];
                                        DATA[PTR] = DATA[PTR + 1];
                                        DATA[PTR + 1] = t;
                                }

                                PTR = PTR + 1;
                        }
                }

                printf("ASCENDING SORTED ARRAY:\n");
                for(i = 0; i < N; i++)
                printf("%d ", DATA[i]);
                printf("\n");
        }
        else
        printf("INVALID ARRAY SIZE..\n");
}
```

OUTPUT:

```
HOW MANY NUMBERS YOU WANT TO STORE? (MAXSIZE = 100): 5 <ENTER>
ENTER 5 INTEGERS: 100 80 -10 120 95 <ENTER>
ASCENDING SORTED ARRAY:
-10 80 95 100 120
```

Patil Sir

INSERTION SORT

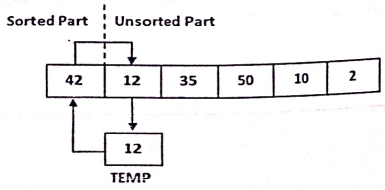
- Consider 1st element in the array as sorted and remaining elements as unsorted array.
- Insert an element from the unsorted array to its correct position in the sorted array.
- Correct position means all the smaller elements must present before the number and all the greater elements present after the number.
- If the list contains N elements then the list will be sorted in N-1 passes.

Example: Insertion Sort

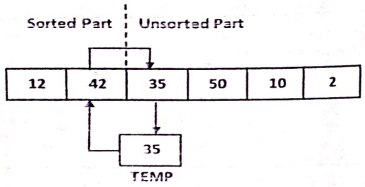
Consider the following list of elements

42	12	35	50	10	2
DATA[1]	DATA[2]	DATA[3]	DATA[4]	DATA[5]	DATA[6]

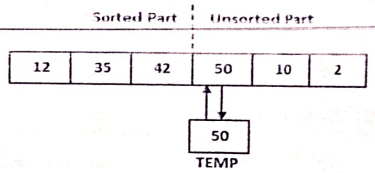
PASS-1



PASS-2

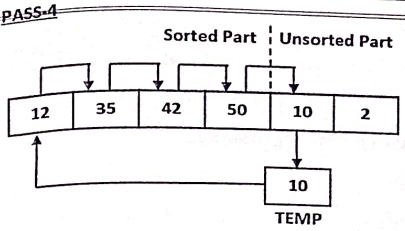


PASS-3

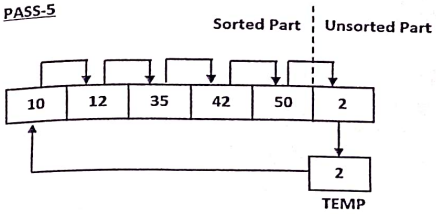


Reference: Data Structures With C By Seymour Lipschutz
FOR EDUCATIONAL PURPOSE ONLY. NOT FOR SALE.

PASS-4



PASS-5



SORTED LIST	2	10	12	35	42	50

Algorithm:

Variable	Description
DATA	It is an array with N integers
TEMP	It is a variable to store value
PTR	It is a variable to store index (Index of previous element)

1. Repeat for k = 2 to N by 1: [k = 2 because unsorted part begins from 2 and end to N]
2. Set TEMP := DATA[k]
3. Set PTR := k – 1
4. Repeat while PTR >= 1 and TEMP < DATA[PTR]:
5. Set DATA[PTR + 1] := DATA[PTR] [It will move previous element to next position]
6. Set PTR := PTR - 1
   [End of Step-4 loop]
7. Set DATA[PTR + 1] := TEMP
   [End of Step-1 loop]
8. Exit

Program:

```
#include<stdio.h>
#define MAXSIZE 100
main()
{
        int DATA[MAXSIZE];
        int N, i, k, TEMP, PTR;

        printf("HOW MANY NUMBERS YOU WANT TO STORE? (MAXSIZE = 100): ");
        scanf("%d", &N);

        if(N > 0 && N <= 100)
        {
                printf(" ENTER %d INTEGERS: ", N);

                for(i = 0; i < N; i++)
                scanf("%d", &DATA[i]);

                // INSERTION SORT

                for(k = 1; k < N; k++)
                {
                        TEMP = DATA[k];
                        PTR = k - 1;      // Index of prev element

                        while(PTR >= 0 && TEMP < DATA[PTR])
                        {
                                DATA[PTR + 1] = DATA[PTR];
                                PTR = PTR - 1;
                        }

                        // INSRTING AT THE CORRECT POSITION
                        DATA[PTR + 1] = TEMP;
                }

                printf("ASCENDING SORTED ARRAY IS AS FOLLOWS:\n");
                for(i = 0; i < N; i++)
                printf("%d ", DATA[i]);
        }
        else
        printf("INVALID ARRAY SIZE..\n");
}
```

OUTPUT:
HOW MANY NUMBERS YOU WANT TO STORE? (MAXSIZE = 100): 5 <ENTER>
ENTER 5 INTEGERS: 88 90 78 91 85 <ENTER>
ASCENDING SORTED ARRAY IS AS FOLLOWS:
78 85 88 90 91

Reference: Data Structures With C By Seymour Lipschutz
FOR EDUCATIONAL PURPOSE ONLY. NOT FOR SALE.

SELECTION SORT

- In first pass find the smallest element in the list and store position (i.e. index) of the smallest element then exchange the 1st list element with the smallest element.
- Then in second pass find second smallest element in the list in between second element to last element and store the position of the smallest element then exchange 2nd list element with the smallest element.
- Repeat this process for remaining elements.
- If the list contains N elements then the list will be sorted in N-1 passes.

Example: Consider the following list of numbers and sort them in ascending order by using selection sort.

77	33	44	11	88	22	66	55
DATA[1]	DATA[2]	DATA[3]	DATA[4]	DATA[5]	DATA[6]	DATA[7]	DATA[8]

Solution:

PASS	DATA[1]	DATA[2]	DATA[3]	DATA[4]	DATA[5]	DATA[6]	DATA[7]	DATA[8]
PASS 1, LOC = 4	77	33	44	**11**	88	22	66	55
PASS 2, LOC = 6	11	33	44	77	88	**22**	66	55
PASS 3, LOC = 6	11	22	44	77	88	**33**	66	55
PASS 4, LOC = 6	11	22	33	77	88	**44**	66	55
PASS 5, LOC = 8	11	22	33	44	88	77	66	**55**
PASS 6, LOC = 7	11	22	33	44	55	77	**66**	88
PASS 7, LOC = 7	11	22	33	44	55	66	**77**	88
SORTED LIST	11	22	33	44	55	66	77	88

Algorithm:

Variable	Description
DATA	It is an array with N integers
MIN	It is a variable which contains the smallest element
LOC	It is a variable which contains the index of the smallest element.

1. Repeat for i = 1 to N - 1 by 1 [Indicates Passes]
   [Assume 1st element is smallest in Pass-1, Assume 2nd element is smallest in Pass-2, ...]
2. Set MIN = DATA[i]
3. Set LOC = i
4. Repeat for j = i + 1 to N by 1:
5. if MIN > DATA[j] then
6. Set MIN = DATA[j]
7. Set LOC = j
   [End of Step-5 if]
   [End of Step-4 inner loop]
8. Set DATA[LOC] = DATA[i] [Moving to smallest's index]
9. Set DATA[i] = MIN [Moving smallest element to pass-index]
   [End of Step-1 outer loop]
10. Exit

Note:

- If index of smallest and pass number is same, the smallest is at its proper position so, no need to exchange.
- So, we can apply if(LOC != i) condition before exchange.
- It will save the exchange time.

Program:

```
#include<stdio.h>
#define MAXSIZE 100
main()
{
	int DATA[MAXSIZE];
	int N, i, MIN, LOC, j;

	printf("HOW MANY NUMBERS YOU WANT TO STORE? (MAXSIZE = 100): ");
	scanf("%d", &N);

	if(N > 0 && N <= MAXSIZE)
	{
		printf("ENTER %d INTEGERS: ", N);
		for(i = 0; i < N; i++)
		scanf("%d", &DATA[i]);

		// SELECTION SORT
		for(i = 0; i < N-1; i++)
		{
			MIN = DATA[i];
			LOC = i;

			for(j = i+1; j < N; j++)
			{
				if(MIN > DATA[j])
				{
					MIN = DATA[j];
					LOC = j;
				}
			}

			DATA[LOC] = DATA[i];
			DATA[i] = MIN;
		}

		printf("\nASCENDING SORTED ARRAY IS AS FOLLOWS:\n");
		for(i = 0; i < N; i++)
		printf("%d ", DATA[i]);
		printf("\n");
	}
	else
		printf("EXCEPTION: INVALID ARRAY SIZE\n");
}
```

OUTPUT:

```
HOW MANY NUMBERS YOU WANT TO STORE? (MAXSIZE = 100): 5 <ENTER>
ENTER 5 INTEGERS: 88 91 87 77 76 <ENTER>
ASCENDING SORTED ARRAY IS AS FOLLOWS:
76 77 87 88 91
```


Reference: Data Structures With C By Seymour Lipschutz
FOR EDUCATIONAL PURPOSE ONLY, NOT FOR SALE.

RADIX SORT

- In this method the number of passes depends on the number of digits of the greatest number in the list.
- In first pass we will place the numbers on the basis of UNIT PLACE digit of that number.
- In second pass we will place the numbers on the basis of TENS PLACE digit of that number
- In third pass we will place the numbers on the basis of HUNDRED PLACE digit of that number.
- Same process is repeated up to the total number of digits in the greatest number in the list.

Example:

Consider the following list of numbers and sort them in ascending order by using radix sort technique.

242	986	428	143	312	101	090
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

Solution:

- **Pass 1** – Place the numbers as per UNIT PLACE

PLACES	ARRAY						
	242	986	428	143	312	101	090
0							090
1						101	
2	242				312		
3				143			
4							
5							
6		986					
7							
8			428				
9							

- The array is as follows :

090	101	242	312	143	986	428
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

- **Pass 2**– Place the numbers as per TENS PLACE

PLACES	ARRAY						
	090	101	242	312	143	986	428
0		101					
1				312			
2							428
3							
4			242		143		
5							
6							
7							
8						986	
9	090						

- The array is as follows :

101	312	428	242	143	986	090
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

- **Pass 3** – Place the numbers as per HUNDRED PLACE

PLACES	ARRAY						
	101	312	428	242	143	986	090
0							090
1	101				143		
2				242			
3		312					
4			428				
5							
6							
7							
8							
9						986	

- The sorted array is as follows :

090	101	143	242	312	428	986
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

Algorithm:

Note:

- For Radix sort we considered array indexes begins with zero.
- If the DATA array contains negative number or zero then the following algorithm will not work.
- The following algorithm will work for numbers greater than zero only.

Variable	Description
DATA	It is an array with N elements
MAX	It is a variable which contains the greatest element in the array
LEN	It is a variable which counts the total number of digits (i.e. length) in the greatest element
PACKETS[10, N]	It is a multi-dimensional array with 10 rows and N columns.

[Procedure:] Int GREATEST (DATA[])
[Assume 1st number is greatest]
1. Set MAX := DATA[0]

2. Repeat for i := 1 to N-1, By 1

3. If MAX < DATA[i]
4. Set MAX := DATA[i]
[End of step 3 If]

[End of Step 2 loop]

5. Return(MAX)

[Procedure to count total number of digits in the greatest number]
Int MAX_LENGTH(MAX)
1. Set LEN := 0

2. Repeat while MAX > 0
3. Set MAX := MAX /10
4. Set LEN := LEN + 1
[End of step 2 loop]
5. Return(LEN)

Patil Sir

Procedure RESET_PACKETS
1. Repeat for x := 0 To 9, By 1

2. Repeat for y := 0 To N-1, By 1
3. Set PACKETS[x][y] := 0
[End of Step-2 loop]

[End of Step-1 loop]
4. Return

RADIX_SORT
1. Read: N Numbers
2. Call RESET_PACKETS()
3. Set MAX := GREATEST(DATA)
4. Set LEN := MAX_LENGTH(MAX)
5. Set Z := 1

6. Repeat for k := 1 to LEN, By 1
[Separates digit and places the number in PACKETS]

7. Repeat for i := 0 To N-1, By 1
8. Set PLACE := MOD(DATA[i] / Z, 10)
9. Set PACKETS[PLACE][i] := DATA[i]
[End of Step-7 loop]

[Clearing the array DATA]
10. Repeat for I = 0 to N-1, by 1
11. Set DATA[i] := 0;
[End of Step-10 loop]

[Arrange the elements from PACKETS to the single dimensional array DATA]

12. Set C := 0

13. Repeat for x := 0 to 9, By 1

14. Repeat for y := 0 to N-1, By 1

15. If PACKETS[x, y] != 0, Then :
16. Set DATA[C] := PACKETS[x, y]
17. Set C := C + 1
[End of Step-15 If]

[End of Step-14 loop]

[End of Step-13 loop]

18. Call RESET_PACKETS()

[Multiply Z by 10 to find next place digit]
19. Set Z := Z * 10
[End of Step-6 loop]
20. Exit

Patil Sir

Program:

```
#include<stdio.h>
#define MAXSIZE 100
int PACKETS[10][MAXSIZE];

void reset_packets(int N)
{
        int x, y;
        for(x = 0; x <= 9; x++)
        {
                for(y = 0; y < N; y++)
                PACKETS[x][y] = 0;
        }
}

main()
{
        int DATA[MAXSIZE], N, MAX, LENGTH = 0;
        int i, j, k, place, x, y, z, c;

        printf("How many numbers you want to store?: ");
        scanf("%d", &N);

        printf("Enter %d integers: ", N);
        for(i = 0; i < N; i++)
        scanf("%d", &DATA[i]);

        /* Find greatest number. Assume 1st number is greatest */
        MAX = DATA[0];
        for(i = 1; i < N; i++)
        {
                if(MAX < DATA[i])
                MAX = DATA[i];
        }

        printf("Greatest number = %d\n", MAX);

        /* Find number of passes. Number of passes is equal to the length of max number */
        while(MAX > 0)
        {
                MAX = MAX / 10;
                LENGTH++;
        }

        printf("Number of passes = %d\n", LENGTH);

        reset_packets(N);

        z = 1;

        for(k = 1; k <= LENGTH; k++)
        {
                for(i = 0; i < N; i++)
```

Patil Sir


```
                {
                        place = (DATA[i]/z)%10;
                        PACKETS[place][i] = DATA[i];
                }

                for(i = 0; i < N; i++)
                DATA[i] = 0;

                c = 0;

                for(x = 0; x <= 9; x++)
                {
                        for(y = 0; y < N; y++)
                        {
                                if(PACKETS[x][y] != 0)
                                {
                                        DATA[c] = PACKETS[x][y];
                                        c = c + 1;
                                }
                        }
                }

                reset_packets(N);
                z = z * 10;
        }

        printf("The asc. sorted array is as follows:\n");
        for(i = 0; i < N; i++)
        printf("%d ", DATA[i]);
}
```

Output:

```
How many numbers you want to store?: 5 <Enter>
Enter 5 integers: 77 62 59 91 10 <Enter>
Greatest number = 91
Number of passes = 2
The asc. sorted array is as follows:
10 59 62 77 91
```

QUICK SORT

- It is divide and conquer algorithm.
- That means it divides the list in the smaller parts, the smaller parts are further divided into smaller parts and apply som logic to solve the problem.

Working:

1. Consider the first element at Pivot (i.e. axis) element.
2. Now, Beginning with the last number, scan the list from right to left, comparing each number with Pivot and stoppi at first number less than pivot, if found then exchange the smaller number and Pivot.
3. As exchange takes place so, change the direction (i.e. choose opposite direction, i.e. left to right) and find first great element than Pivot in between the exchanged elements, if found then exchange the greater element and Pivot.
4. As exchange takes place so, change the direction (i.e. choose opposite direction, i.e. right to left) and find first small element than Pivot in between the exchanged elements, if found then exchange the smaller element and Pivot.
5. If no smaller or greater is found that means the Pivot is at proper position.
6. As Pivot is at proper position, so two sub-lists are created. 1st sub-list before Pivot. 2nd sub-list after Pivot.
7. Repeat the steps 1 to 6 for each sub-list until the sub-list contains single element.

Patil Sir

Note:

- Pivot means axis (केन्द्रबिन्दु, धुरी)
- Whenever the exchange takes place change the direction (i.e. choose opposite direction)
- If the direction is Right to Left then find first smaller element than pivot, if found then exchange.
- If the direction is Left to Right then find first greater element than pivot, if found then exchange.
- If no smaller or greater is found that means the Pivot is at proper position.
- The searching of smaller or greater will be takes place between the exchanged elements.

Example:

Consider following list of elements:

88	77	99	11	22	66

Solution:

- Consider 1st element 88 as Pivot element.

88 (Pivot)	77	99	11	22	66

- Now, beginning from last element 66 scan from Right to Left comparing each number with Pivot (88) and stopping a first number less than Pivot (88).
- Now, 66 is smaller than Pivot (88) so, exchange 66 and Pivot (88)

66	77	99	11	22	88 (Pivot)

- As the exchange took place so, change the direction to Left to right (i.e. choose opposite direction).
- Now, between the exchanged elements (i.e. 66 to Pivot (88)) find the find the first greater element than Pivot (88).
- It is 99, exchange 99 and Pivot (88)

66	77	88 (Pivot)	11	22	99

- As the exchange took place so, change the direction to Right to Left (i.e. choose opposite direction).
- Now, between the exchanged elements (99 and Pivot (88)) find the first smaller element than the Pivot (88).
- It is 22, exchange 22 and Pivot (88).

66	77	22	11	88 (Pivot)	99

- As the exchange took place so, change the direction to Left to Right (i.e. choose opposite direction).
- Now, between the exchanged elements 22 and Pivot (88) find the first greatest element than the Pivot (88).
- No element found, that means the Pivot is at proper position.
- Due to this two sub-lists are formed (i.e. before pivot and after pivot) as follows:

66	77	22	11	88 (Pivot)	99
1st sub-list					2nd sub-list

- Repeat the above process for 1st and 2nd sub-list.
- As 2nd sub-list contains only one element so, no need to sort 2nd sub-list.
- Now apply QuickSort to 1st sub-list
- Consider the 1st element as Pivot element

66 (Pivot)	77	22	11	88	99
1st sub-list				X	X

- Now, beginning from last element 11 scan from Right to Left comparing each number with Pivot (66) and stoppin first number less than Pivot (66).
- Now, 11 is smaller than Pivot (66) so, exchange 11 and Pivot (66)

11	77	22	66 (Pivot)	88	99
1st sub-list				X	X

- As the exchange took place so, change the direction to Left to right (i.e. choose opposite direction).
- Now, between the exchanged elements (i.e. 11 to Pivot (66)) find the find the first greater element than Pivot (66).
- It is 77, exchange 77 and Pivot (66)

11	66 (Pivot)	22	77	88	99
1st sub-list				X	X

- As the exchange took place so, change the direction to right to left (i.e. choose opposite direction).
- Now, between the exchanged elements (i.e. 77 to Pivot (66)) find the find the first smaller element than Pivot (66).
- It is 22, exchange 22 and Pivot (66)

11	22	66 (Pivot)	77	88	99
1st sub-list				X	X

- As the exchange took place so, change the direction to Left to right (i.e. choose opposite direction).
- Now, between the exchanged elements (i.e. 22 to Pivot (66)) find the find the first greater element than Pivot (66).
- No element found, that means the Pivot element is placed at its proper position.
- Due to this two sub-lists are formed (i.e. before pivot and after pivot) as follows:

11	22	66 (Pivot)	77	88	99
1st sub-list			2nd sub-list	X	X

- Repeat the above process for 1st and 2nd sub-list.
- As 2nd sub-list contains only one element so, no need to sort 2nd sub-list.
- Now apply QuickSort to 1st sub-list
- Consider the 1st element as Pivot element

11 (Pivot)	22	66	77	88	99
1st sub-list		X	X	X	X

- Now, beginning from last element 22 scan from Right to Left comparing each number with Pivot (11) and stopping a first number less than Pivot (11).
- No element found, that means the Pivot element is placed at its proper position.
- Due to this one sub-list is formed as follows:

11 (Pivot)	22	66	77	88	99
X	1st sub-list	X	X	X	X

- As the sub-list contains only one element so, no need to sort it.
- The sorted array is as follows:

11	22	66	77	88	99

55

UNIT – III: STACKS AND QUEUES

STACK:

- A stack is a linear list of elements in which an element can be inserted or deleted only at one end called as **TOP** of the stack.
- The stack works on the principle First In Last Out (FILO) / Last-In-First-Out (LIFO).
- The process of inserting a new element in stack is called as **PUSH**.
- The process of deleting an element from the stack is called as **POP.**

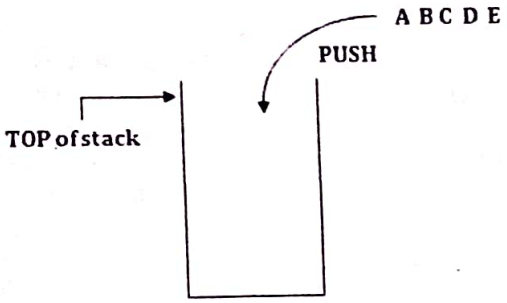


Fig: An empty stack

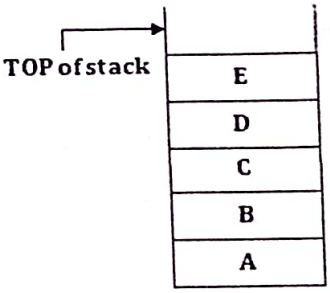


Fig: State of stack after PUSH operations

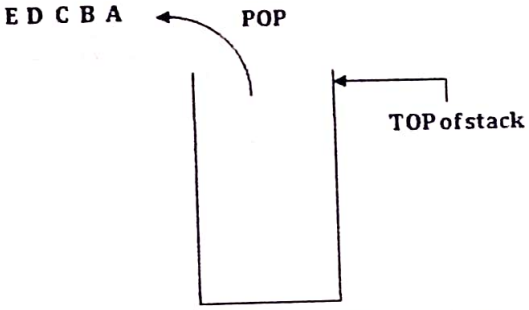


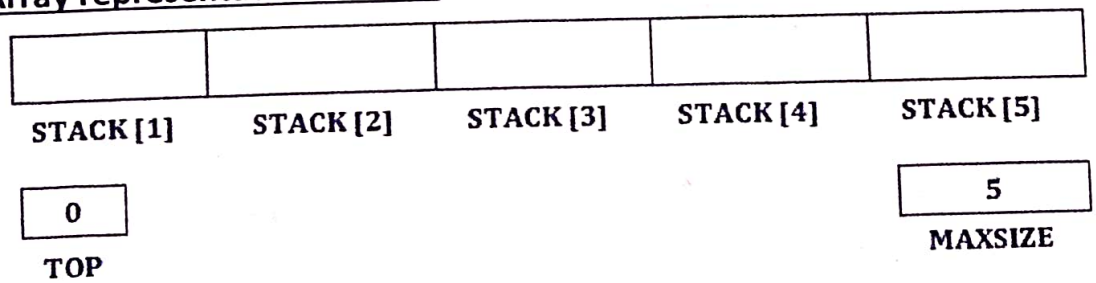
Fig: State of stack after POP operations

Stack Conditions:

1. **Overflow: (Stack Full / Stack Overflow)**
   If we want to insert a new data-item into a data structure but there is no available space i.e. free storage memory list is empty this situation is called as overflow.

2. **Underflow: (Stack Empty / Stack Underflow)**
   If we want to delete a data-item from a data structure that is already empty, then this situation is called as underflow.

- **Array representation of stack**



1. Algorithm for PUSH an item on to the stack

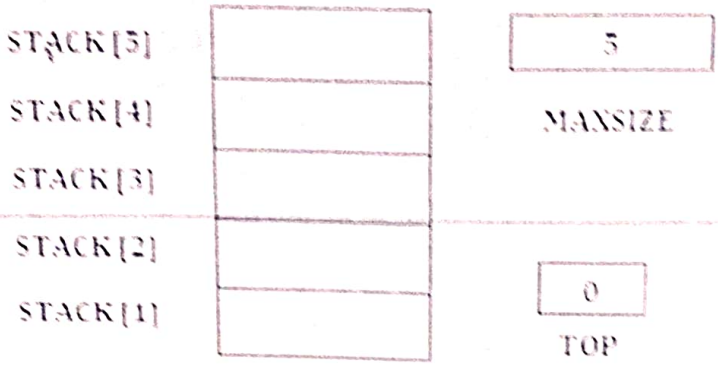


Fig : Array representation of stack

- **Where**

Variable	Description
STACK	It is an array used to represent the stack
MAXSIZE	It indicates the Maximum capacity of the STACK (i.e. array)
TOP	▪ It is an integer variable which contains the index number of the top most element in the stack. ▪ If the stack is Empty then TOP will be 0. (UNDERFLOW) ▪ If the STACK is full then TOP will be MAXSIZE (OVERFLOW)
ITEM	It is a variable which contains the value to push in the stack.

```
1. If TOP = MAXSIZE, Then:
2.        Write: 'Overflow'
3. Else:
4.        If TOP < MAXSIZE, Then:
5.                Set TOP := TOP + 1
6.                Set STACK[TOP] := ITEM
          [End of Step-4 If]
   [End of Step-1 If]
7. Exit
```

2. Algorithm for POP an item from the stack

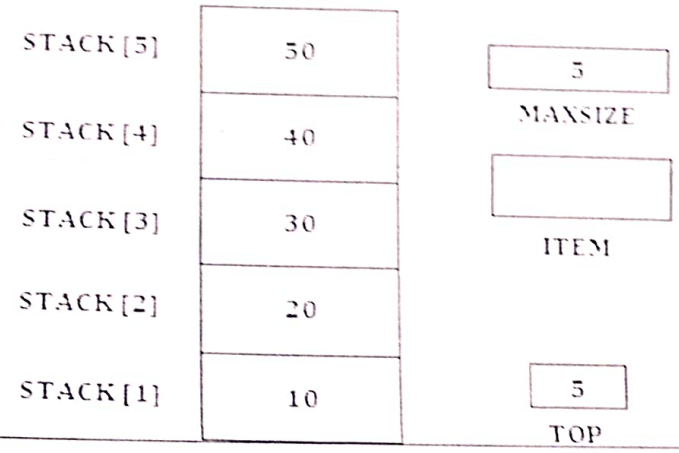


Fig : Array representation of stack (STACK is full)

Where

ariable	Description
STACK	It is an array used to represent the stack
TOP	It is an integer variable which contains the index number of top most element in the stack
ITEM	It is a variable to store the deleted/popped element from the stack

1. If TOP <= 0, Then:
2. Write: 'Underflow'
3. Else:
4. ITEM := STACK[TOP]
5. Set TOP := TOP -1

[End of Step-1 If]

6. Exit

Program:

```
#include<stdio.h>
#define MAXSIZE 5
main()
{
        int STACK[MAXSIZE], TOP = -1, ITEM;
        int x, i;

        do
        {
                printf("\n\n***STACK OPERATIONS***\n");
                printf("Options:\n");
                printf("1. PUSH\n");
                printf("2. POP\n");
                printf("3. VIEW STACK (TRAVERSE)\n");
                printf("4. EXIT\n");
                printf("Enter valid input: ");
                scanf("%d", &x);

                switch(x)
                {
                        case 1: //push
                                        if(TOP == (MAXSIZE-1))
                                        printf("Overflow..\n");
                                        else
                                        {
                                                if(TOP < (MAXSIZE-1))
                                                {
                                                        printf("\nEnter a value to push: ");
                                                        scanf("%d", &ITEM);
                                                        TOP = TOP + 1;
                                                        STACK[TOP] = ITEM;
                                                        printf("Values pushed..\n");
                                                }
                                        }
                                        break;
                        case 2: //pop
                                        if(TOP < 0)
                                        printf("Underflow..\n");
```

```
                else
                {
                        ITEM = STACK[TOP];
                        TOP = TOP - 1;
                        printf("Popped value = %d\n", ITEM);
                }
                break;
        case 3: //view stack
                printf("\nSTACK: \n\n");
                if(TOP == -1)
                printf("Stack is empty..\n");
                else
                {
                        for(i = TOP; i >= 0; i--)
                        {
                                if(i == TOP)
                                printf("%d ---->TOP\n", STACK[i]);
                                else
                                printf("%d\n", STACK[i]);
                        }
                }
                break;
        case 4: break;
        default: printf("Invalid input..\n");
    }
  }
  while(x != 4);
}
```

Applications of Stack:

Precedence order:

Operators	Symbols
Parenthesis	{ }, ( ), [ ]
Exponential notation	^
Multiplication and Division	*, /
Addition and Subtraction	+, -

Polish notation / Prefix notation

- It is a mathematical notation in which operators are placed before their operands.
- **Example:** +AB

Reverse Polish notation / Postfix notation

- It is a mathematical notation in which operators are placed after their operands.
- **Example:** AB+

nversion of infix to postfix

gorithm of Infix to Postfix

- Suppose Q is an arithmetic expression written in infix notation.
- This algorithm finds the equivalent postfix expression P.

Step 1: Push left (on stack and add right) to the end of Q.
Step 2: Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the stack is empty.
Step 3: If operand is encountered, add it to P.
Step 4: If left (is encountered, push it to STACK.
Step 5: If operator is encounter, then:

a) Repetitively pop from STACK and add to P each operator which has the same (i.e. equal) precedence or higher precedence than operator
b) Add the new operator to STACK
[End of if structure]

Step 6: If right parenthesis) is encountered, then:

a) Repetitively pop from stack and add to P each operator until left (is encountered.
b) Remove the left ([Do not add the left parenthesis to P.]
[End of if structure]
[End of step 2 loop]

Step 7: Exit

Example:
Consider the following arithmetic infix expression Q:
Q: A + (B * C – (D / E ^ F) * G) * H
Convert it to postfix.
Solution:
First push (onto STACK and then add) to end of the Q.

A	+	(	B	*	C	-	(	D	/	E	^	F	)	*	G	)	*	H	)
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Sr. No.	Symbol Scanned	STACK	Expression P
		(	
1	A	(	A
2	+	(+	A
3	(	(+ (	A
4	B	(+ (	A B
5	*	(+ (*	A B
6	C	(+ (*	A B C
7	-	(+ (-	A B C *
8	(	(+ (- (	A B C *
9	D	(+ (- (	A B C * D
10	/	(+ (- (/	A B C * D
11	E	(+ (- (/	A B C * D E
12	^	(+ (- (/ ^	A B C * D E
13	F	(+ (- (/ ^	A B C * D E F
14	)	(+ (-	A B C * D E F ^ /
15	*	(+ (- *	A B C * D E F ^ /
16	G	(+ (- *	A B C * D E F ^ / G
17	)	(+	A B C * D E F ^ / G * -
18	*	(+ *	A B C * D E F ^ / G * -
19	H	(+ *	A B C * D E F ^ / G * - H
20	)		A B C * D E F ^ / G * - H * +

The required postfix expression P = A B C * D E F ^ / G * - H * +

Example-2:

P * Q ^ R - S / T + (U / V)

Convert it to postfix

Solution:

P	*	Q	^	R	-	S	/	T	+	(	U	/	V	)	)
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Sr. No.	Symbol Scanned	STACK	Expression P
		(	
1	P	(	P
2	*	(*	P
3	Q	(*	PQ
4	^	(* ^	PQ
5	R	(* ^	PQR
6	-	(-	PQR^*
7	S	(-	PQR^*S
8	/	(- /	PQR^*S
9	T	(- /	PQR^*ST
10	+	(+	PQR^*ST/-
11	(	(+ (	PQR^*ST/-
12	U	(+ (	PQR^*ST/-U
13	/	(+ (/	PQR^*ST/-U
14	V	(+ (/	PQR^*ST/-UV
15	)	(+	PQR^*ST/-UV/
16	)	STACK EMPTY	PQR^*ST/-UV/+

Evaluation of a Postfix Expression:

- The following algorithm uses STACK to hold operand and evaluate P
- Suppose P is an arithmetic expression written in postfix notation.
- VALUE indicates the final result of the postfix expression.

Step 1: Add a right parenthesis) at the end of P.

Step 2: Scan P from left to right and repeat steps 3 and 4 for each element of P until right parenthesis) is encountered .

Step 3: If an operand is encountered, push it to STACK

Step 4: If an operator is encountered, then

a) Remove top two elements of STACK where A is the top element and B is the next-to-top element.

b) Evaluate B operator A

c) Place the result on stack

[End of if structure]

[End of Step-2 loop]

Step 5: Set VALUE equal to the top element of the STACK

Step 6: Exit

Example:

Consider the following arithmetic expression P written in postfix notation:

P: 5, 6, 2, +, *, 12, 4, /, -

Solution:

5	6	2	+	*	12	4	/	-	)

Patil Sir

1	2	3	4	5	6	7	8	9	10

Sr. No.	Symbol Scanned	STACK
1	5	5
2	6	5, 6
3	2	5, 6, 2
4	+	5, 8
5	*	40
6	12	40, 12
7	4	40, 12, 4
8	/	40, 3
9	-	37
10	)	

Conversion from infix to prefix

Algorithm of Infix to Prefix

- Q is infix expression
- P is prefix expression

Step 1: Push right parenthesis ")" onto STACK, and add left parenthesis "(" to start of the Q
Step 2: Scan Q from right to left and repeat step 3 to 6 for each element of Q until the STACK is empty
Step 3: If an operand is encountered add it to P
Step 4: If a right parenthesis is encountered push it onto STACK
Step 5: If an operator is encountered then:
a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has same or higher precedence than the operator.
b) Add operator to STACK
Step 6: If left parenthesis is encountered then
a) Repeatedly pop from the STACK and add to P each operator on top of stack until a right parenthesis is encountered
b) Remove the left parenthesis
Step 7: Reverse the P.
Step 8: Exit

Example:
Consider the following arithmetic infix expression Q.
Q: (A + B) * C
Convert it to prefix
Solution:
Push right parenthesis) to STACK and left parenthesis (to the start of Q

(	(	A	+	B	)	*	C
8	7	6	5	4	3	2	1

Sr. No.	Symbol Scanned	STACK	Prefix
		)	
1	C	)	C
2	*	)*	C
3	)	)*)	C
4	B	)*)	C B
5	+	)*)+	C B
6	A	)*)+	C B A
7	(		C B A + *

8	(		
			Reverse: * + A B C Required Prefix expression

Example:
Consider the following arithmetic infix expression Q.
Q: A + (B * C – (D / E ^ F) * G) * H
Convert it to prefix.
Solution:
Push right parenthesis) to STACK and left parenthesis (to the start of Q
(A + (B * C – (D / E ^ F) * G) * H

(	A	+	(	B	*	C	-	(	D	/	E	^	F	)	*	G	)	*	H
20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Sr. No.	Symbol Scanned	STACK	Postfix
		)	
1	H	)	H
2	*	)*	H
3	)	)*)	H
4	G	)*)	H G
5	*	)*)*	H G
6	)	)*)*)	H G
7	F	)*)*)	H G F
8	^	)*)*)^	H G F
9	E	)*)*)^	H G F E
10	/	)*)*)/	H G F E ^
11	D	)*)*)/	H G F E ^ D
12	(	)*)*	H G F E ^ D /
13	-	)*)-	H G F E ^ D / *
14	C	)*)-	H G F E ^ D / * C
15	*	)*)-*	H G F E ^ D / * C
16	B	)*)-*	H G F E ^ D / * C B
17	(	)*	H G F E ^ D / * C B * -
18	+	)+	H G F E ^ D / * C B * - *
19	A	)+	H G F E ^ D / * C B * - * A
20	(		H G F E ^ D / * C B * - * A +
			Reverse: + A * - * B C * / D ^ E F G H Required Prefix

Evaluation of Prefix expression

- The following algorithm uses STACK to hold operand and evaluate P
- Suppose P is an arithmetic expression written in prefix notation.
- VALUE is the final result of the prefix expression

Step 1: Add left parenthesis (at the start of the prefix expression P
Step 2: Scan P from right to left and repeat steps 3 and 4 for each element of P until left parenthesis (is encountered.
Step 3: If an operand is encountered, push it to STACK
Step 4: If an operator is encountered, then
a) Remove top two elements of STACK where A is the top element and B is the next-to-top element.
b) Evaluate A operator B
c) Place the result on stack
[End of if structure]
[End of step 2 loop]

Patil Sir

Step 5: Set VALUE equal to the top element of the STACK
Step 6: Exit

Example:
* + 6 9 - 3 1
Solution:
Add left parenthesis to the start of the prefix expression

(	*	+	6	9	-	3	1
8	7	6	5	4	3	2	1

Sr. No.	Symbol Scanned	STACK
1	1	1,
2	3	1, 3
3	-	2
4	9	2, 9
5	6	2, 9, 6
6	+	2, 15
7	*	30
8	(	

QUEUE:

- A queue is a linear list of elements in which deletion can takes place only at one end called as FRONT and insertion can takes place only at other end called as REAR.
- The queue data structure works on the principle First-In-First-Out (FIFO).

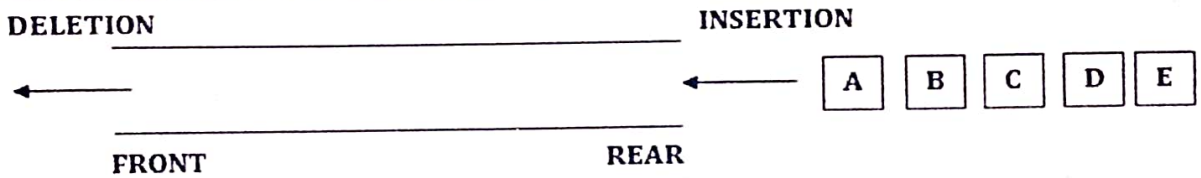


Fig : Structure of queue when it is empty

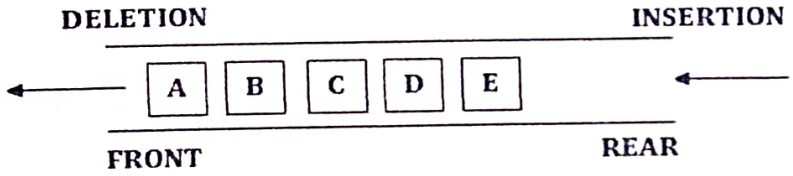


Fig : Structure of queue after inserting items

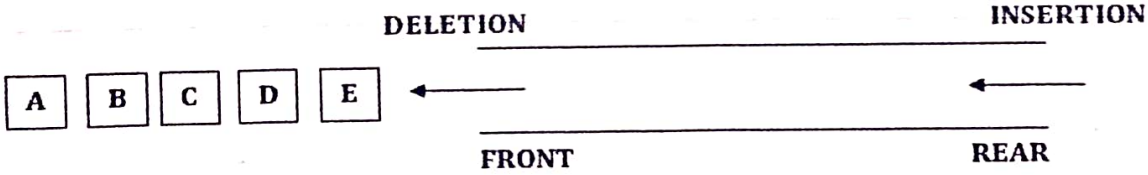


Fig : Sequence of item removal

Patil Sir

Types of queue:

- Linear Queue
- Circular Queue
- Priority Queue

Linear Queue:

- A linear queue is just like an array.
- In linear queue the elements are arranged in sequential manner such that the FRONT position is always less than or equal to the REAR position.
- Whenever an element is inserted in queue then the REAR pointer is incremented and whenever an element is removed from the queue the FRONT pointer is incremented.
- A queue can be implemented by using arrays as well as by using linked list.

Queue Conditions:

1. **Overflow: (Queue Full / Queue Overflow)**
   If we want to insert a new data-item into a data structure but there is no available space i.e. free storage memory list is empty this situation is called as overflow.

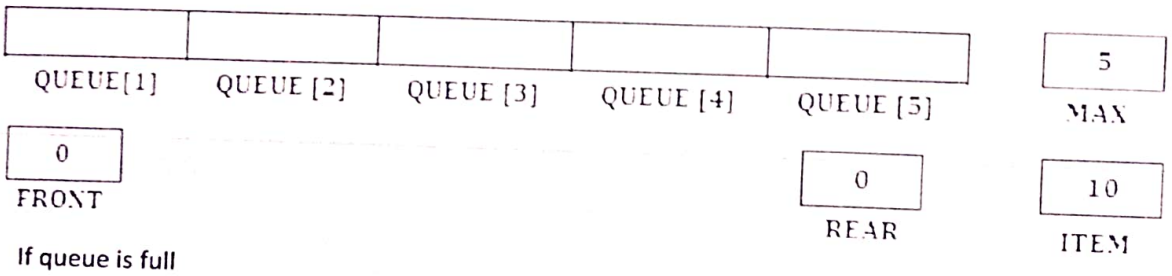
2. **Underflow: (Queue Empty / Queue Underflow)**
   If we want to delete a data-item from a data structure that is already empty, then this situation is called as underflow.

Array representation of linear queue:

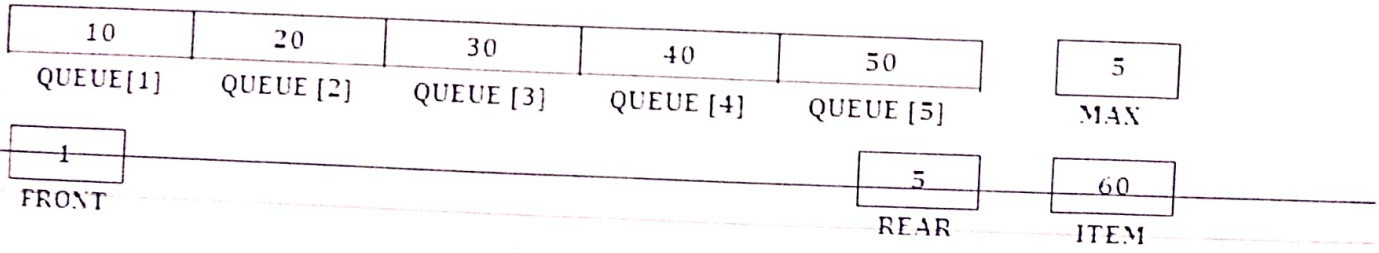
1. Algorithm to insert a new item in the linear queue (by using array)

Variable	Description
QUEUE	It is an array to implement the queue
MAX	It indicates the Maximum capacity of QUEUE array
FRONT	It is a variable used to store index number of first element in the QUEUE If QUEUE is empty then FRONT = 0 and REAR = 0
REAR	It is a variable used to store index number of last element in the QUEUE If QUEUE is empty then FRONT = 0 and REAR = 0
ITEM	It is a variable used to store the value to insert.

- **Possibilities:**
  1) If queue is empty

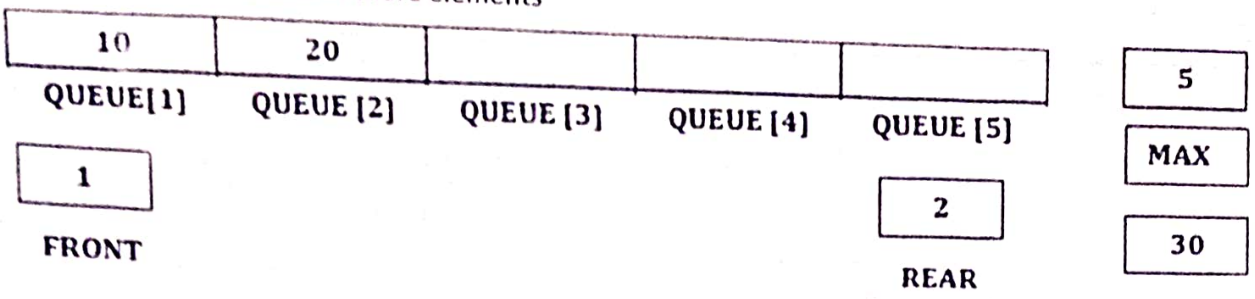


  2) If queue is full



Patil Sir

3) If queue contains one or more elements



QINSERT

```
1. If FRONT >= 1 and REAR = MAX, Then:
2.       Write: 'OVERFLOW'
3. Else:
       [If queue is empty]
4.     If FRONT = 0 And REAR = 0, Then:
5.             Set FRONT := 1
6.             Set REAR := 1
7.     Else:
8.             Set REAR := REAR + 1
       [End of Step 4 If]

9.     Set QUEUE[REAR] := ITEM
10.    Write : 'Item Inserted'
   [End of Step 1 If]
11. Exit
```

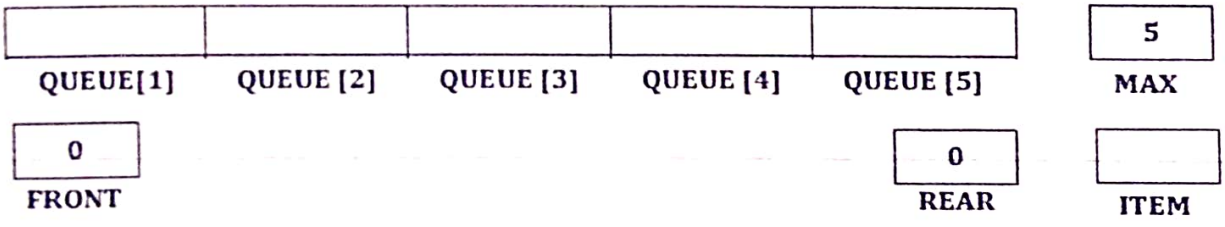
2. Algorithm to delete an item from a linear queue (by using array)

- Where:

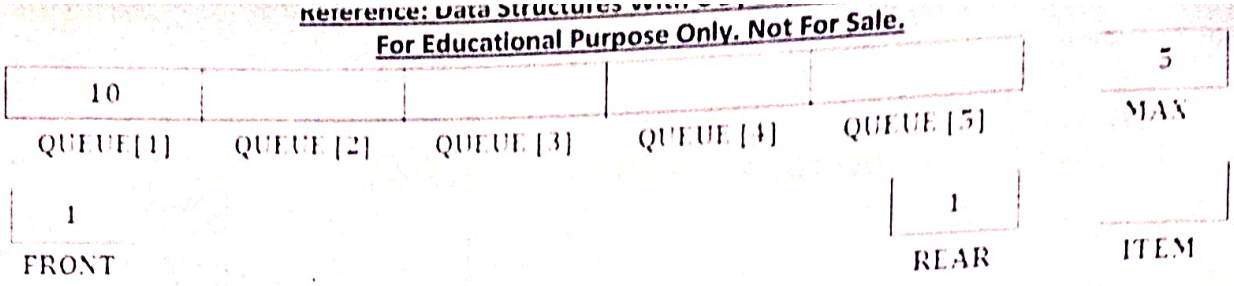
Variable	Description
QUEUE	It is an array used to implement the queue
MAX	It indicates the Maximum capacity of QUEUE
FRONT	It is a variable used to store the index of first element in the QUEUE
REAR	It is a variable used to store the index of last element in the QUEUE
ITEM	It is a variable used to store the deleted item

- Possibilities

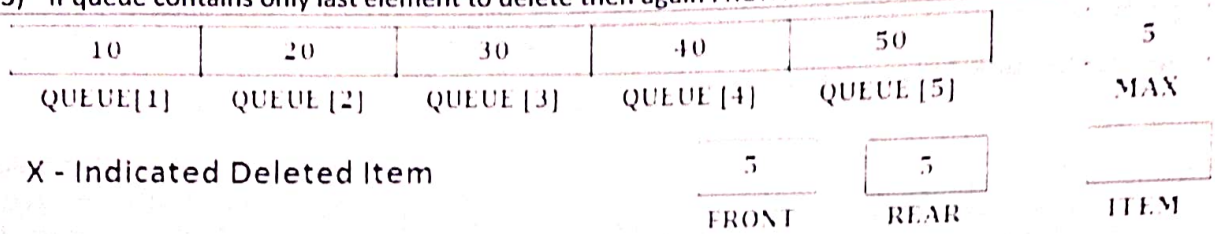
1) If queue is empty



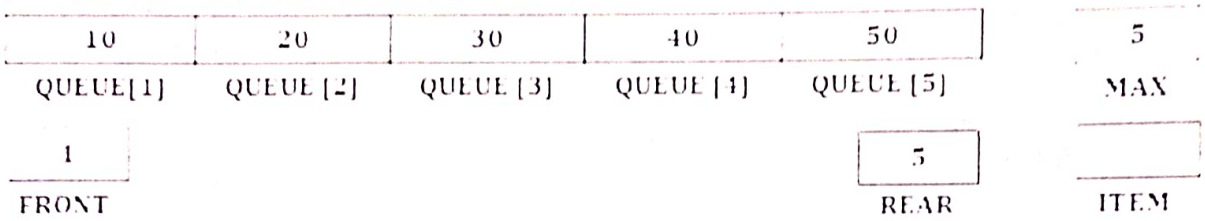
2) If queue contains only one element to delete, then FRONT = REAR



3) If queue contains only last element to delete then again FRONT = REAR



4) If queue contains more than one elements



QDELETE

[Checking for empty queue]
1. If FRONT = 0, Then:
2. Write: 'UNDERFLOW'
3. Else:
4. Set ITEM := QUEUE[FRONT]
[if queue contains only one element 1st or middle or last element]
5. If FRONT = REAR, Then:
6. Set FRONT := 0
7. Set REAR := 0
8. Else:
9. Set FRONT := FRONT + 1
[End of Step 5 If]

10. Write: 'Deleted Item ', ITEM
[End of Step 1 If]
11. Exit

Program:

```
#include<stdio.h>
#define MAXSIZE 5
main()
{
        int n, i;
        int QUEUE[MAXSIZE], FRONT = -1, REAR = -1, ITEM;
        do
        {
```

```
printf("\n*** QUEUE OPERATIONS ***\n");
printf("\t 1: QINSERT\n");
printf("\t 2: QDELETE\n");
printf("\t 3: SHOW QUEUE\n");
printf("\t 4: EXIT\n");
printf("Enter valid option: ");
scanf("%d", &n);

switch(n)
{
        case 1: //QINSERT
                        if(REAR >= (MAXSIZE-1))
                        printf("\nOverflow: Queue is full\n");
                        else
                        {
                                // checking for empty queue
                                if(FRONT == -1 && REAR == -1)
                                {
                                        FRONT = 0;
                                        REAR = 0;
                                }
                                else
                                {
                                        // if queue contains one or more elements
                                        REAR = REAR + 1;
                                }
                                printf("\nEnter a number to insert: ");
                                scanf("%d", &ITEM);
                                QUEUE[REAR] = ITEM;
                                printf("\nNumber inserted..\n");
                        }
                        break;
        case 2: // QDELETE
                        // Checking for empty queue
                        if(FRONT == -1 && REAR == -1)
                        printf("\nUnderflow: Queue is empty..\n");
                        else
                        {
                                ITEM = QUEUE[FRONT];
                                printf("\nDeleted number = %d\n", ITEM);
                                // checking for single element
                                if(FRONT == REAR)
                                {
                                        FRONT = -1;
                                        REAR = -1;
                                }
                                else
                                FRONT = FRONT + 1;
                        }
                        break;
        case 3: // QUEUE DISPLAY
                        if(FRONT == -1 && REAR == -1)
                        printf("\nQueue is empty..\n");
                        else
```

```
                {
                        printf("\nQUEUE:\n");
                        for(i = FRONT; i <= REAR; i++)
                        printf("%d ", QUEUE[i]);
                        printf("\n");
                }
                break;
        case 4: printf("\nPress any key to exit..");
                break;
        default: printf("\nInvalid input...");
    }
}
while(n != 4);
```

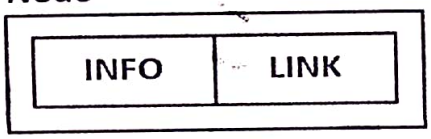
UNIT-IV: LINKED LIST

- The term list refers to a linear collection of data items.
- It is not necessary that the linear means consecutive or continuous.
- Every consecutive is linear but, every linear is not consecutive.
- A list can be implemented either by using array or linked list.
- The biggest disadvantage of array is, the array occupies a large block of memory and in most cases all the locations in an array may not come in use.
- Also, it is not possible to increase or decrease the size of an array at runtime (i.e. while program is executing) if required.
- Another disadvantage of array is that, it is also difficult & expensive to insert or delete an element from the array.
- Suppose we want to insert a new item in between two array elements then, we need to shift the remaining elements by one position, also in deletion of element we need to shift the remaining elements one position before. This process takes more time.
- So, to manage list in memory instead of array we can use the linked list.
- In linked it is easier to insert or delete elements in the list.
- Also, we can increase or decrease the size of linked list at runtime as per the requirement.

Structure and definition of linked list

- A linked list is a linear collection of data items where a data item (i.e. value) is stored in a node and the linear order is maintained by using pointers.
- In linked list each node is divided into two parts, the first part contains the information of an element (i.e. value) and the second part contains the address of the next node in the list.
- Here the part of node which contains value (i.e. information of data item) is called as **INFO or Information Field or Data Field** and the part of node which contains address of the next node is called as **LINK** or **Link Field** or **Next Pointer Field**.
- The last node in linked list contains a special value **NULL** in its Link part which indicates end of the linked list.

Node



Types of linked list

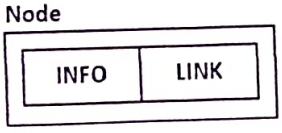
Basically, there are three types of linked list as follows.

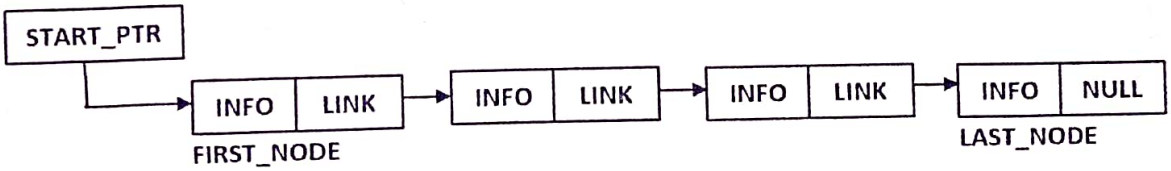
1. Singly linked list
2. Doubly linked list
3. Circular linked list

Singly linked list

In singly linked list each node contains the information about the data item and the address of the next node in memory.

Node





Patil Sir

Algorithm to traverse a Singly linked list

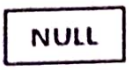
Possibilities:

1. If linked list is empty then START = NULL
2. If linked list is not empty

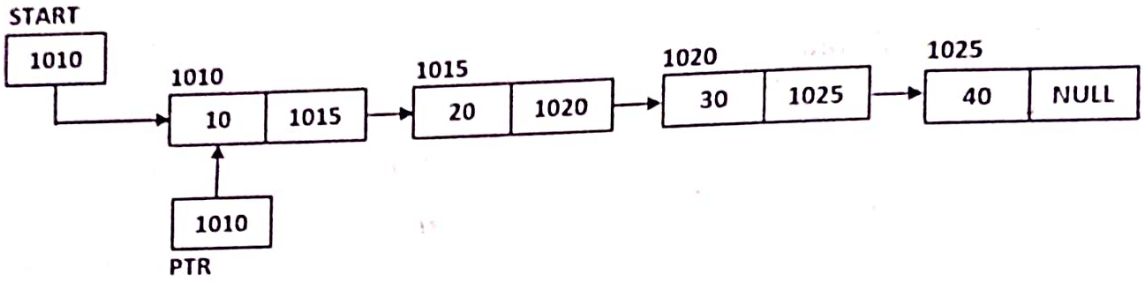
Variable	Description
START	It is a pointer which points to the first node in the linked list
PTR	It is a pointer which is used to traverse the linked list
INFO	It indicates the information part of the node
LINK	It indicates the NEXT node address within the linked list

Case 1: If linked list is empty

START



Case 2: If linked list is not empty.



1. Set PTR := START
2. Repeat Steps 3 and 4 while PTR != NULL
3. Print: INFO[PTR]
4. Set PTR := LINK[PTR]
   [End of Step 2 loop]
5. Exit

Searching in a Singly linked list:

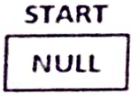
Possibilities:

1. Linked list is empty
2. Linked list is not empty

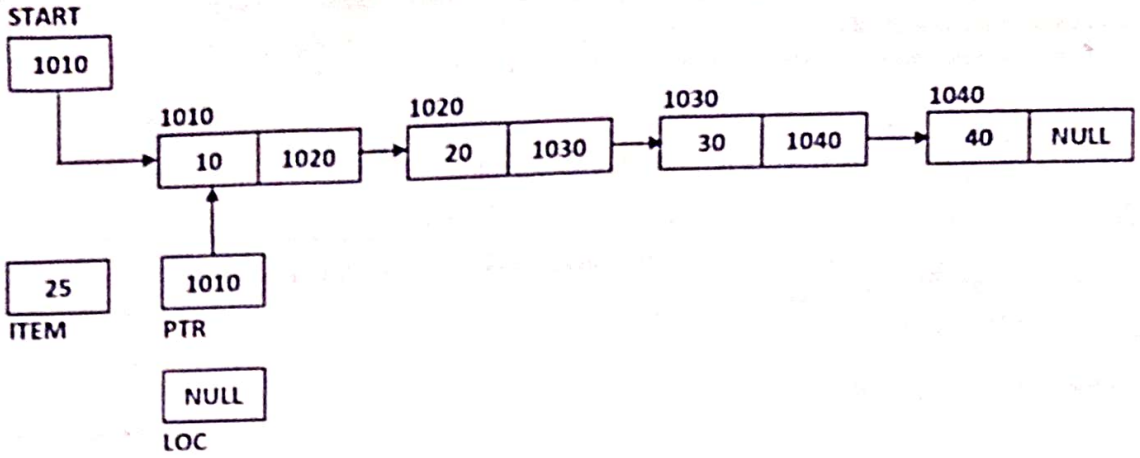
Variable	Description
START	It is a pointer which points to the first node in the linked list.
ITEM	It is a regular variable which contains the value to search.
LOC	It is a pointer which contains address of the node of ITEM, if found
PTR	It is a pointer used to traverse the linked list

Case 1: If linked list is empty

START



Case 2: If linked list is not empty.



1. Set PTR := START
2. Set LOC := NULL

3. Repeat while PTR != NULL:
4. If ITEM = INFO[PTR], Then:
5. Set LOC := PTR
6. Break
7. Else:
8. Set PTR := LINK[PTR]
   [End of Step-4 If]
   [End of Step-3 Loop]

9. If LOC = NULL, Then:
10. Print: Not Found
11. Else:
12. Print: Found
    [End Step-9 if]
13. Exit

Garbage Collection:

- When a value is deleted from a list then the memory occupied by the value becomes free.
- This freed space can be used in future.
- The operating system contains a program having name Garbage Collection.
- The garbage collection program collects the free memory onto the free-storage list.
- The operating system of a computer periodically runs the Garbage Collection program.
- Garbage collection usually works in two steps as follows:
  1. First the garbage collection program runs through the memory, tagging those cells which are currently in use.
  2. Secondly the garbage collection program runs through the memory collecting all untagged space onto free-storage list.
- The garbage collection take place when there is minimum amount of space left in the free-storage list.
- The garbage collection runs in the background and it is invisible to programmer.

Overflow:

Sometimes there is no available space (i.e. the free-storage list is empty) this situation is called as overflow (i.e. AVAIL = NULL).

Underflow:

Deleting from an empty data structure is called as underflow (i.e. START = NULL).

Variable
START
AVAIL

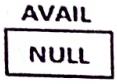
Inserting at the beginning of Singly linked list

Possibilities:

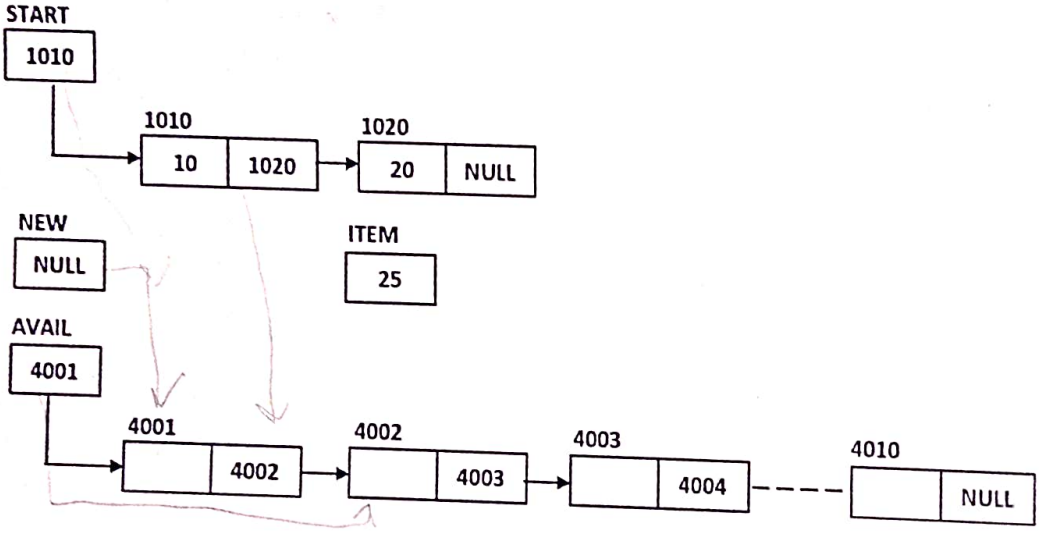
1. If the free memory is not available
2. If the free memory is available

Variable	Description
START	It is a pointer which points to the 1st node in the linked list.
NEW	It is a pointer which is used to point to the new node
ITEM	It is a regular variable which contains the value to insert in new node
AVAIL	It is a pointer which points to the 1st node in the available memory (i.e. free memory).
INFO	Information part of the node
LINK	Indicated the next node address

Case 1: If free memory is not available.



Case 2: If free memory is available



1. If AVAIL = NULL Then:
2. Write: OVERFLOW
3. Exit
4. Else:
5. Set NEW := AVAIL
6. Set AVAIL := LINK[AVAIL]
7. Set INFO[NEW] := ITEM
8. Set LINK[NEW] := START
9. Set START := NEW
   [End of Step-1 if]
10. Exit

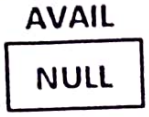
Inserting after a given node in Singly Linked List:

Possibilities:

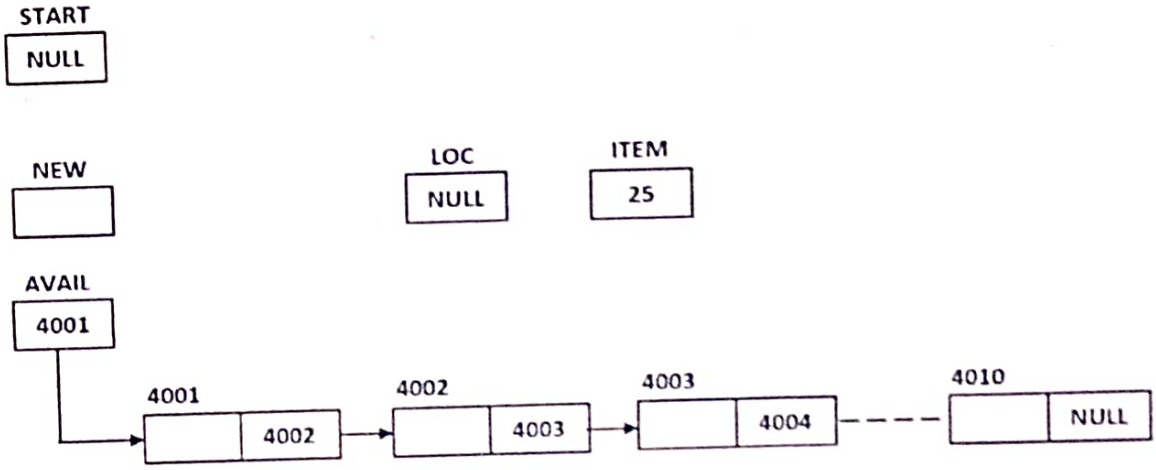
1. If free memory is not available
2. If linked list is empty
3. If linked list is not empty

Variable	Description
START	It is a pointer which points to the first node in the linked list
AVAIL	It is a pointer which points to the 1st node in the available memory (i.e. free memory).
LOC	• It is a pointer which points to the node after which the new node is inserted. • If Linked list is empty then LOC = NULL
NEW	It is a pointer which points to the new node
ITEM	It is a regular variable which contains the value to store in the INFO part of new node

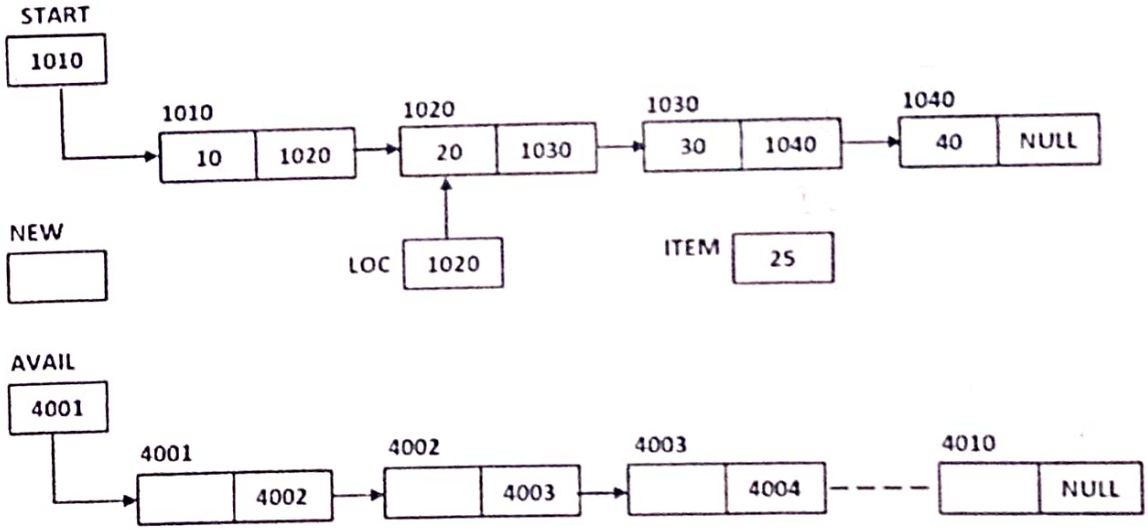
Case 1: If free memory is not available.



Case 2: If linked list is empty



Case 3: If linked list is not empty



1. If AVAIL = NULL, Then:
    a. Write: OVERFLOW
    b. Exit
2. Else:
    a. Set NEW := AVAIL
    b. Set AVAIL := LINK[AVAIL]

c. Set INFO[NEW] := ITEM

[Checking for empty linked list]
d. If LOC = NULL, Then:
    i. Set LINK[NEW] := START
    ii. Set START := NEW
e. Else:
    i. Set LINK[NEW] := LINK[LOC]
    ii. Set LINK[LOC] := NEW
[End of If]
[End of If]
3. Exit

Deleting the node with given Location in Singly Linked List:

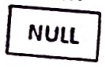
Possibilities:

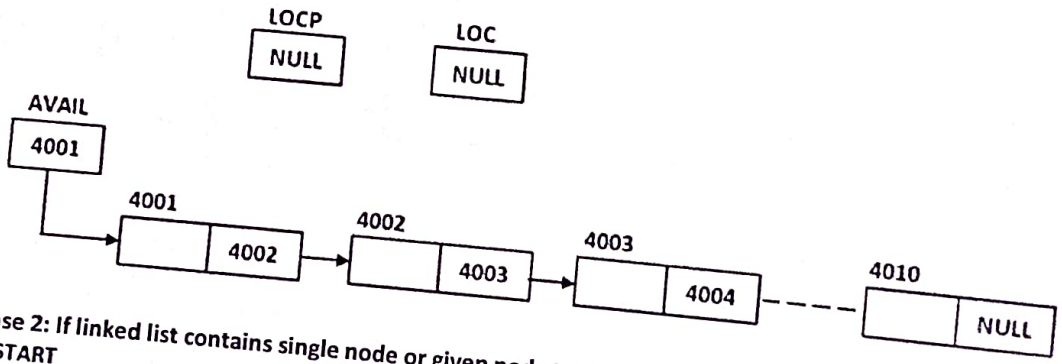
1. If the linked list is empty then START contains NULL and LOC = NULL and LOCP = NULL
2. If linked list contains single node or given node to delete is the first node, then there will be no previous node that means LOCP contains NULL.
3. If linked list contains multiple nodes.

Note:
In the following algorithm, the deleted node will added as first node in the available memory.

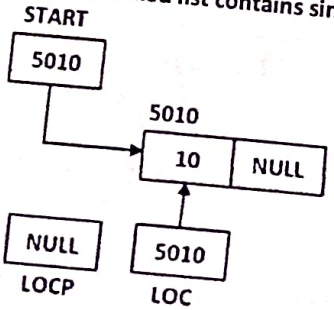
Variable	Description
START	It is a pointer which points to the first node in the linked list
AVAIL	It is a pointer which points to the first node in the available memory
LOC	It is a pointer which points to the node which we want to delete
LOCP	It is a pointer which points to the previous node (i.e. previous of LOC)

Case 1: If linked list is empty

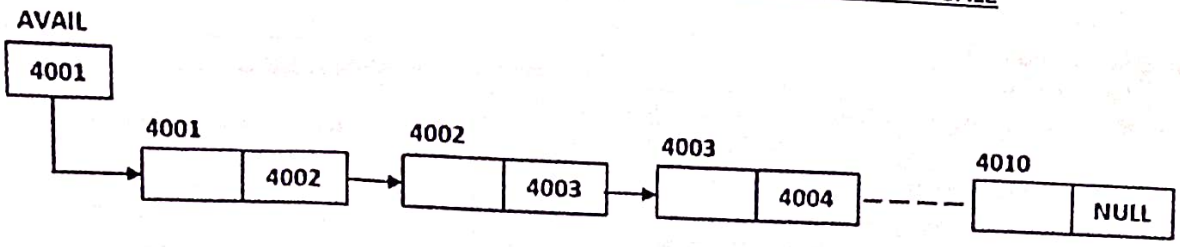




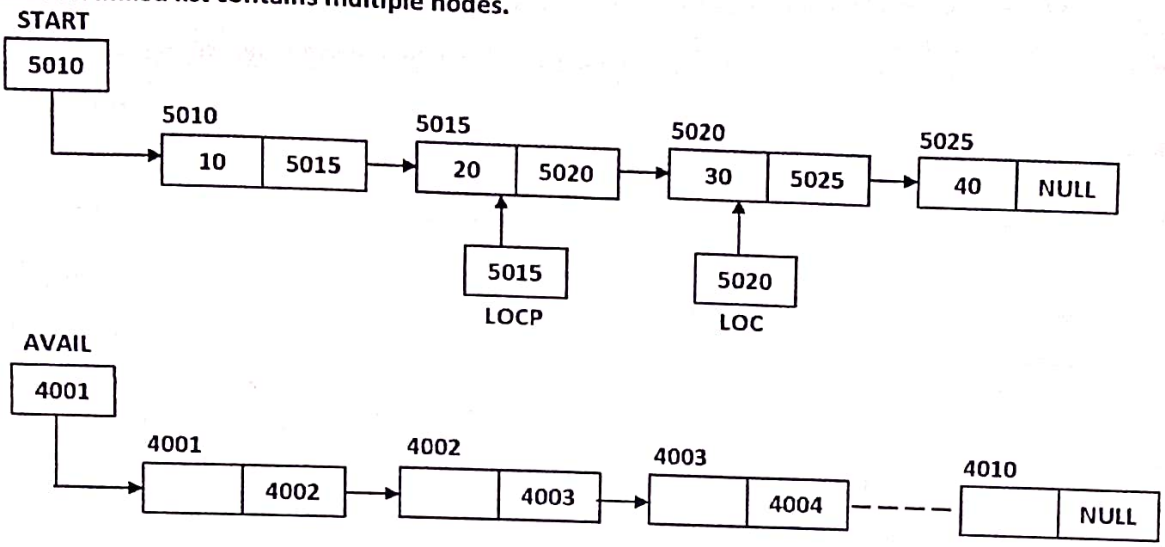
Case 2: If linked list contains single node or given node to delete is the first node, then there will be no previous node



Patil Sir



Case 3: If linked list contains multiple nodes.



[Checking for empty linked list]
1. If START = NULL, Then:
2. Write: Underflow
3. Exit
4. Else:
[if deleting 1st node or linked list contains only one node]
5. If LOCP = NULL, Then:
6. Set START := LINK[START]
[Deletes first Node]
7. Else:
8. Set LINK[LOCP] := LINK[LOC]
[End of If]

[Return deleted node to free memory]
9. Set LINK[LOC] := AVAIL
10. Set AVAIL := LOC
[End of If]
11. Exit

Deleting node with given item of information in Singly Linked List:

Note:

- The following algorithm deletes first occurrence if ITEM if found.
- The deleted node will be added as first node in the free (i.e. available) memory.

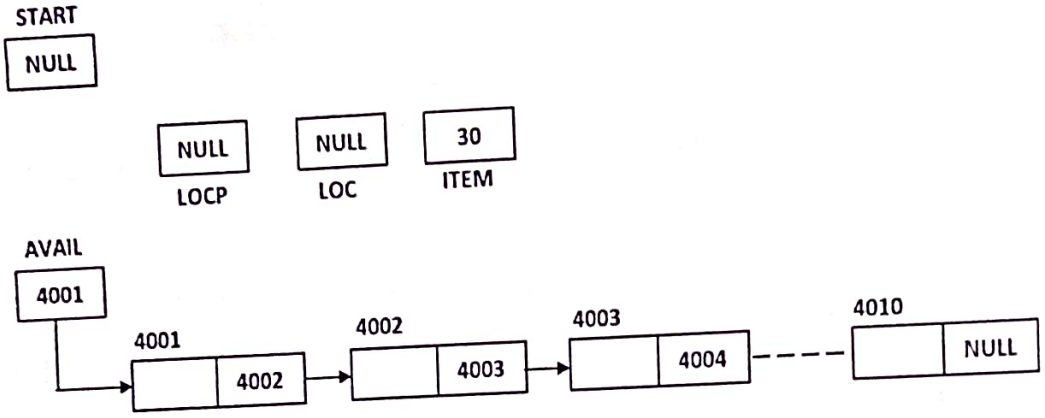
Possibilities:

1. If linked list is empty then START contain NULL, LOC = NULL and LOCP = NULL
2. If ITEM found in 1st node, then LOCP = NULL. And after deleting the node we need to update the value of START.
3. If ITEM found in the middle of linked list then we need to update the LINK of previous node.
4. If ITEM not found then LOC = NULL and LOCP = NULL

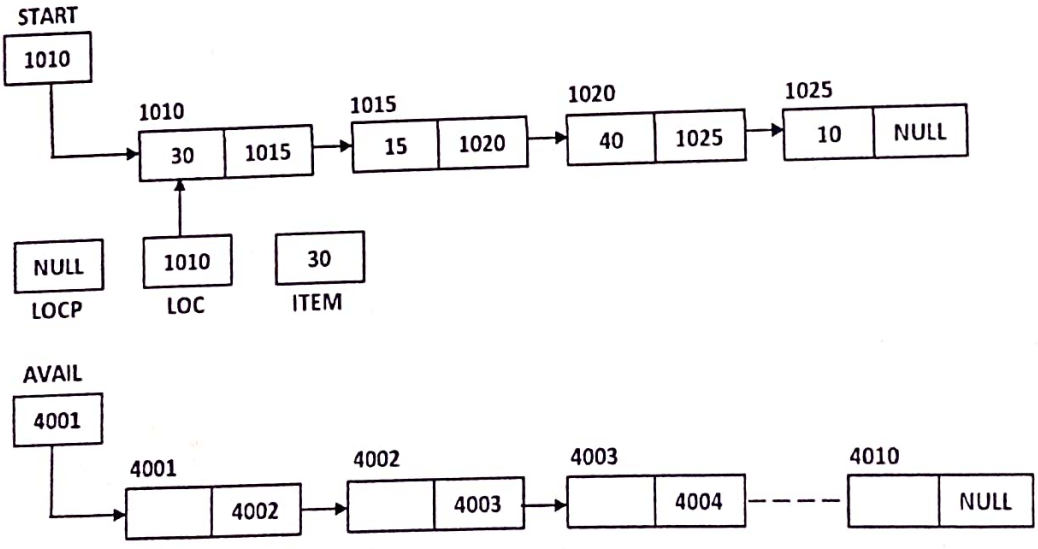
Procedure|function to search the ITEM:

Variable	Description
START	It is a pointer which points to the first node in the linked list.
AVAIL	It is a pointer which points to the first node in the available memory (i.e. Free Memory)
ITEM	It is a regular variable which contains the value which we want to search.
LOC	It is a pointer which points to the node of ITEM, if ITEM found.
LOCP	▪ It is a pointer which points to the previous node of LOC. ▪ If Item found at 1st node then LOCP = NULL
PTR	It is a pointer used to traverse the linked list
SAVE	It is a pointer used to store the address of previous node while traversing the linked list.

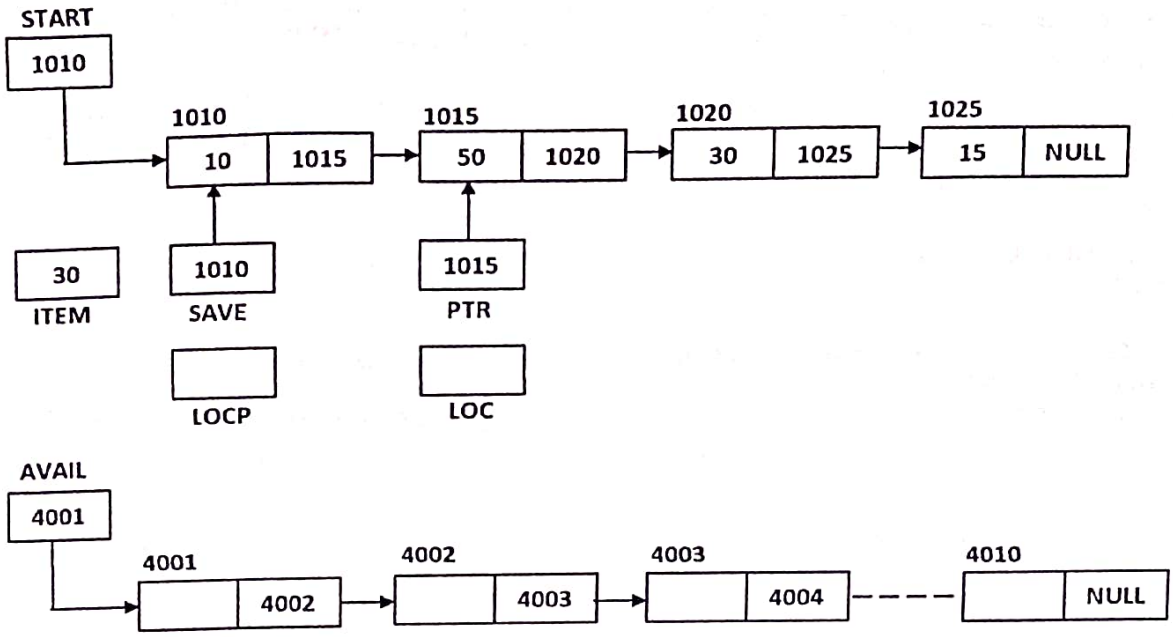
Case 1: If linked list is empty



Case 2: If ITEM found in 1st node, then LOCP = null



Case 3: If ITEM found in the middle of linked list



Procedure/function: FIND

```
1. If START = NULL, Then:
2.      Set LOC := NULL
3.      Set LOCP = NULL
4.      Return
5. Else:
        [Checking for the 1st node.]
6.      If INFO[START] = ITEM, Then:
7.              Set LOC := START
8.              Set LOCP := NULL
9.              Return
10.     Else:
                [Traversing from 2nd element up to last element to search the ITEM]
11.             Set SAVE := START
12.             Set PTR := LINK[START]

13.             Repeat While PTR != NULL:

14.                     If INFO[PTR] = ITEM, Then:
15.                             Set LOC := PTR
16.                             Set LOCP := SAVE
17.                             Return
18.                     Else:
19.                             Set SAVE := PTR
20.                             Set PTR := LINK[PTR]
                        [End of If]
                [End of loop]
21.             Set LOC := NULL and LOCP := NULL
        [End of If]
    [End of If]
22. Return
```

Delete Algorithm:

1. Call Find
2. If LOC = NULL, Then:
3. Write: ITEM not found
4. Exit
5. Else:
6. If LOCP = NULL Then: [If LOC == START Then:]
7. Set START := LINK[START]
   [Delete first node]
8. Else:
9. Set LINK[LOCP] := LINK[LOC]
   [End of If]

   [Return deleted node to the available memory]
10. Set LINK[LOC] := AVAIL
11. Set AVAIL := LOC
    [End of if]
12. Exit

Program: Write C program to perform the operations (Insert as first node, Delete, Traverse and Search) on Singly Linked List.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int INFO;
        struct node *LINK;
};

struct node *START = NULL;

void insert_first()
{
        int ITEM;
        struct node *NEW;

        printf("\nEnter a value to insert: ");
        scanf("%d", &ITEM);

        NEW = (struct node*) malloc(sizeof(struct node));

        if(NEW == NULL)
        {
                printf("\nOVERFLOW..\n");
                return;
        }
        else
        {
                NEW->INFO = ITEM;
                NEW->LINK = START;
                START = NEW;
                printf("Value/Node inserted..\n");
        }
}
```

Patil Sir

```
void insert_after()
{
        int ITEM;
        struct node *PTR, *LOC = NULL, *NEW;

        if(START == NULL)
        printf("Linked List is empty..\n");
        else
        {
                printf("Enter ITEM: ");
                scanf("%d", &ITEM);

                PTR = START;
                while(PTR != NULL)
                {
                        if(PTR->INFO == ITEM)
                        {
                                LOC = PTR;
                                break;
                        }
                        else
                        PTR = PTR->LINK;
                }

                if(LOC == NULL)
                printf("\nItem not found. Cannot inserted..\n");
                else
                {
                        printf("Enter value for INFO: ");
                        scanf("%d", &ITEM);

                        NEW = (struct node*)malloc(sizeof(struct node));
                        NEW->INFO = ITEM;
                        NEW->LINK = LOC->LINK;
                        LOC->LINK = NEW;
                        printf("Inserted..\n");
                }
        }
}

void delete_node()
{
        int ITEM;
        struct node *LOC = NULL, *LOCP = NULL, *PTR, *SAVE;

        if(START == NULL)
        {
                printf("UNDERFLOW..\n");
                return;
        }
        else
        {
                printf("\nEnter ITEM: ");
                scanf("%d", &ITEM);

                if(START->INFO == ITEM)
```

```
                {
                        LOCP = NULL;
                        LOC = START;
                }
                else
                {
                        SAVE = START;
                        PTR = START->LINK;

                        while(PTR != NULL)
                        {
                                if(PTR->INFO == ITEM)
                                {
                                        LOC = PTR;
                                        LOCP = SAVE;
                                        break;
                                }
                                else
                                {
                                        SAVE = PTR;
                                        PTR = PTR->LINK;
                                }
                        }
                }

                if(LOC == NULL)
                printf("Cannot delete. ITEM not found..\n");
                else
                {
                        if(LOCP == NULL)
                        START = START->LINK;
                        else
                        LOCP->LINK = LOC->LINK;

                        free(LOC);
                        printf("Deleted..\n");
                }
        }
}

void search()
{
        struct node *PTR, *LOC = NULL;
        int ITEM;
        printf("\nEnter ITEM to search: ");
        scanf("%d", &ITEM);

        PTR = START;

        while(PTR != NULL)
        {
                if(PTR->INFO == ITEM)
                {
                        LOC = PTR;
                        break;
                }
```

```
                else
                PTR = PTR->LINK;
        }

        if(LOC == NULL)
        printf("Not found..\n");
        else
        printf("Found..\n");
}

void traverse()
{
        struct node *PTR;

        if(START == NULL)
        printf("\nLinked List is empty..\n");
        else
        {
                PTR = START;
                printf("\nLinked List: ");

                while(PTR != NULL)
                {
                        printf("%d ", PTR->INFO);
                        PTR = PTR->LINK;
                }
                printf("\n");
        }
}

main()
{
        int option;

        do
        {
                printf("\n\n1: Insert as 1st node\n");
                printf("2: Insert after a node having given ITEM of INFO.\n");
                printf("3: Delete node having given ITEM of INFO.\n");
                printf("4: Search\n");
                printf("5. Traverse/Display\n");
                printf("6. Exit\n");
                printf("Enter valid option: ");
                scanf("%d", &option);

                switch(option)
                {
                        case 1: insert_first();
                                break;
                        case 2: insert_after();
                                break;
                        case 3: delete_node();
                                break;
                        case 4: search();
                                break;
                        case 5: traverse();
```

```
                    break;
                case 6: printf("Press any key to exit..\n");
                    break;
                default: printf("Invalid input..\n");
            }
        }
        while(option != 6);
}
```

Implementing Stack by using Linked-List

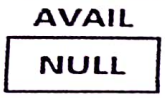
Algorithm to PUSH:

Possibilities:

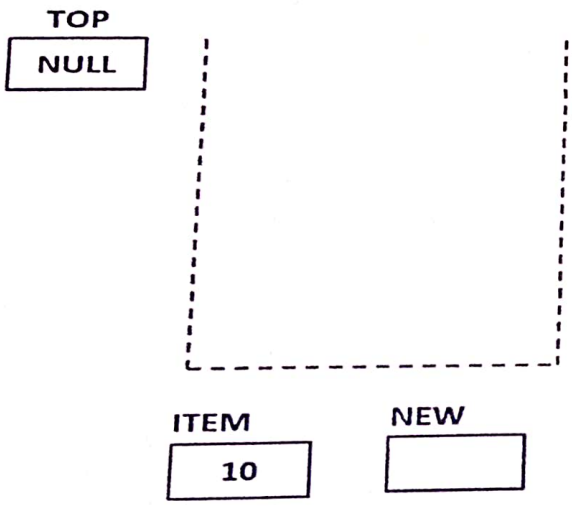
1. If free memory (i.e. available memory) is not available. OVERFLOW
2. If stack is empty then TOP = NULL
3. If stack contains one or more nodes.

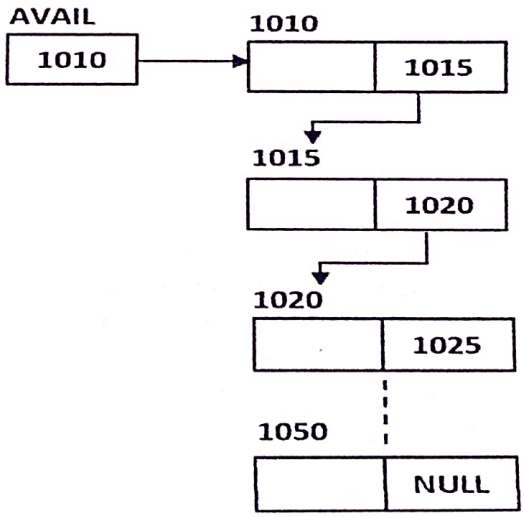
Variable	Description
TOP	It is a pointer which points to the topmost node in the stack.
ITEM	▪ It is a regular variable. ▪ It indicates the value to be inserted in the INFO part of new node.
AVAIL	It is a pointer which points to the 1st node in the available (i.e. free) memory.
NEW	It is a pointer which points to the new node

Case 1: If free memory (i.e. available memory) is not available. OVERFLOW

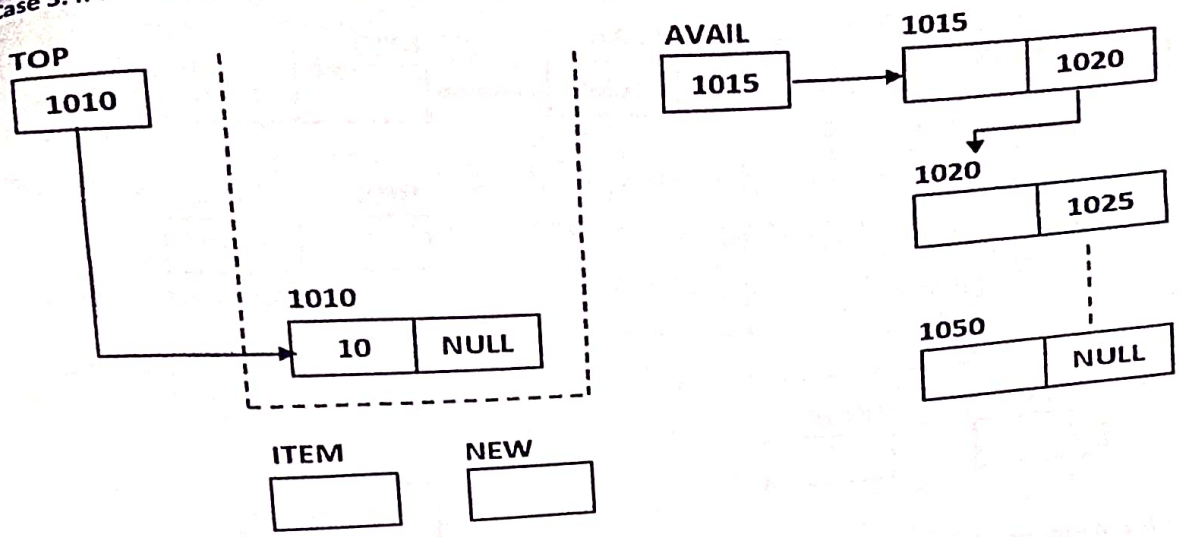


Case 2: If stack is empty then TOP = NULL





Case 3: If stack contains one or more nodes



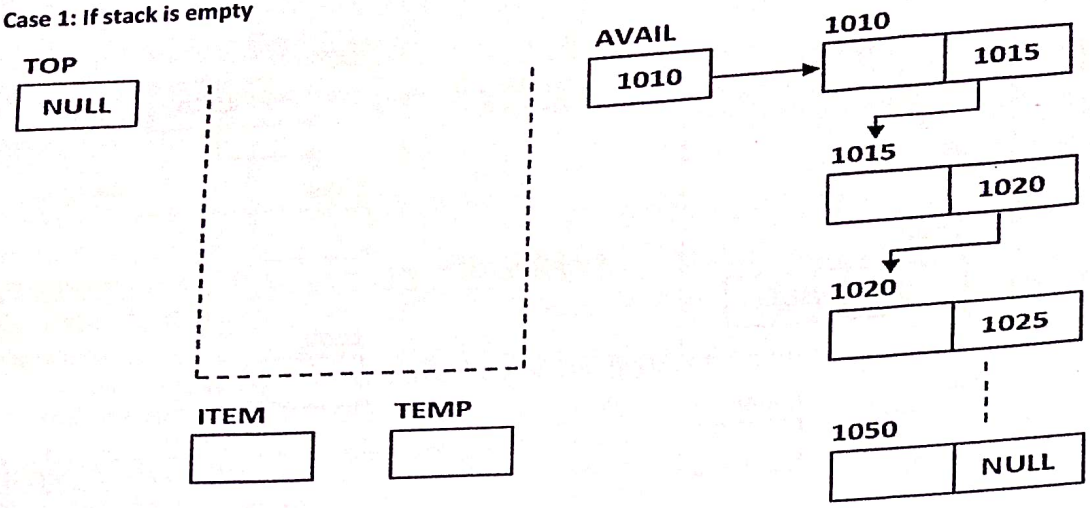
1. If AVAIL = NULL, Then:
2. Write: OVERFLOW
3. Exit
4. Else:
5. Set NEW := AVAIL
6. Set AVAIL := LINK[AVAIL]
7. Set INFO[NEW] := ITEM
8. Set LINK[NEW] := TOP
9. Set TOP := NEW
   [End of If]
10. Exit

Algorithm to POP:

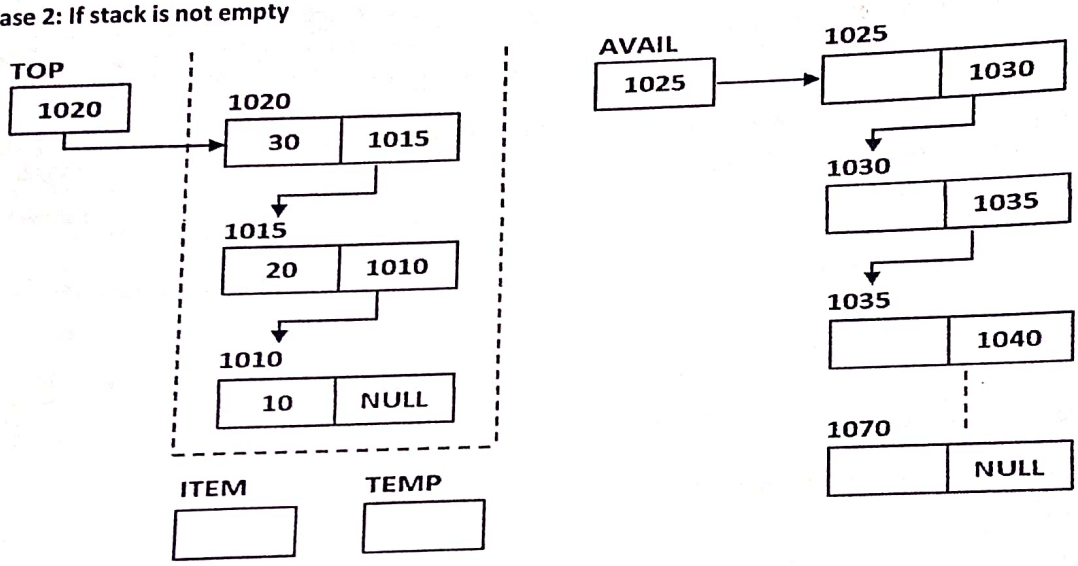
Note:
The popped node (i.e. deleted node will be added as first node in the free memory)
Possibilities:
1. If stack is empty (i.e. UNDERFLOW) then TOP = NULL
2. If stack is not empty

Variable	Description
TOP	It is a pointer which points to the topmost node in the stack.
ITEM	It is a regular variable. It indicates the deleted item
AVAIL	It is a pointer which points to the available (i.e. free) memory.
TEMP	It is a pointer which points to the deleted node.

Case 1: If stack is empty



Case 2: If stack is not empty



[If stack is empty]
1. If TOP = NULL, Then:
2. Write: UNDERFLOW
3. Exit
4. Else:
5. Set ITEM := INFO[TOP]
6. Set TEMP := TOP
7. Set TOP := LINK[TOP]

[Return the deleted node to free memory]
8. Set LINK[TEMP] := AVAIL
9. Set AVAIL := TEMP
[End of If]
10. Exit

Program: Write a program to design a stack using Linked List.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int INFO;
        struct node *LINK;
};

struct node *TOP = NULL;

void push()
{
        int ITEM;
        struct node *NEW = NULL;

        NEW = (struct node*) malloc(sizeof(struct node));

        if(NEW == NULL)
        {
                printf("\nOVERFLOW..\n");
                return;
        }
        else
        {
                printf("\nEnter a value to push: ");
                scanf("%d", &ITEM);

                NEW->INFO = ITEM;
                NEW->LINK = TOP;
                TOP = NEW;
                printf("\nValue pushed..\n");
        }
}

void pop()
{
        int ITEM;
        struct node *TEMP;

        if(TOP == NULL)
        {
                printf("\nUNDERFLOW..\n");
                return;
        }
        else
        {
                ITEM = TOP->INFO;
                TEMP = TOP;
                TOP = TOP->LINK;

                free(TEMP);
                printf("Deleted Item: %d\n", ITEM);
        }
}
```

```
void display()
{
        struct node *PTR;

        if(TOP == NULL)
        printf("\nStack is empty..\n");
        else
        {
                printf("\nStack:\n");
                PTR = TOP;

                while(PTR != NULL)
                {
                        if(PTR == TOP)
                        printf("%d --->TOP\n", PTR->INFO);
                        else
                        printf("%d\n", PTR->INFO);

                        PTR = PTR->LINK;
                }
        }
}

main()
{
        int option;

        do
        {
                printf("\n\n*** STACK OPERATIONS ***\n");
                printf("1: PUSH\n");
                printf("2: POP\n");
                printf("3: DISPLAY STACK\n");
                printf("4: EXIT\n");
                printf("Enter valid option: ");
                scanf("%d", &option);

                switch(option)
                {
                        case 1: push();
                                break;

                        case 2: pop();
                                break;

                        case 3: display();
                                break;

                        case 4: printf("Press any key to exit..\n");
                                break;

                        default: printf("Invalid option..\n");
                }
        }
        while(option != 4);
}
```

Patil Sir

Implementing Queue by using Linked List:

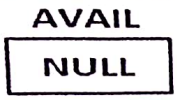
Algorithm to insert in queue:

Possibilities:

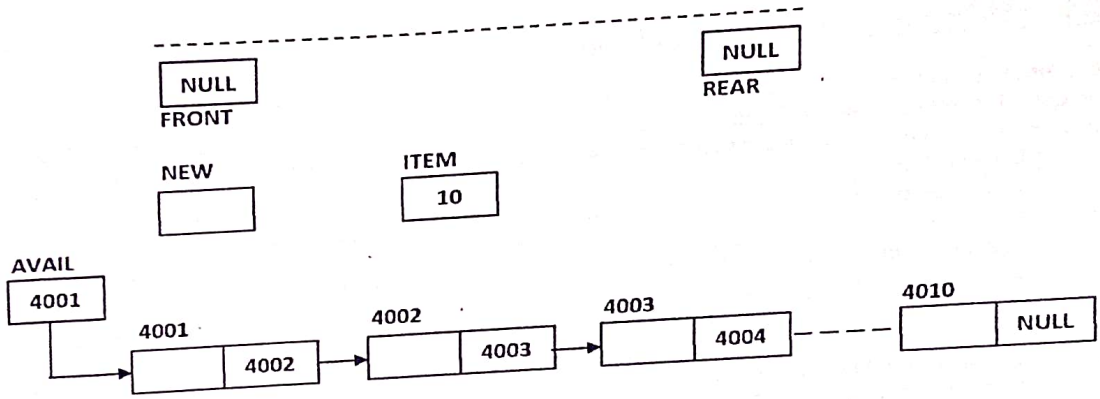
1. If free memory is not available then AVAIL = NULL
2. If queue is empty, then FRONT=NULL and REAR=NULL, then the new node will be the FRONT and REAR
3. If queue contains two or more elements then only REAR will be changed.

Variable	Description
FRONT	It is a pointer which points to the first node
REAR	It is a pointer which points to the last node
AVAIL	It is a pointer which points to the first node in the available memory.
ITEM	It is a regular variable which contains the value to insert
NEW	It is a pointer which points to the new node

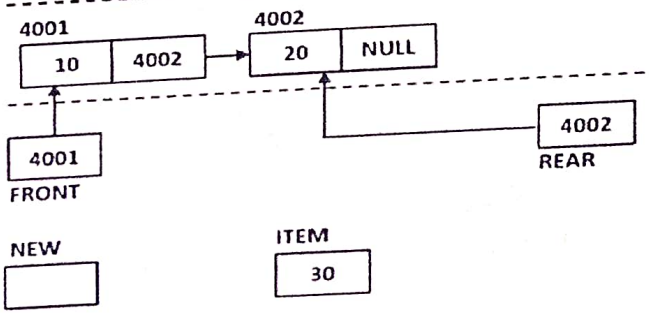
Case 1: If free memory is not available then AVAIL = NULL

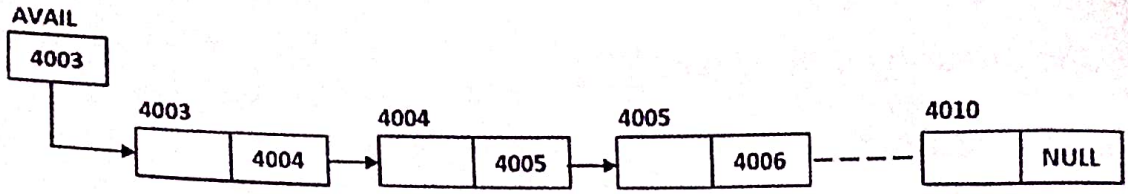


Case 2: If queue is empty, then FRONT=NULL and REAR=NULL, then the new node will be the FRONT and REAR



Case 3: If queue contains two or more elements then only REAR will be changed.





1. If AVAIL = NULL, Then:
2. Write: OVERFLOW
3. Exit
4. Else:
5. Set NEW := AVAIL
6. Set AVAIL := LINK[AVAIL]
7. Set INFO[NEW] := ITEM
8. Set LINK[NEW] := NULL

[If queue is empty]
9. If FRONT == NULL, Then:
10. FRONT := NEW
12. REAR := NEW
13. Else:
14. LINK[REAR] := NEW
15. REAR := NEW
[End of If]
16. [End of If]
17. Exit

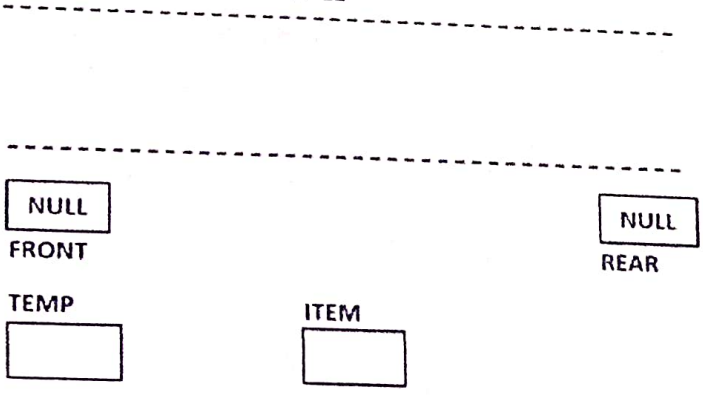
Algorithm to delete from queue (Linked List):

Possibilities:

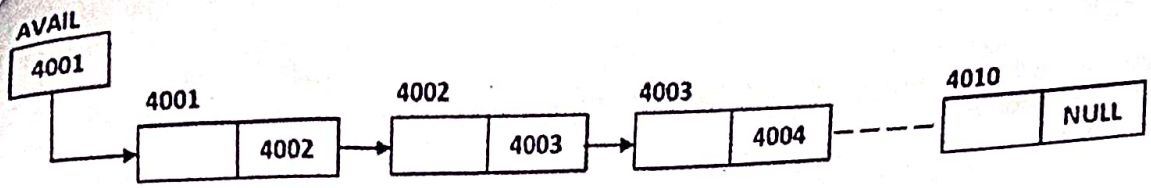
1. If queue is empty then FRONT = NULL
2. If the que contains only one element then after deleting the element then FRONT = NULL and REAR = NULL
3. If the queue contains two or more elements then after deleting the element FRONT will be change.

Variable	Description
FRONT	It is a pointer which points to the first node in the queue
REAR	It is a pointer which points to the last node in the queue
AVAIL	It is a pointer which points to the first node in the free memory
ITEM	It is a regular variable to store the deleted item
TEMP	It is a pointer which points to the deleted node

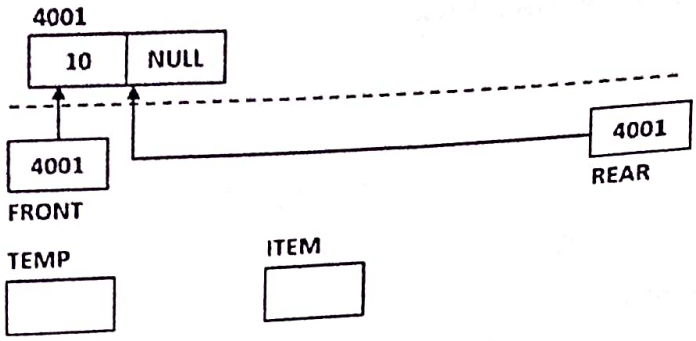
Case 1: If queue is empty then FRONT = NULL

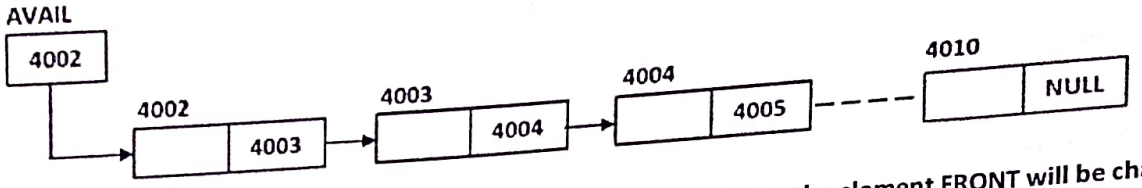


nd called as

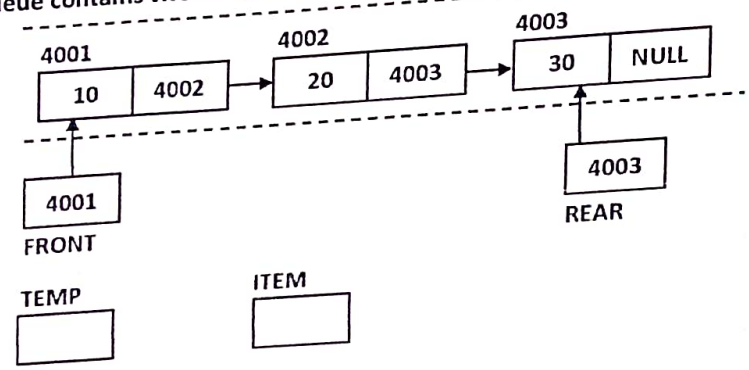


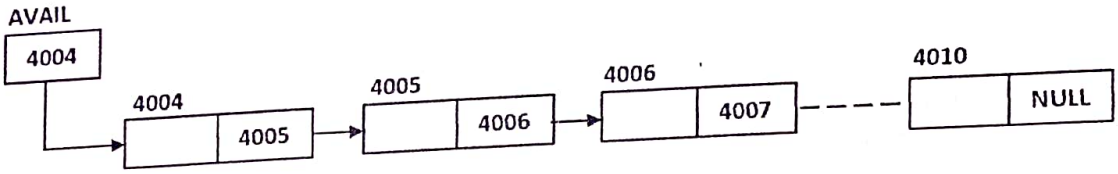
Case 2: If the que contains only one element then after deleting the element then FRONT = NULL and REAR = NULL





Case 3: If the queue contains two or more elements then after deleting the element FRONT will be change.

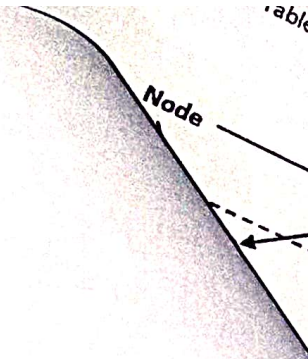




[If queue is empty]
1. If FRONT == NULL, Then:
2. Write: UNDERFLOW
3. Exit
4. Else:
5. Set TEMP := FRONT
6. Set ITEM := INFO[TEMP]
7. Set FRONT := LINK[TEMP]

8. If FRONT = NULL, Then:

g.



9. Set REAR = NULL
[End of If]

[Returning the deleted node to free memory as 1st node]
10. LINK[TEMP] := AVAIL
11. AVAIL := TEMP
[End of If]
12. Exit

Program: Write a program to implement queue using Linked List

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
        int INFO;
        struct node *LINK;
};

struct node *FRONT = NULL, *REAR = NULL;

void qinsert()
{
        int ITEM;
        struct node *NEW;

        NEW = (struct node*)malloc(sizeof(struct node));

        if(NEW == NULL)
        {
                printf("\nOVERFLOW..\n");
                return;
        }
        else
        {
                printf("\nEnter value of INFO: ");
                scanf("%d", &ITEM);

                NEW->INFO = ITEM;
                NEW->LINK = NULL;

                //if queue is empty
                if(FRONT == NULL)
                {
                        FRONT = NEW;
                        REAR = NEW;
                }
                else
                {
                        REAR->LINK = NEW;
                        REAR = NEW;
                }

                printf("Inserted..\n");
        }
}
```

```
void qdelete()
{
        struct node *TEMP;
        int ITEM;

        if(FRONT == NULL)
        {
                printf("\nUNDERFLOW..\n");
                return;
        }
        else
        {
                TEMP = FRONT;
                ITEM = TEMP->INFO;
                FRONT = TEMP->LINK;

                if(FRONT == NULL)
                REAR = NULL;

                free(TEMP);
                printf("\nDeleted Item: %d\n", ITEM);
        }
}

void traverse()
{
        struct node *PTR;

        if(FRONT == NULL && REAR == NULL)
        {
                printf("\nQueue is empty..\n");
                return;
        }
        else
        {
                PTR = FRONT;
                printf("\nQUEUE: ");

                while(PTR != NULL)
                {
                        printf("%d ", PTR->INFO);
                        PTR = PTR->LINK;
                }
                printf("\n");
        }
}

main()
{
        int option;

        do
        {
                printf("\n\n*** QUEUE OPERATIONS ***\n");
                printf("1: INSERT\n");
                printf("2: DELETE\n");
```

```
            printf("3: TRAVERSE/DISPLAY\n");
            printf("4: EXIT\n");
            printf("Enter valid option: ");
            scanf("%d", &option);

            switch(option)
            {
                    case 1: qinsert();
                            break;

                    case 2: qdelete();
                            break;

                    case 3: traverse();
                            break;

                    case 4: printf("\nPress any key to exit\n");
                            break;

                    default: printf("Invalid input..\n");
            }
    }
    while(option != 4);
}
```

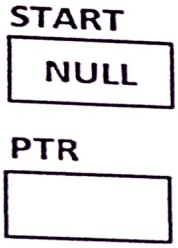
Algorithm to traverse a circular singly linked list:

Possibilities:

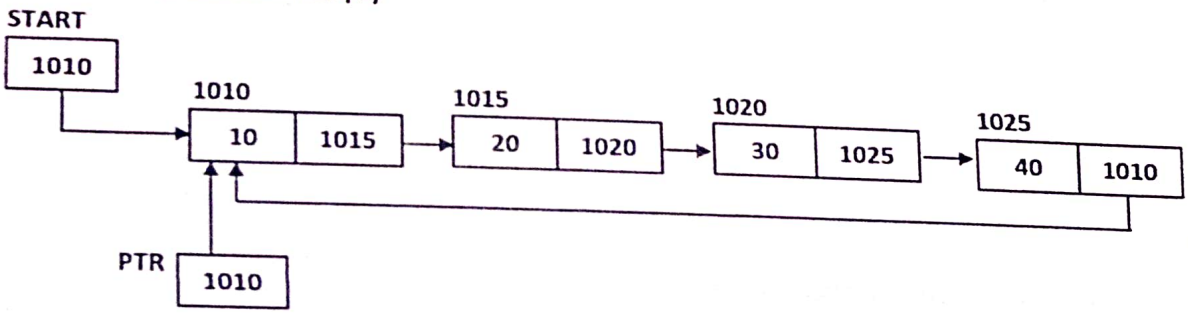
1. If linked list is empty
2. If linked list is not empty

Variable	Description
START	It is a pointer which points to the first node in the linked list
PTR	It is a pointer used to traverse the linked list.

Case 1: If Linked List is empty



Case 2: If linked List is not empty



```
1. Set PTR := START
[Checking for empty linked list]
2. If PTR = NULL, Then:
        Write: Linked List is empty.
3. Else:
4.      Repeat While LINK[PTR] != START:
5.              Write: INFO[PTR]
6.              Set PTR := LINK[PTR]
        [End of Step-4 loop]

7.      Write: INFO[PTR]
   [End of Step-2 If]
8. Exit
```

Linked List Algorithms

Algorithm to traverse a Singly linked list

```
1. Set PTR := START
2. Repeat Steps 3 and 4 while PTR != NULL
3.      Print: PTR -> INFO
4.      Set PTR := PTR -> LINK
   [End of Step 2 loop]
5. Exit
```

Searching in a Singly linked list:

```
1. Set PTR := START
2. Set LOC := NULL

3. Repeat while PTR != NULL:
4.      If ITEM = PTR->INFO, Then:
5.              Set LOC := PTR
6.              Break
7.      Else:
8.              Set PTR := PTR->LINK
        [End of Step-4 If]
   [End of Step-3 Loop]

9.  If LOC = NULL, Then:
10.     Print: Not Found
11. Else:
12.     Print: Found
    [End Step-9 if]
13. Exit
```

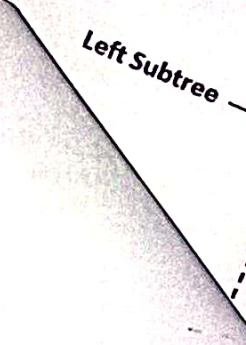
Inserting at the beginning of Singly linked list

```
1. If AVAIL = NULL Then:
2.      Write: OVERFLOW
3.      Exit
4. Else:
5.      Set NEW := AVAIL
6.      Set AVAIL := AVAIL->LINK
7.      Set NEW->INFO := ITEM
8.      Set NEW->LINK := START
9.      Set START := NEW
   [End of Step-1 if]
10. Exit
```

Inserting after a given node in Singly Linked List:

```
1. If AVAIL = NULL, Then:
        a. Write: OVERFLOW
        b. Exit
2. Else:
        a. Set NEW := AVAIL
        b. Set AVAIL := AVAIL->LINK
        c. Set NEW->INFO := ITEM

        [Checking for empty linked list]
        d. If LOC = NULL, Then: [If START = NULL]
                i. Set NEW->LINK := START
                ii. Set START := NEW
        e. Else:
                i. Set NEW->LINK := LOC->LINK
                ii. Set LOC->LINK := NEW
        [End of If]
   [End of If]
3. Exit
```



Deleting the node with given Location in Singly Linked List:

```
[Checking for empty linked list]
1. If START = NULL, Then:
2.      Write: Underflow
3.      Exit
4. Else:
        [if deleting 1st node or linked list contains only one node]
5.      If LOCP = NULL, Then:
6.              Set START := START -> LINK
                [Deletes first Node]
7.      Else:
8.              Set LOCP->LINK := LOC->LINK
        [End of If]
        [Return deleted node to free memory]
9.      Set LOC->LINK := AVAIL
10.     Set AVAIL := LOC
    [End of If]
11. Exit
```

Deleting node with given item of information in Singly Linked List:

Procedure/function: FIND

```
1. If START = NULL, Then:
2.      Set LOC := NULL
3.      Set LOCP = NULL
4.      Return
5. Else:
        [Checking for the 1st node.]
6.      If START->INFO = ITEM, Then:
7.              Set LOC := START
8.              Set LOCP := NULL
9.              Return
10.     Else:
                [Traversing from 2nd element up to last
                element to search the ITEM]
11.             Set SAVE := START
12.             Set PTR := START -> LINK
13.             Repeat While PTR != NULL:
14.                     If PTR->INFO = ITEM, Then:
15.                             Set LOC := PTR
16.                             Set LOCP := SAVE
17.                             Return
18.                     Else:
19.                             Set SAVE := PTR
20.                             Set PTR := PTR -> LINK
                        [End of If]
                [End of loop]
21.             Set LOC := NULL and LOCP := NULL

        [End of If]
    [End of If]
22. Return
```

Delete Algorithm:

```
1. Call Find
2. If LOC = NULL, Then:
3.      Write: ITEM not found
4.      Exit
5. Else:
6.      If LOCP = NULL Then: [If LOC == START Then:]
7.              Set START := START -> LINK
                [Delete first node]
8.      Else:
9.              Set LOCP -> LINK := LOC -> LINK
        [End of If]

        [Return deleted node to the available memory]
10.     Set LOC -> LINK := AVAIL
11.     Set AVAIL := LOC
    [End of if]
12. Exit
```

Implementing Stack by using Linked-List

Algorithm to PUSH:

```
1. If AVAIL = NULL, Then:
2.        Write: OVERFLOW
3.        Exit
4. Else:
5.        Set NEW := AVAIL
6.        Set AVAIL := AVAIL -> LINK
7.        Set NEW->INFO := ITEM
8.        Set NEW->LINK := TOP
9.        Set TOP := NEW
   [End of If]
10. Exit
```

Algorithm to POP:

```
[If stack is empty]
1. If TOP = NULL, Then:
2.        Write: UNDERFLOW
3.        Exit
4. Else:
5.        Set ITEM := TOP->INFO
6.        Set TEMP := TOP
7.        Set TOP := TOP->LINK

          [Return the deleted node to free memory]
8.        Set TEMP->LINK := AVAIL
9.        Set AVAIL := TEMP
   [End of If]
10. Exit
```

Implementing Queue by using Linked List:

Algorithm to insert in queue:

```
1. If AVAIL = NULL, Then:
2.        Write: OVERFLOW
3.        Exit
4. Else:
5.        Set NEW := AVAIL
6.        Set AVAIL := AVAIL->LINK
7.        Set NEW->INFO := ITEM
8.        Set NEW->LINK := NULL
          [If queue is empty]
9.        If FRONT == NULL, Then:
10.                FRONT := NEW
12.                REAR := NEW
13.       Else:
14.                REAR->LINK := NEW
15.                REAR := NEW
          [End of If]
16. [End of If]
17. Exit
```

Algorithm to delete from queue (Linked List):

```
[If queue is empty]
1. If FRONT == NULL, Then:
2.        Write: UNDERFLOW
3.        Exit
4. Else:
5.        Set TEMP := FRONT
6.        Set ITEM := TEMP->INFO
7.        Set FRONT := TEMP->LINK

8.        If FRONT = NULL, Then:
9.                Set REAR = NULL
          [End of If]

          [Returning the deleted node to free memory as 1st
node]
10.       TEMP->LINK := AVAIL
11.       AVAIL := TEMP
   [End of If]
12. Exit
```

Algorithm to traverse a circular singly linked list:

```
1. Set PTR := START
[Checking for empty linked list]
2. If PTR = NULL, Then:
          Write: Linked List is empty.
3. Else:
4.        Repeat While PTR->LINK != START:
5.                Write: PTR->INFO
6.                Set PTR := PTR->LINK
          [End of Step-4 loop]
7.        Write: PTR->INFO
   [End of Step-2 If]
8. Exit
```