CBCS SCHEME

USN | | | | | | | | | | |

20MCA11

First Semester MCA Degree Examination, Jan./Feb. 2021
Data Structures with Algorithms

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

1 a. What are data structures? Explain the classifications of data structures. **(06 Marks)**

b. Define STACK. Write a C program to implement stack operations using arrays (by passing parameters). **(08 Marks)**

c. Write a C program to convert infix to postfix expression. **(06 Marks)**

OR

2 a. Convert the following infix expression into postfix using application stack,
A + (B*C – (D/E^F)*G)*H **(10 Marks)**

b. Implement a program in C for evaluating a postfix expression. **(10 Marks)**

Module-2

3 a. What is recursion? Write a program to implement tower of Hanoi problem using recursion and trace the output for 3 disks. **(10 Marks)**

b. What is queue? Write algorithms for the primitive operations performed on queue. **(10 Marks)**

OR

4 a. Give the disadvantages of an ordinary queue and how it is solved in a circular queue? Write C program to implement circular queue. **(10 Marks)**

b. What are priority queue? Write a program to simulate the working of priority queue. **(10 Marks)**

Module-3

5 a. Differentiate static versus dynamic memory allocation. How dynamic memory allocation is done in C? **(10 Marks)**

b. Write C program to implement following operations on singly linked list :
(i) Insert a node at the end of the list.
(ii) Remove a node at end of list. **(10 Marks)**

OR

6 a. What are the limitations of array over linked lists? **(10 Marks)**

b. Explain linked implementation of stack with suitable diagram. Also write algorithms to implement stack push and pop operation using singly linked lists. **(10 Marks)**

Module-4

7 a. Explain various steps in the fundamentals of algorithmic problem solving. **(10 Marks)**

b. List out important problem types in the study of algorithms. Explain any two of them. **(10 Marks)**

OR

8 a. Define algorithm. Explain different asymptotic notations. **(10 Marks)**

b. Show that if $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$ then,
$t_1(n) \in t_2(n) \in O(\max(g_1(n), g_2(n)))$. **(10 Marks)**

Module-5

9 a. Write an algorithm to sort given 'n' element using bubble sort and find its time efficiency. **(10 Marks)**

b. Write an algorithm to implement Brute Force's string matching process and apply the same for the given input.
Text string = [Hello, How Are You?]
Pattern string = [How] **(10 Marks)**

OR

10 a. Write an algorithm for selection sort and analyze. **(10 Marks)**

b. Write DFS graph travels algorithm and write a trace for the following graph: (Refer Fig. Q10 (b)).

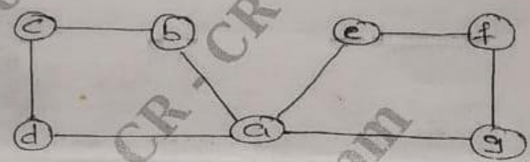


Fig. Q10 (b)

(10 Marks)

* * * * *

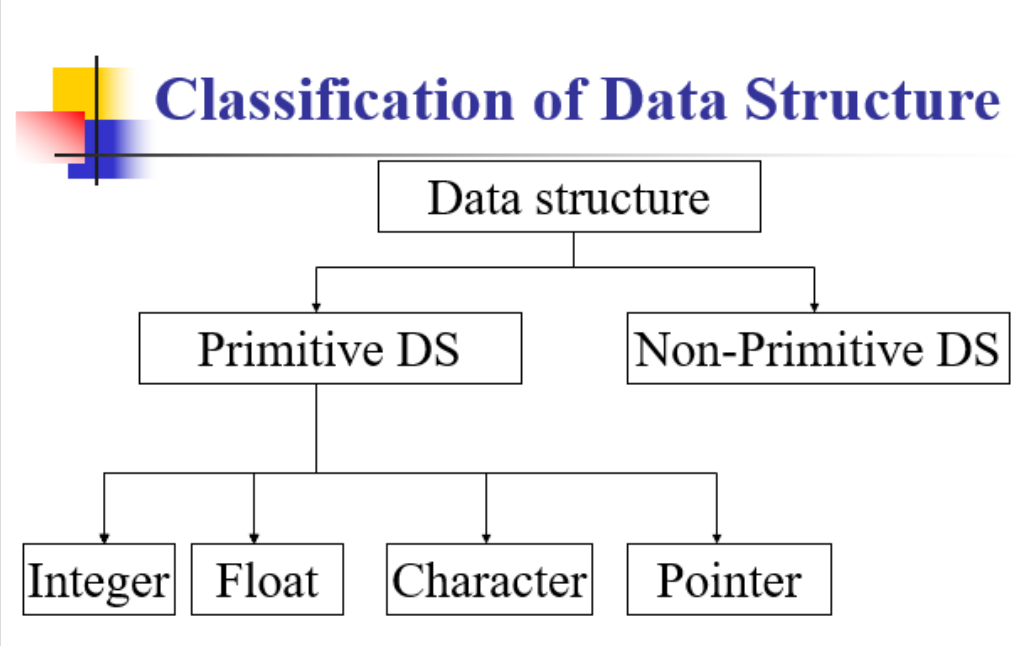
1a. Data structure is representation of the logical relationship existing between individual elements of data. In other words, a data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other. Data structure affects the design of both structural & functional aspects of a program.

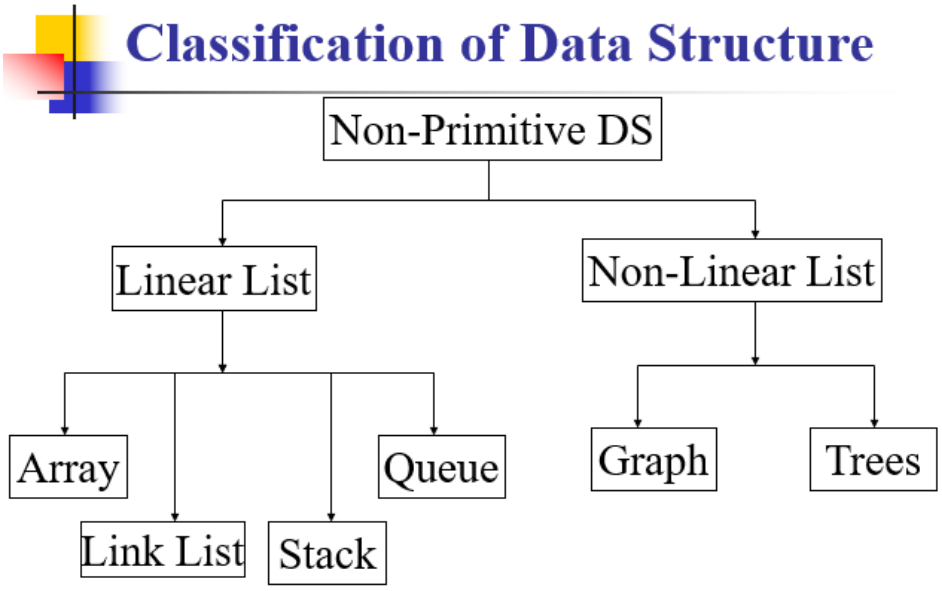
Program=algorithm + Data Structure

Classification of Data Structure

Data structure are normally divided into two broad categories:

- Primitive Data Structure
- Non-Primitive Data Structure





Primitive Data Structure

There are basic structures and directly operated upon by the machine instructions. In general, there are different representation on different computers. Integer, Floating-point number, Character constants, string constants, pointers etc., fall in this category.

Non-Primitive Data Structure

There are more sophisticated data structures. These are derived from the primitive data structures. The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items. Lists, Stack, Queue, Tree, and Graph are example of non-primitive data structures. The design of an efficient data structure must take operations to be performed on the data structure.

A primitive data structure is generally a basic structure that is usually built into the language, such as an integer, a float.

A non-primitive data structure is built out of primitive data structures linked together in meaningful ways, such as a or a linked-list, binary search tree, AVL Tree, graph etc.

LINEAR DATA STRUCTURES:

In linear data structure the elements are stored in sequential order. The linear data structures are

- **Array:** Array is a collection of data of same data type stored in consecutive memory location and is referred by common name
- **Linked list:** Linked list is a collection of data of same data type but the data items need not be stored in consecutive memory locations.
- **Stack:** A stack is a Last-In-First-Out linear data structure in which insertion and deletion takes place at only one end called the top of the stack.

- **Queue**: A Queue is a First in First-Out Linear data structure in which insertions takes place one end called the rear and the deletions takes place at one end called the Front.

NON-LINEAR DATA STRUCTURE:

Elements are stored based on the hierarchical relationship among the data. The following are some of the Non-Linear data structure

Trees:

- Trees are used to represent data that has some hierarchical relationship among the data elements.

Graph:

- Graph is used to represent data that has relationship between pair of elements not necessarily hierarchical in nature. For example electrical and communication networks, airline routes, flow chart, graphs for planning projects.

Q1b. Ans

- At stack is ordered list in which all insertion and deletion are made at one end, called the TOP
- Stores a set of elements in a particular order
- Stack principle: LAST IN FIRST OUT
- = LIFO
- It means: the last element inserted is the first one to be removed

```
 int top=-1,stack[MAX];
void push(int);
int pop();
void display();

void main()
{
        int ch;
```

```
        while(1)        //infinite loop, will end when choice will be 4
        {
                printf("\n*** Stack Menu ***");
                printf("\n\n1.Push\n2.Pop\n3.Display\n4.Exit");
                printf("\n\nEnter your choice(1-4):");
                scanf("%d", &ch);
                switch(ch)
                {
                        case 1: printf("\nEnter element to push:");
                                        scanf("%d",&val);

                                        push(val);
                                        break;
                        case 2: printf("Deleted ite is %d", pop());
                                        break;
                        case 3: display();
                                        break;
                        case 4: exit(0);

                        default: printf("\nWrong Choice!!");
                }
        }
}

void push(int item)
{
        int val;

        if(top==MAX-1)
        {
```

```
                printf("\nStack is full!!");
        }
        else
        {

                top=top+1;
                stack[top]=item;
        }
}

int pop()
{
        if(top==-1)
        {
                printf("\nStack is empty!!");
                return 0;
        }
        else
        {
                Int i=stack[top]);
                top=top-1;
                return i;

        }
}

void display()
{
        int i;
                if(top==-1)
```

```
        {
                printf("\nStack is empty!!");
        }
        else
        {
                printf("\nStack is...\n");
                for(i=top;i>=0;i--)
                        printf("%d\n",stack[i]);
        }
}
```

Q1c Ans:

```
#include<stdio.h>
#include<ctype.h>

char stack[100];
int top = -1;

void push(char x)
{
   stack[++top] = x;
}

char pop()
{
   if(top == -1)
      return -1;
   else
      return stack[top--];
}
```

```
int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
    return 0;
}

int main()
{
    char exp[100];
    char *e, x;
    printf("Enter the expression : ");
    scanf("%s",exp);
    printf("\n");
    e = exp;

    while(*e != '\0')
    {
        if(isalnum(*e))
            printf("%c ",*e);
        else if(*e == '(')
            push(*e);
        else if(*e == ')')
        {
            while((x = pop()) != '('
```

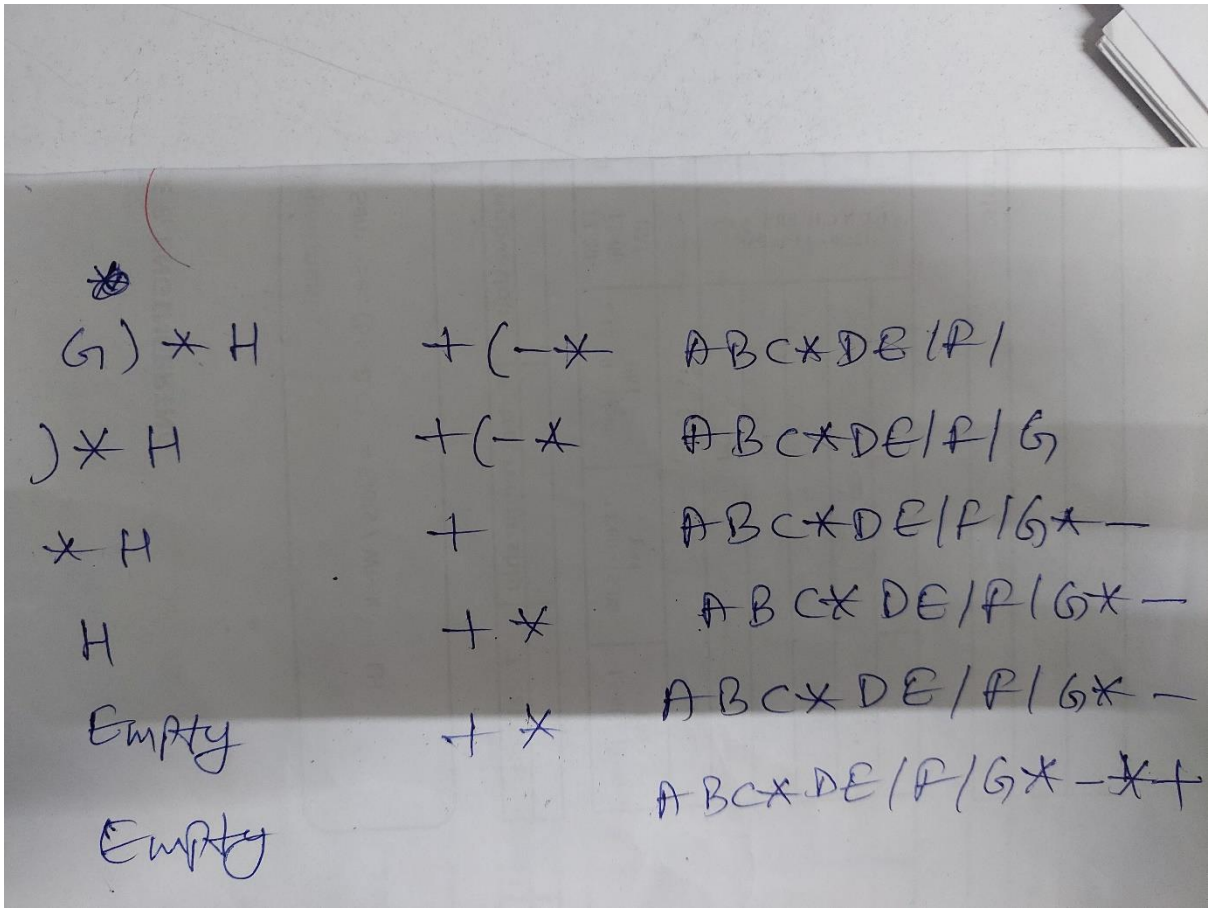
```
            printf("%c ", x);
        }
        else
        {
            while(priority(stack[top]) >= priority(*e))
                printf("%c ",pop());
            push(*e);
        }
        e++;
    }

    while(top != -1)
    {
        printf("%c ",pop());
    }return 0;
}
```

I/O: Enter the Expression: a+b*c

abc*+

Q2a Ans:

Expression	stack	Output	Comm
A+(B*C-(D/E/F)*G)*H	Empty	-	In
+(B*C-(D/E/F)*G)*H	Empty	A	P
(B*C-(D/E/F)*G)*H	+	A	Pu
B*C-(D/E/F)*G)*H	+(	A	Pu
*C-(D/E/F)*G)*H	+(	AB	Pu
C-(D/E/F)*G)*H	+(*	AB	Pus
-(D/E/F)*G)*H	+(*	ABC	Pus
(D/E/F)*G)*H	+(-	ABC*	
D/E/F)*G)*H	+(-(	ABC*	
/E/F)*G)*H	+(-(	ABC*D	
E/F)*G)*H	+(-(/	ABC*D	
/F)*G)*H	+(-(/	ABC*DE	
F)*G)*H	+(-(/	ABC*DE/	
)*G)*H	+(-(/	ABC*DE/F	
*G)*H	+(-	ABC*DE/F/	



Q2b Ans:

```
#include<stdio.h>
int stack[20];
int top = -1;

void push(int x)
{
   stack[++top] = x;
}

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

```
int main()
{
    char exp[20];
    char *e;
    int n1,n2,n3,num;
    printf("Enter the expression :: ");
    scanf("%s",exp);
    e = exp;
    while(*e != '\0')
    {
        if(isdigit(*e))
        {
            num = *e - 48;
            push(num);
        }
        else
        {
            n1 = pop();
            n2 = pop();
            switch(*e)
            {
            case '+':
            {
                n3 = n1 + n2;
                break;
            }
            case '-':
            {
                n3 = n2 - n1;
```

```
            break;
        }
        case '*':
        {
            n3 = n1 * n2;
            break;
        }
        case '/':
        {
            n3 = n2 / n1;
            break;
        }
        }
        push(n3);
      }
      e++;
   }
   printf("\nThe result of expression %s  =  %d\n\n",exp,pop());
   return 0;
}
```

I/O: Enter the Expression: 234+*

The result of expression 234+* = 14

Q3a Ans:

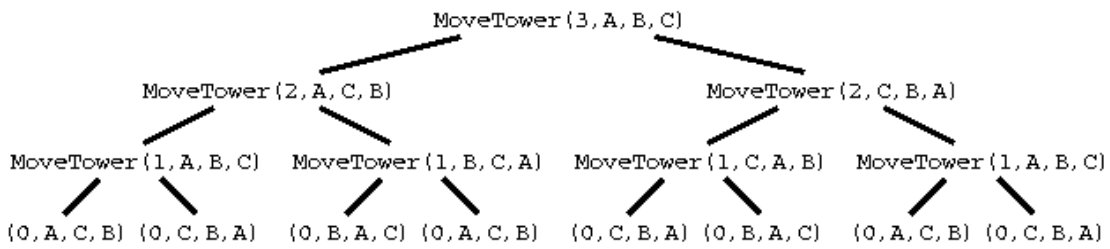
Sometimes, the best way to solve a problem is by solving a smaller version of the exact same problem first. Recursion is a technique that solves a problem by solving a smaller problem of the same type. When you turn this into a program, you end up with functions that call themselves (*recursive functions*)

```
int f(int x)
{
 int y;

 if(x==0)
    return 1;
 else {
    y = 2 * f(x-1);
    return y+1;
 }
}
```

Program for tower of Hanoi:

```
void hanoi(int from, int to, int num)
{
   int temp = 6 - from - to; //find the temporary
                        //storage column
   if (num == 1){
      cout << "move disc 1 from " << from
          << " to " << to << endl;
    }
   else {
      hanoi(from, temp, num - 1);
      cout << "move disc " << num << " from " << from
          << " to " << to << endl;
      hanoi(temp, to, num - 1);
   }
}
int main() {
   int num_disc;  //number of discs
```

```
    cout << "Please enter a positive number (0 to quit)";
    cin >> num_disc;

    while (num_disc > 0){
        hanoi(1, 3, num_disc);
        cout << "Please enter a positive number ";
        cin >> num_disc;
    }
    return 0;
}
```



Q3 b Ans:

It is an ordered group of homogeneous items of elements.

Queues have two ends:

Elements are added at one end.

Elements are removed from the other end.

The element added first is also removed first (**FIFO**: First In, First Out).

Definitions: (provided by the user)

- *MAX_ITEMS*: Max number of items that might be on the queue
- *ItemType*: Data type of the items on the queue

Operations

- Boolean IsEmpty
- Boolean IsFull
- Enqueue (ItemType newItem)

- Dequeue (ItemType& item)

Procedure for Insertion

QINSERT(QUEUE,N,FRONT,REAR,ITEM)

This procedure inserts an element ITEM on to a queue.

1. [Queue already filled?]
   if FRONT=1 and REAR=N, or if FRONT=REAR+1,then;
   Write: OVERFLOW, and Return

2. [Find new value of REAR]
   if FRONT:=NULL, then:[Queue initially empty]
   Set FRONT:=1 and REAR:=1
   else if REAR=N, then:
   Set REAR:=REAR+1

3. [Inserts new element]
   Set QUEUE[REAR]:=ITEM.

4. return.

Procedure for Deletion

QINSERT(QUEUE,N,FRONT,REAR,ITEM)

This procedure deletes an element from a queue and assigns it to the variable ITEM.

1. [Queue already empty]
   if FRONT:=NULL, then: Write : UNDERFLOW, and Return

2. Set ITEM:=QUEUE[FRONT]

3. [Find new value of FRONT]
   if FRONT=REAR, then: [Queue has only one element to start]
   Set FRONT:=NULL and REAR:=NULL
   else If FRONT=N, then:
   Set FRONT:=1
   else
   Set FRONT:=FRONT+1

4. return

Q4a Ans:

In Circular Queue, the elements of a given queue can be stored efficiently in an array. Efficient utilization of memory: In the circular queue, there is no wastage of memory as it uses the unoccupied space, and memory is used properly in a valuable and effective manner as compared to a linear queue.

```
#include<stdio.h>
#define max 3

int CQueue[max];
int rear=-1;
int front=-1;

void CQInsert()
{
	if((front==(rear+1)%max))
	printf("\n Queue is full");
	else
	{
		printf("\n Enter the element to be inserted:\n");
		rear=(rear+1)%max;
		scanf("%d",&CQueue[rear]);
		if(front==-1)
		front=0;
	}
}

void CQDelete()
{
	if(front==-1)
	printf("\n Queue is empty");
	else
	{
		/*item=CQueue[front];*/
		printf("Deleted item is %d",CQueue[front]);
		/*CQueue[front]=NULL;*/
		if(front==rear)
		front=rear=-1;
		else
		front=(front+1)%max;
	}

}

void CQDisplay()
{
	int i;
	if(front==-1)
	printf("\n Queue is empty");
	else
	{
		printf("\n The CQ elements are");
		for(i=front;i<=rear;i++)
			printf("\n%d",CQueue[i]);
		if(front>rear)
		{
```

```
                        for(i=front;i<max;i++)
                                printf("\n%d",CQueue[i]);
                        for(i=0;i<=rear;i++)
                                printf("\n%d",CQueue[i]);
                }
        }

}

void main()
{
        int c;
        clrscr();
        for(;;)
        {

                printf("\n 1.Insert ");
                printf("\n 2.Delete ");
                printf("\n 3.Display");
                printf("\n 4.Exit   ");
                printf("\n Enter the choice");
                scanf("%d",&c);
                switch(c)
                {
                        case 1 : CQInsert();
                                 break;
                        case 2 : CQDelete();
                                 break;
                        case 3 : CQDisplay();
                                 break;
                        case 4 : exit(0);

                        default : printf("\n Invalid choice");
                                  getch();
                }
        }
}
```

==========SAMPLE OUTPUT==========

```
1.Insert
2.Delete
3.Display
4.Exit
Enter the choice: 1
```

Enter the element to be inserted: 23

1.Insert
2.Delete
3.Display
4.Exit
Enter the choice: 1

Enter the element to be inserted: 45

1.Insert
2.Delete
3.Display
4.Exit
Enter the choice:3

The CQ elements are
23
45

1.Insert
2.Delete
3.Display
4.Exit
Enter the choice: 2

Deleted item is 23

Q4 b. Ans:

Priority Queue is an extension of the Queue data structure where each element has a particular priority associated with it. It is based on the priority value, the elements from the queue are deleted. dequeue(): This function removes the element with the highest priority from the queue.

```
#include<stdio.h>
#include<conio.h>

int arr[10],front=0,rear=-1;

void check(int data)
{
   int i,j;

   for (i = 0; i <= rear; i++)
```

```
        {
            if (data >= pri_que[i])
            {
                for (j = rear + 1; j > i; j--)
                {
                    pri_que[j] = pri_que[j - 1];
                }
                pri_que[i] = data;
                return;
            }
        }
        pri_que[i] = data;
    }

void addq(int data)
{
    if (rear >= MAX - 1)
    {
        printf("\nQueue overflow no more elements can be inserted");
        return;
    }
    if ((front == -1) && (rear == -1))
    {
        front++;
        rear++;
        pri_que[rear] = data;
        return;
    }
    else
        check(data);
    rear++;
}
void delq()
{
        if(front>rear || front==-1)
        printf("\n Queue is empty");
        else
        {
                printf("\n Deleted item is %d",arr[front]);
                front++;
        }
}

void disp()
{
        int i;
        if(rear==-1 || front>rear)
        printf("\n Queue is empty");
        else
        for(i=front;i<=rear;i++)
```

```
        printf("\n%d",arr[i]);
}

void main()
{
        int c;
        clrscr();
        do
        {
                printf("\n 1.Insert");
                printf("\n 2.Deletion");
                printf("\n 3.Display");
                printf("\n 4.Exit");
                printf("\n Enter the choice:\n");
                scanf("%d",&c);
                if(c==1)
                addq();
                else if(c==2)
                delq();
                else if(c==3)
                disp();
                else
                exit(1);
        }while(1);
}
```

Q5a Ans:

S.No	Static Memory Allocation	Dynamic Memory Allocation
1	In the static memory allocation, variables get allocated permanently.	In the Dynamic memory allocation, variables get allocated only if your program unit gets active.
2	Static Memory Allocation is done before program execution.	Dynamic Memory Allocation is done during program execution.
3	It uses stack for managing the static allocation of memory	It uses heap for managing the dynamic allocation of memory

4	It is less efficient	It is more efficient
5	In Static Memory Allocation, there is no memory re-usability	In Dynamic Memory Allocation, there is memory re-usability and memory can be freed when not required
6	In static memory allocation, once the memory is allocated, the memory size can not change.	In dynamic memory allocation, when memory is allocated the memory size can be changed.
7	In this memory allocation scheme, we cannot reuse the unused memory.	This allows reusing the memory. The user can allocate more memory when required. Also, the user can release the memory when the user needs it.
8	In this memory allocation scheme, execution is faster than dynamic memory allocation.	In this memory allocation scheme, execution is slower than static memory allocation.
9	In this memory is allocated at compile time.	In this memory is allocated at run time.
10	In this allocated memory remains from start to end of the program.	In this allocated memory can be released at any time during the program.
11	**Example:** This static memory allocation is generally used for array.	**Example:** This dynamic memory allocation is generally used for linked list.

, C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

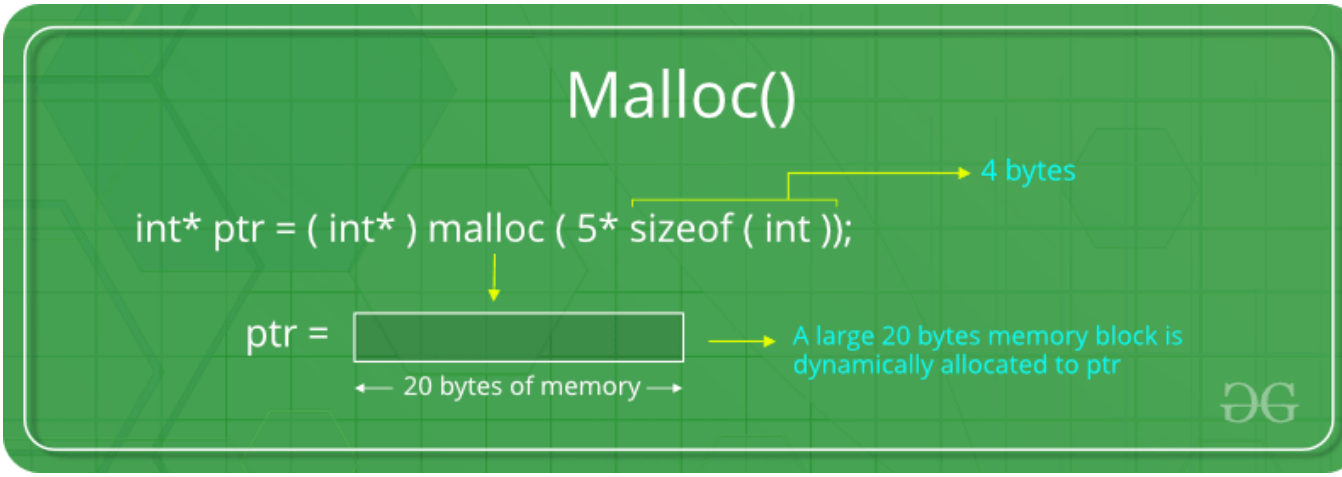
1. malloc()
2. calloc()
3. free()
4. realloc()

Let’s look at each of them in greater detail.

1. C malloc() method

“malloc” or **“memory allocation”** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value.
Syntax:
ptr = (cast-type*) malloc(byte-size)

For Example:
ptr = (int*) malloc(100 * sizeof(int));
Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

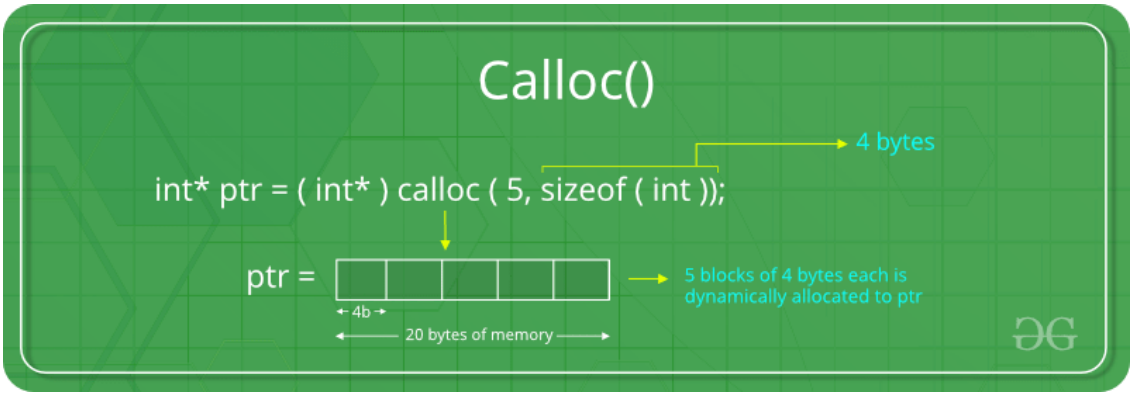


If space is insufficient, allocation fails and returns a NULL pointer.

2. C calloc() method

“calloc” or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.
Syntax:
ptr = (cast-type*)calloc(n, element-size);

For Example:
ptr = (float*) calloc(25, sizeof(float));
This statement allocates contiguous space in memory for 25 elements each with the size of the float.



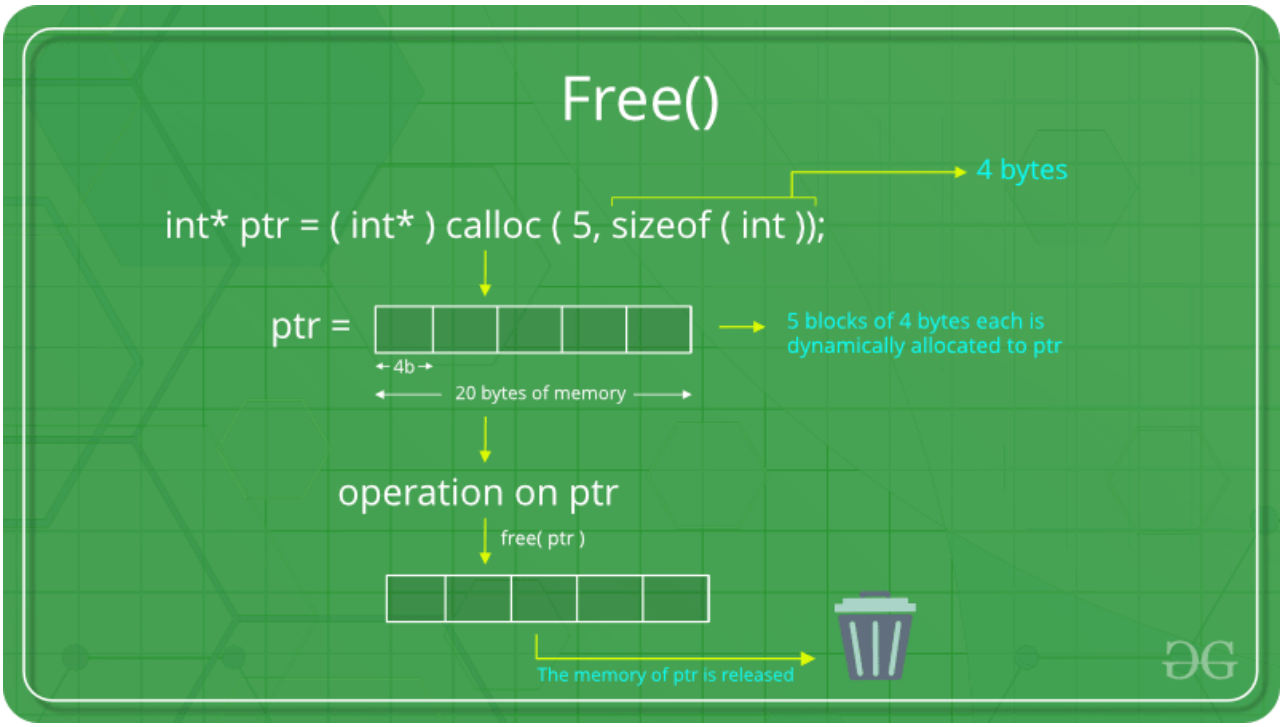
If space is insufficient, allocation fails and returns a NULL pointer.

3. C free() method

"free" method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax:

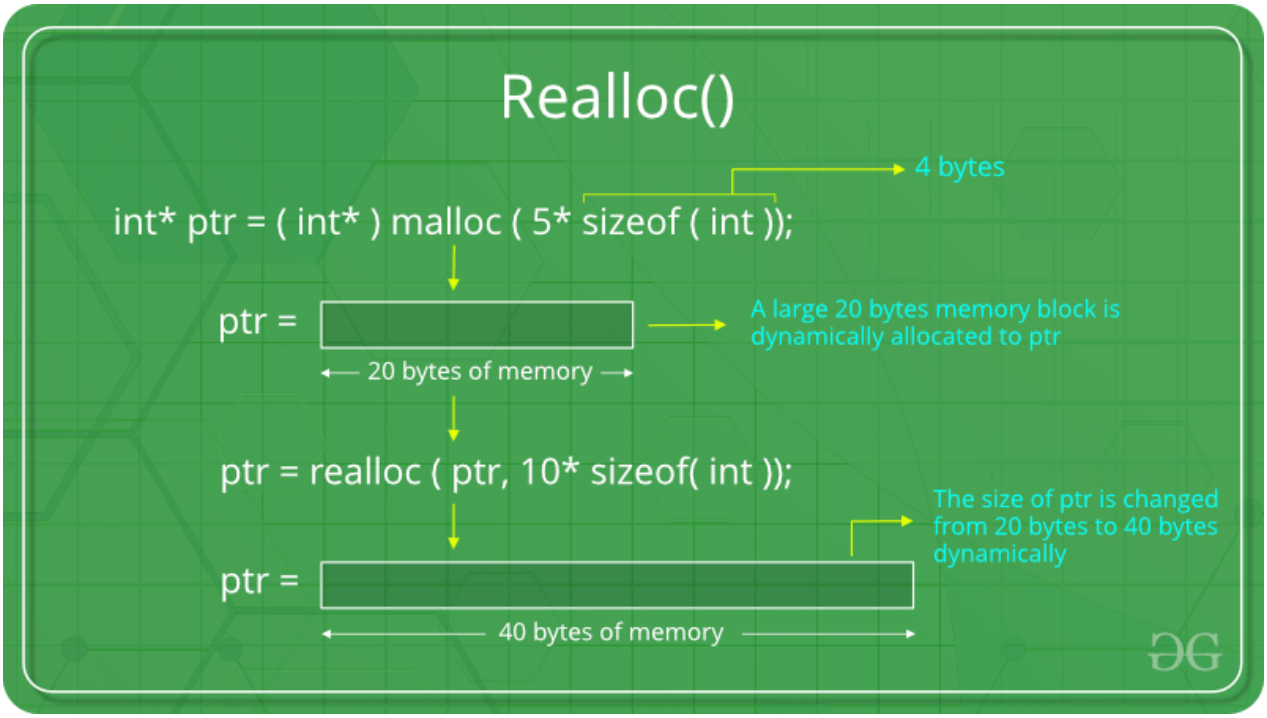
free(ptr);



4. C realloc() method

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with default garbage value.
Syntax:
ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'.



If space is insufficient, allocation fails and returns a NULL pointer.

```
#include <stdio.h>

#include <stdlib.h>



int main()

{
```

```
// This pointer will hold the

// base address of the block created

int* ptr;

int n, i;



// Get the number of elements for the array

n = 5;

printf("Enter number of elements: %d\n", n);



// Dynamically allocate memory using calloc()

ptr = (int*)calloc(n, sizeof(int));



// Check if the memory has been successfully

// allocated by malloc or not

if (ptr == NULL) {

    printf("Memory not allocated.\n");

    exit(0);

}
```

```
else {

    // Memory has been successfully allocated
    printf("Memory successfully allocated using calloc.\n");

    // Get the elements of the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 1;
    }

    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }

    // Get the new size for the array
    n = 10;
    printf("\n\nEnter the new size of the array: %d\n", n);
```

```
// Dynamically re-allocate memory using realloc()
ptr = realloc(ptr, n * sizeof(int));

// Memory has been successfully allocated
printf("Memory successfully re-allocated using realloc.\n");

// Get the new elements of the array
for (i = 5; i < n; ++i) {
    ptr[i] = i + 1;
}

// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
    printf("%d, ", ptr[i]);
}

free(ptr);
```

```
        }


        return 0;

    }
```

Output:

Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array: 10

Memory successfully re-allocated using realloc.

The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

Q 5b. Ans:

```
#include<stdio.h>
#include<stdlib.h>
struct Link
{
   int info;
   struct Link* next;
};
typedef struct Link* NODE;
NODE first=NULL;
NODE getnode()
{
   NODE ptr;
   ptr= (NODE)malloc(sizeof(struct Link));
   if (ptr==NULL)
```

```
    {
        printf("Can Not Allocate");
        return 0;
    }
    ptr->next= NULL;
    return ptr;
}
void freenode(NODE ptr)
{
    free(ptr);
}
void insert_last()
{
    NODE ptr;
    ptr=getnode();
    printf("Enter the element to insert");
    scanf("%d",&ptr->info);
    if(first==NULL)
    first=ptr;
    else
    {   NODE temp;
        temp=first;
        while(temp->next!=NULL)
        {
            temp=temp->next;
        }
        temp->next=ptr;



    }
```

```
}
void delete_last()
{
   NODE ptr,temp,prev;
   if(first==NULL)
   printf("List is Empty, can not perform deletion");
   else if(first->next==NULL)
   {
      printf("Deleted value is %d", first->info);
      free(first);
   }
   else
   {
      prev=NULL;
      temp=first;
      while(temp->next!=NULL)
      {
         prev=temp;
         temp=temp->next;
      }
      printf("Deleted value is %d", temp->info);
      free(temp);
      prev->next=NULL;
   }
}
void main ()
{
   int choice =0;
   while(choice != 9)
   {
```

```
      printf("\n\n*********Main Menu*********\n");
       printf("\nChoose one option from the following list ...\n");
       printf("\n===============================================\n");
       printf("\n1..Delete from last\n2.Insert at Last\n 3.Exit\n");
       printf("\nEnter your choice?\n");
       scanf("\n%d",&choice);
       switch(choice)
       {
case 1:
          insert_last();
          break;
          case 2:
          delete_last();
          break;
          case 3:
          exit(0)
          break;
          default:
          printf("Please enter valid choice..");
       }
   }
}
```

Q 6 a Ans:

Arrays store elements in contiguous memory locations, resulting in easily calculable addresses for the elements stored and this allows a faster access to an element at a specific index. Linked lists are less rigid in their storage structure and elements are usually not stored in contiguous locations, hence they need to be stored with additional tags giving a reference to the next element.

Major differences are listed below:

- **Size:** Since data can only be stored in contiguous blocks of memory in an array, its size cannot be altered at runtime due to risk of overwriting over other data. However in a

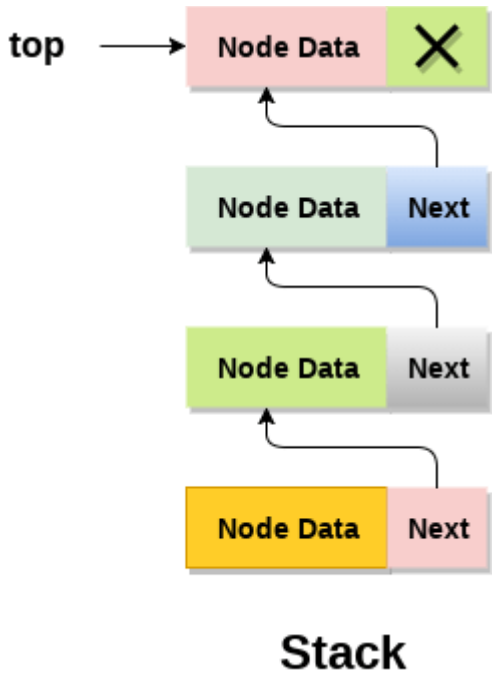
linked list, each node points to the next one such that data can exist at scattered (non-contiguous) addresses; this allows for a dynamic size which can change at runtime.

- **Memory allocation:** For arrays at compile time and at runtime for linked lists. but, dynamically allocated array also allocates memory at runtime.
- **Memory efficiency:** For the same number of elements, linked lists use more memory as a reference to the next node is also stored along with the data. However, size flexibility in linked lists may make them use less memory overall; this is useful when there is uncertainty about size or there are large variations in the size of data elements; memory equivalent to the upper limit on the size has to be allocated (even if not all of it is being used) while using arrays, whereas linked lists can increase their sizes step-by-step proportionately to the amount of data.
- **Execution time:** Any element in an array can be directly accessed with its index; however in case of a linked list, all the previous elements must be traversed to reach any element. Also, better cache locality in arrays (due to contiguous memory allocation) can significantly improve performance. As a result, some operations (such as modifying a certain element) are faster in arrays, while some other (such as inserting/deleting an element in the data) are faster in linked lists.

Following are the points in favour of Linked Lists.
(1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, the upper limit is rarely reached.

Q6 b Ans:

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.

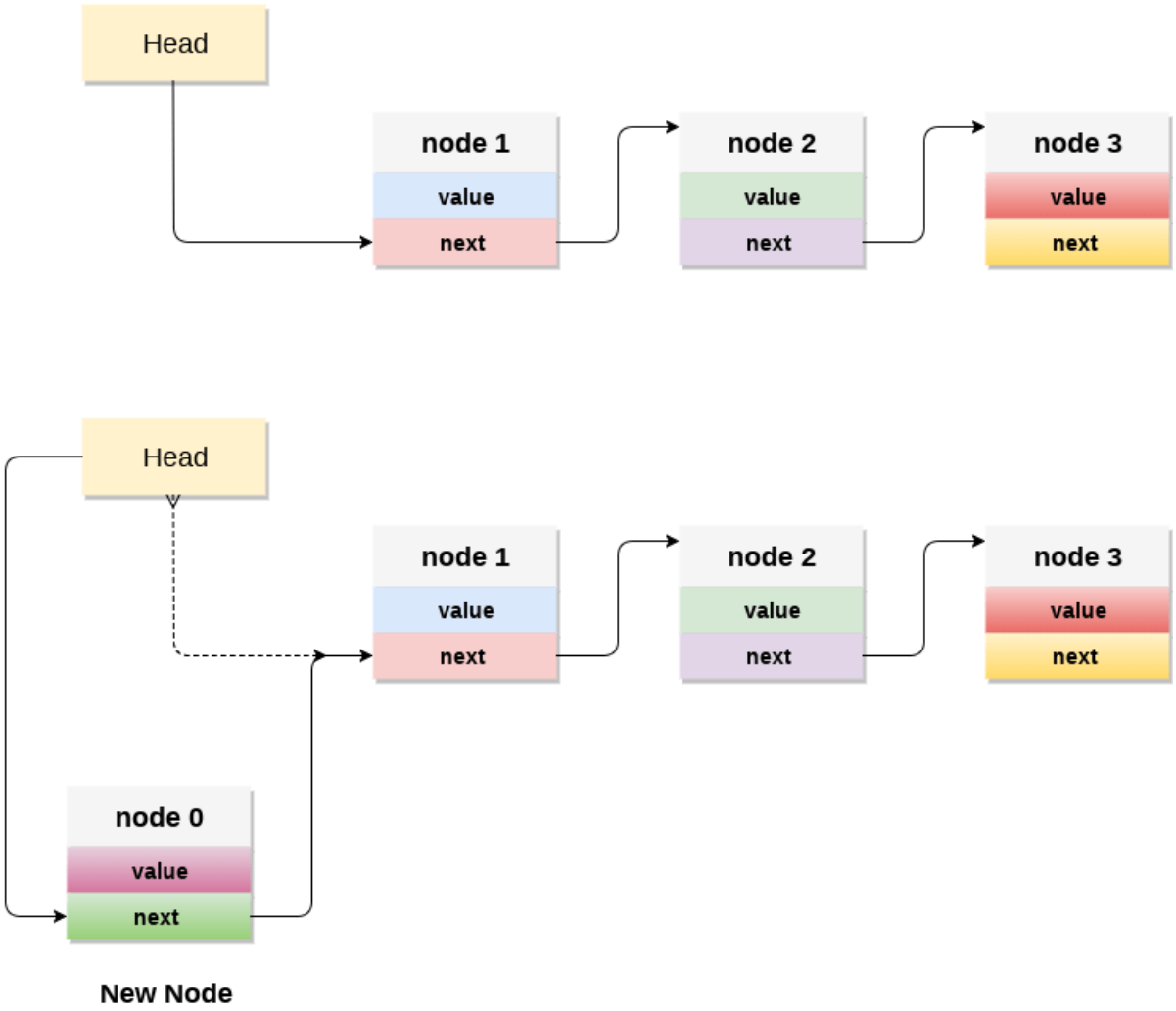


Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

   Time Complexity: o (1)



Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps:

1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity: o (n)

Display the nodes (Traversing)

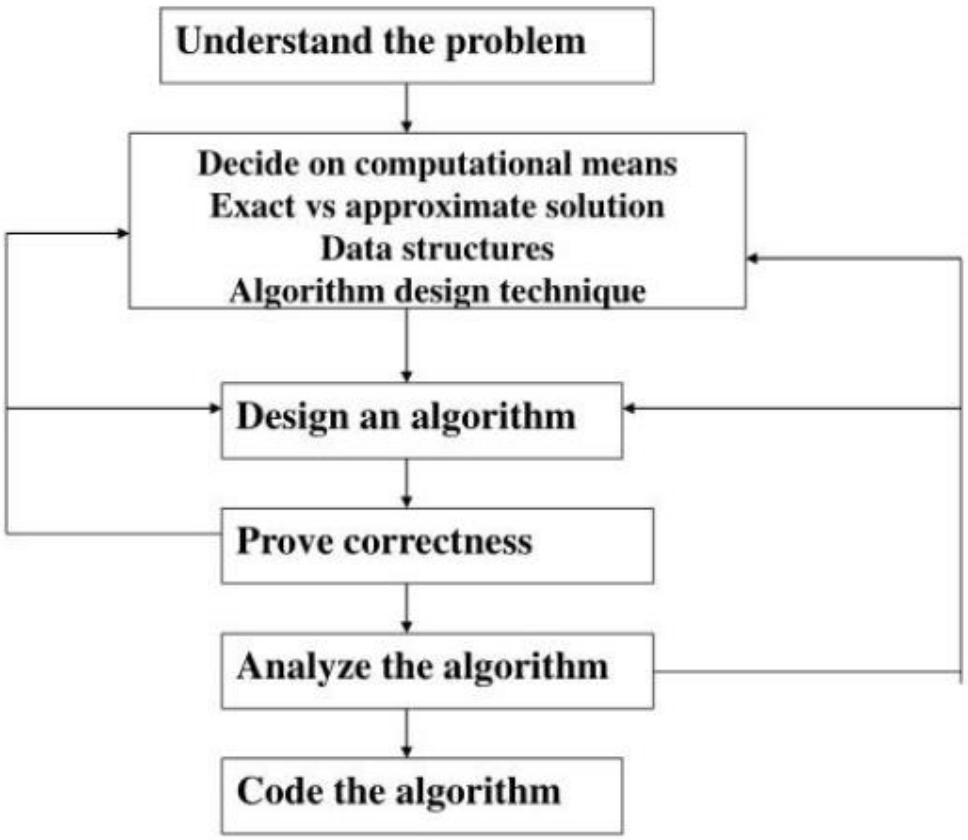
Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

Time Complexity: o (n)

Q 7 a. Ans:

Steps in Designing and Implementing an Algorithm



What does it mean to understand the problem?

- What are the problem objects?
- What are the operations applied to the objects?

Deciding on computational means

- How the objects would be represented?
- How the operations would be implemented?

Design an algorithm

- **Build a computational model of the solving process**

Prove correctness

- **Correct output for every legitimate input in finite time**
- **Based on correct math formula**
- **By induction**

Q7b Ans:

Important problem types

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

10

Sorting

The ***sorting problem*** is to rearrange the items of a given list in nondecreasing order. Of course, for this problem to be meaningful, the nature of the list items must allow such an ordering. (Mathematicians would say that there must exist a relation of total ordering.) As a practical matter, we usually need to sort lists of numbers, characters from an alphabet, character strings, and, most important, records similar to those maintained by schools about their students, libraries about their holdings, and companies about their employees. In the case of records, we need to choose a piece of information to guide sorting. For example, we can choose to sort student records in alphabetical order of names or by student number or by student grade-point average. Such a specially chosen piece of information is called a ***key***. Computer scientists often talk about sorting a list of keys even when the list's items are not records but, say, just integers.

Why would we want a sorted list? To begin with, a sorted list can be a required output of a task such as ranking Internet search results or ranking students by their GPA scores. Further, sorting makes many questions about the list easier to answer. The most important of them is searching: it is why dictionaries, telephone books, class lists, and so on are sorted. You will see other examples of the usefulness of list presorting in Section 6.1. In a similar vein, sorting is used as an auxiliary step in several important algorithms in other areas, e.g., geometric algorithms and data compression. The greedy approach—an important algorithm design technique discussed later in the book—requires a sorted input.

Searching

The ***searching problem*** deals with finding a given value, called a ***search key***, in a given set (or a multiset, which permits several elements to have the same value). There are plenty of searching algorithms to choose from. They range from the straightforward sequential search to a spectacularly efficient but limited binary search and algorithms based on representing the underlying set in a different form more conducive to searching. The latter algorithms are of particular importance for real-world applications because they are indispensable for storing and retrieving information from large databases.

For searching, too, there is no single algorithm that fits all situations best. Some algorithms work faster than others but require more memory; some are very fast but applicable only to sorted arrays; and so on. Unlike with sorting algorithms, there is no stability problem, but different issues arise. Specifically, in applications where the underlying data may change frequently relative to the number of searches, searching has to be considered in conjunction with two other operations: an addition to and deletion from the data set of an item. In such situations, data structures and algorithms should be chosen to strike a balance among the requirements of each operation. Also, organizing very large data sets for efficient searching poses special challenges with important implications for real-world applications.

Q8a Ans: An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. It is Unambiguous, step by step procedure to solve a given problem in finite number of steps by accepting a set of inputs and producing the desired output.

A way of comparing functions that ignores constant factors and small input sizes.

O(g(n)): set of functions t(n) that grow no faster than g(n)

Ω(g(n)): set of functions t(n) that grow at least as fast as g(n)

Θ(g(n)): set of functions t(n) that grow at same rate as g(n)

Formal Definition of Big-Oh O

A function t(n) is said to be in O(g(n)), denoted as $t(n) \in O(g(n))$, if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that

$t(n) \leq cg(n)$ for all $n \geq n0$.

// g(n) is upper bound of t(n)

Examples: n ∈ O(n 2) and 100n + 5 ∈ O(n 2), n(n - 1)/2 ∈ O(n 2)

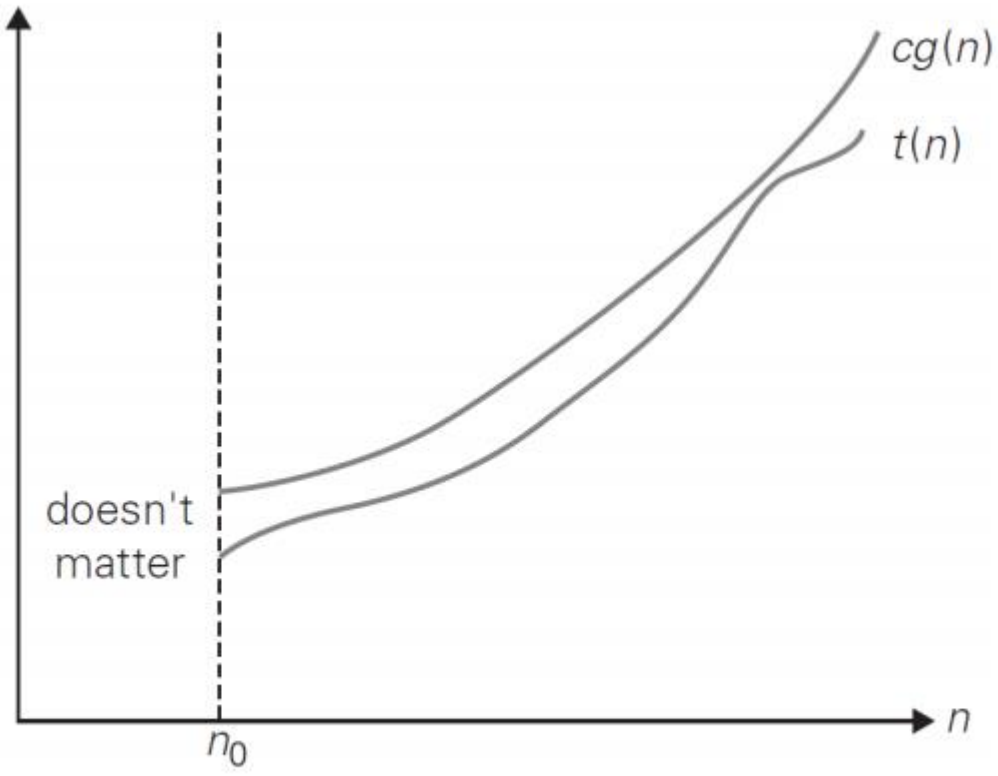


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$.

Formal Definition of Big-Omega Ω

A function t(n) is said to be in Ω(g(n)), denoted as t(n) ∈ Ω(g(n)), if t(n) is bounded below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that

t(n) ≥ cg(n) for all n ≥ n0 .

// g(n) is lower bound of t(n)

Examples: n 3 ∈ Ω(n 2), n(n – 1)/2 ∈ Ω(n 2)

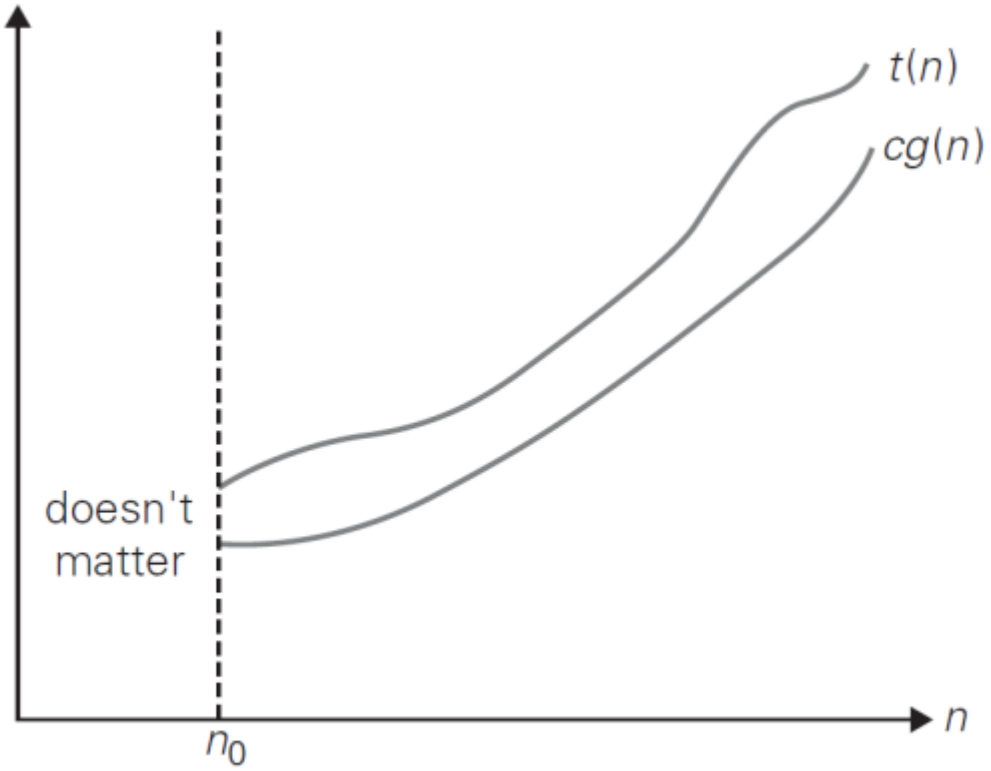


Figure 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$.

Formal Definition of Big-Theta Θ

A function t(n) is said to be in Θ(g(n)), denoted as t(n) ∈ Θ(g(n)), if t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that

c2g(n) ≤ t(n) ≤ c1g(n) for all n ≥ n0 .

Examples: n(n – 1)/2 ∈ Θ(n 2)

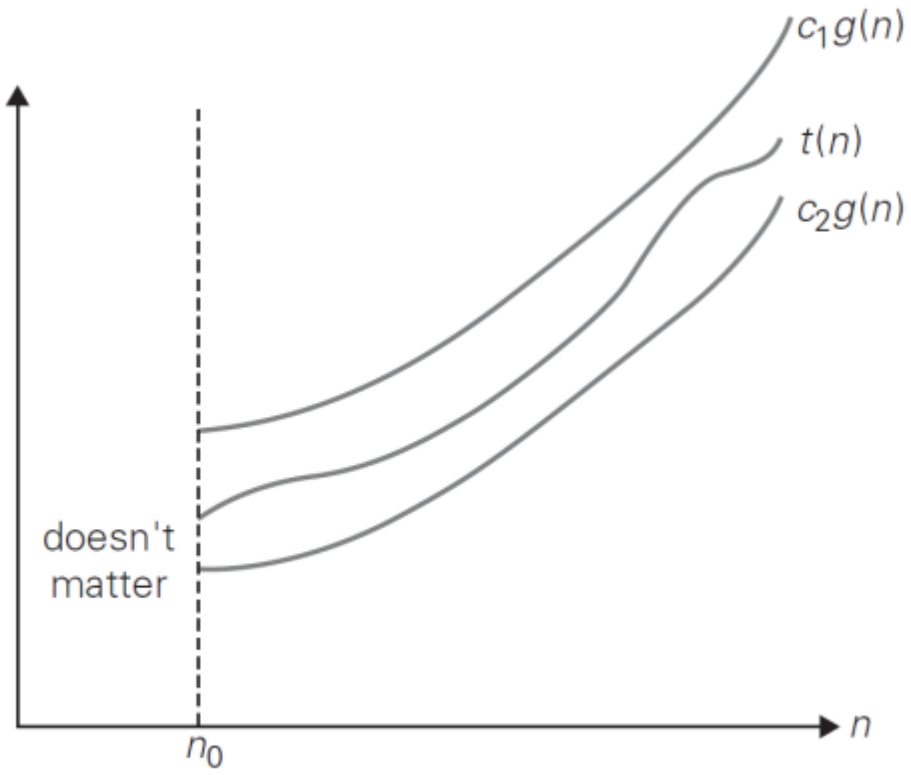


Figure 2.3 Big-omega notation: $t(n) \in \Theta(g(n))$.

Q8b Ans:

PROOF The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■

So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part:

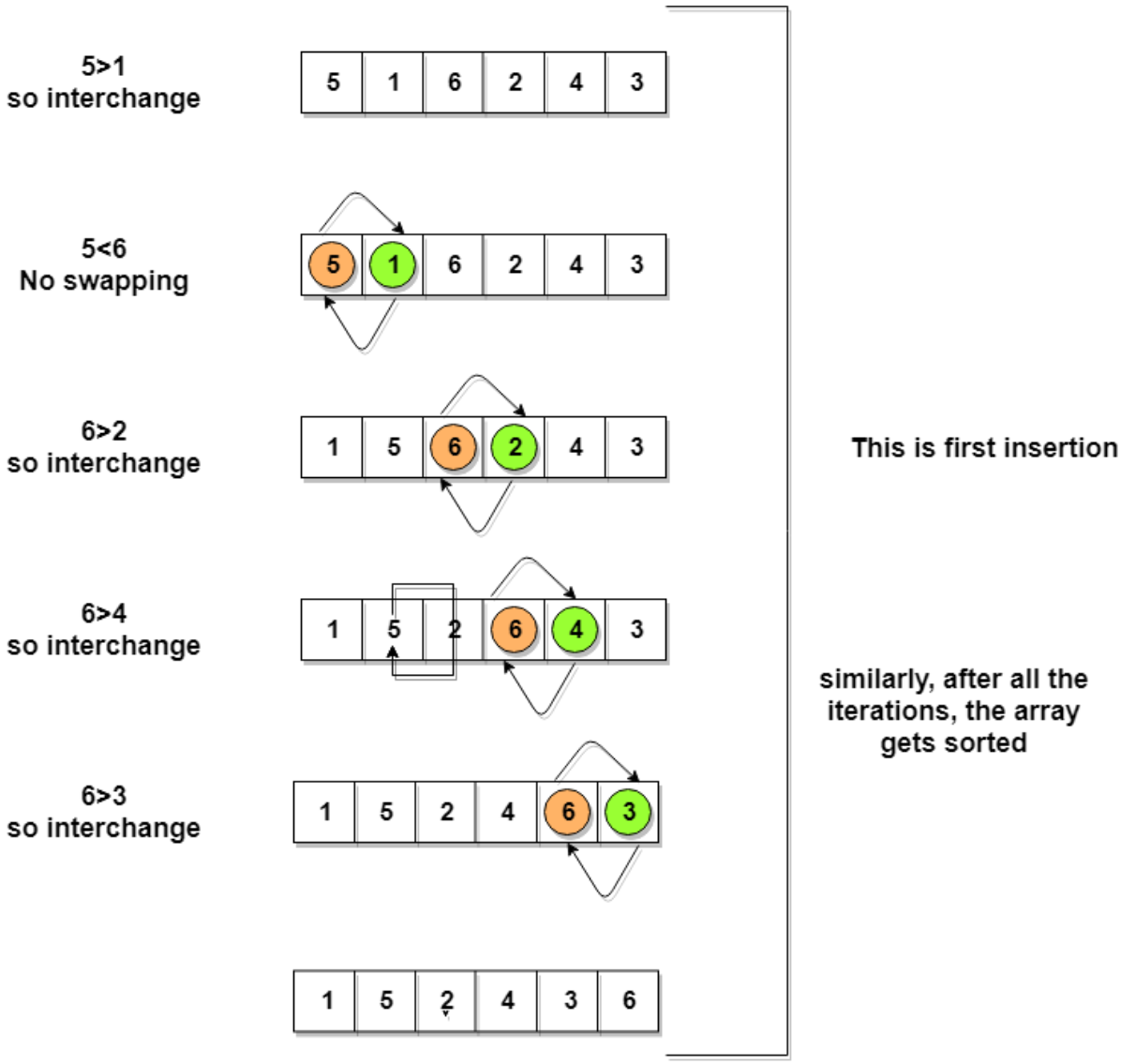
$$\left.\begin{array}{l} t_1(n) \in O(g_1(n)) \\ t_2(n) \in O(g_2(n)) \end{array}\right\} \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

Q9a Ans: Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Let's consider an array with values {5, 1, 6, 2, 4, 3}

Below, we have a pictorial representation of how bubble sort will sort the given array.



So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.

Similarly after the second iteration, 5 will be at the second last index, and so on.

Complexity Analysis of Bubble Sort

In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,

```
(n-1) + (n-2) + (n-3) + ..... + 3 + 2 + 1

Sum = n(n-1)/2

i.e O(n²)
```

Hence the **time complexity** of Bubble Sort is **O(n²)**.

The main advantage of Bubble Sort is the simplicity of the algorithm.

The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for temp variable.

Also, the **best case time complexity** will be **O(n)**, it is when the list is already sorted.

Q9b Ans:

String Matching –Program

The simplest algorithm for string matching is a brute force algorithm, where we simply try to match the first character of the pattern with the first character of the text, and if we succeed, try to match the second character, and so on; if we hit a failure point, slide the pattern over one character and try again. When we find a match, return its starting location. Java code for the brute force method:

```
 for (int i = 0; i < n-m; i++)

{

int j = 0;

while (j < m && t[i+j] == p[j])

 { j++; }

 if (j == m) return i;

}

System.out.println(‘‘No match found’’);

 return -1;
```

The outer loop is executed at most n-m+1 times, and the inner loop m times, for each iteration of the outer loop. Therefore, the running time of this algorithm is in O(nm).

```
#include <stdio.h>

#include<string.h>

int SM(char s[], char p[])
```

```
{ int i,j;

   for(i=0;i<strlen(s)-strlen(p);i++)
   {
      j=0;
      while(j<strlen(p) && p[j]==s[i+j])
      j=j+1;

      if (j==strlen(p))
      return i;
   }


      return -1;
}
int main()
{
   char s[50], p[10];
   int i;

   printf("Enter String");
   scanf("%s", s);
   printf("Enter pattern");
   scanf("%s", p);
  i=  SM(s,p);
  if(i==-1)
  printf("Pattern not found");
  else
  printf("Pattern found at position %d",i);

   return 0;
```

}

Q10 a Ans:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

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

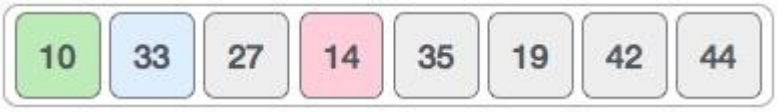
Consider the following depicted array as an example.

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

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

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

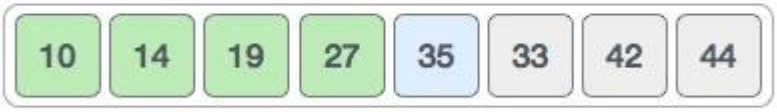
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

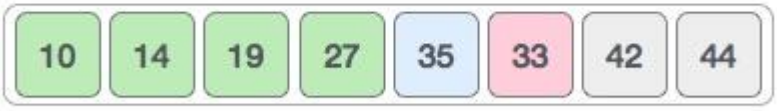


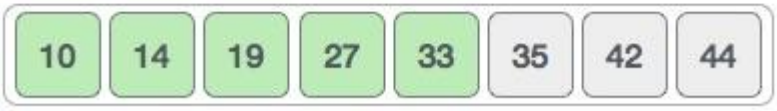
After two iterations, two least values are positioned at the beginning in a sorted manner.

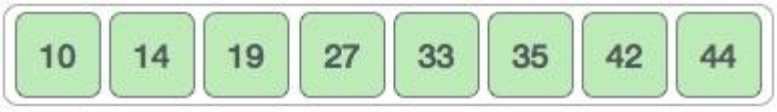
The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −









This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

Q10 b Ans:

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.

Algorithm

- **Step 1:** SET STATUS = 1 (ready state) for each node in G
- **Step 2:** Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- **Step 3:** Repeat Steps 4 and 5 until STACK is empty
- **Step 4:** Pop the top node N. Process it and set its STATUS = 3 (processed state)
- **Step 5:** Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state) [END OF LOOP]
- **Step 6:** EXIT

Tracing of Example

c d b a e f g