Course Curriculum for Data Structures Placement Preparation.

TOPIC 1: ESSENTIALS OF DATA STRUCTURE.

1.1 Introduction to pointer

1.2 Pointer and Array

1.3 Pointer and String.

1.4 Pointer and Structure.

1.5 Algorithm Complexity

1.6 Asymptotic Notation.

1.7 Recursion

1.8 Short Type Questions and Answers on above topics.

1.9 Three essential and two advance Quiz (20 Questions in each quiz) from above topics.

TOPIC 2: LINKED LIST.

2.1 Single Linked List

2.2 Doubly Linked

List 2.3 Circular

Linked List.

2.4 Doubly Circular Linked List

2.5 Application of Linked Lists

2.6 Short Type Questions and Answers on above topics

2.7 Three essential and two advance Quiz (20 Questions in each quiz) from above topics.

TOPIC 3: STACK.

3.1 Representing a stack using an array and Linked List.

3.3 Applications of Stacks.

3.3 Short Type Questions and Answers on above topics

3.4 Three essential and two advance Quiz (20 Questions in each quiz) from above topics.

TOPIC 4: QUEUE.

4.1 Operations on Queue using Array and Linked List.

4.2 Application of Queue

4.3 De-queue (Double Ended Queue)

4.4 Priority Queue

4.5 Circular Queue

4.6 Doubly Circular Queue.

4.5 Short Type Questions and Answers on above topics

4.6 Three essential and two advance Quiz (20 Questions in each quiz) from above topics.

TOPIC 5: TREE

5.1 Tree Terminology, types, and representation.

5.2 Binary tree with Properties

5.3 Binary Search Tree

5.4 AVL Tree

5.5 Threaded Binary Tree.

5.6 B Tree
5.7 B+ Tree
5.8 Heap Tree
5.9 Short Type Questions and Answers on above topics
5.10 Three essential and two advance Quiz (20 Questions in each quiz) from above topics.

TOPIC 6: GRAPH.

6.1 Graph Terminology.
6.2 Graph Representation (Adjacency matrix, Adjacency list and Adjacency Multi list)
6.3 Graph Traversal
6.4 Application of Graph
Topological sort
Minimum spanning tree
Finding shortest paths
Shortest path for given source and destination (Single Source Shortest Path)
Shortest path among all-pair of vertices (All pair Shortest Path)
Euler Graph
Hamiltonian Graph
6.5 Short Type Questions and Answers on above topics
6.6 Three essential and two advance Quiz (20 Questions in each quiz) from above topics.

TOPIC 7: SEARCHING.

7.1 Introduction
7.2 Basic Searching Techniques
Algorithmic Notation
Sequential or Linear search
Indexed Sequential search
Binary search.
7.3 Short Type Questions and Answers on above topics
7.4 Three essential and two advance Quiz (20 Questions in each quiz) from above topics.

TOPIC 8: SORTING.

8.1 Introduction to sorting.
8.2 Classification of sorting
Internal Sorting, External Sorting,
Stable Sorting, In place sorting, Not In place sorting
Comparison based sorting, Counting based sorting.
8.3 Types of Sorting
Bubble, Selection, Insertion, Quick, Merge, Heap sort, Radix sort, Shell sort
(Revision of time complexity of each sorting algorithm)
8.4 Short Type Questions and Answers on above topics
8.5 Three essential and two advance Quiz (20 Questions in each quiz) from above topics.

TOPIC 9: HASH TABLE AND HASHING.

9.1 Introduction to Hashing

9.2 Direct Address table

9.3 Hash Function

9.4 Collision Resolution Techniques.

9.5 Short Type Questions and Answers on above topics

9.6 Three essential and two advance Quiz (20 Questions in each quiz) from above topics.

Topic wise contribution by faculty member.

Sr. No	Faculty Name	Topic
1	Prof. Shweta Rani	1,2
2	Prof. Arti Sharma	3,4
3	Prof. Vivek Tomar	5
4	Prof. Umang Rastogi	6,9
5	Prof. Ansula	7,8
6	Prof. Preeti Garg	Moderation Team
7	Prof. Hriday Kumar Gupta	Moderation Team

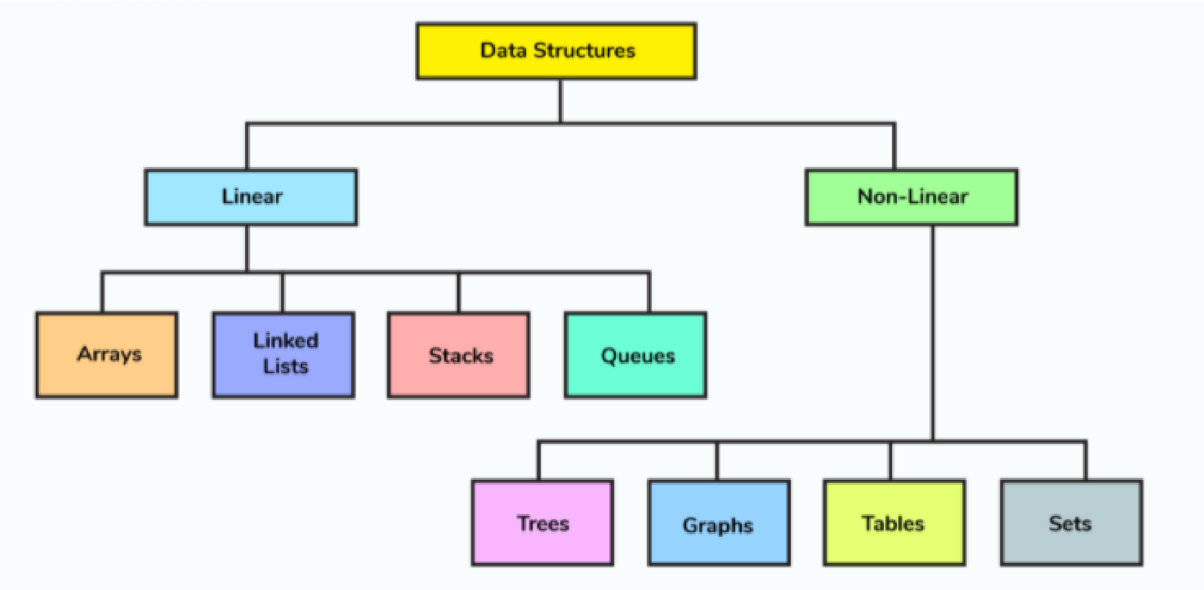
CHAPTER-1

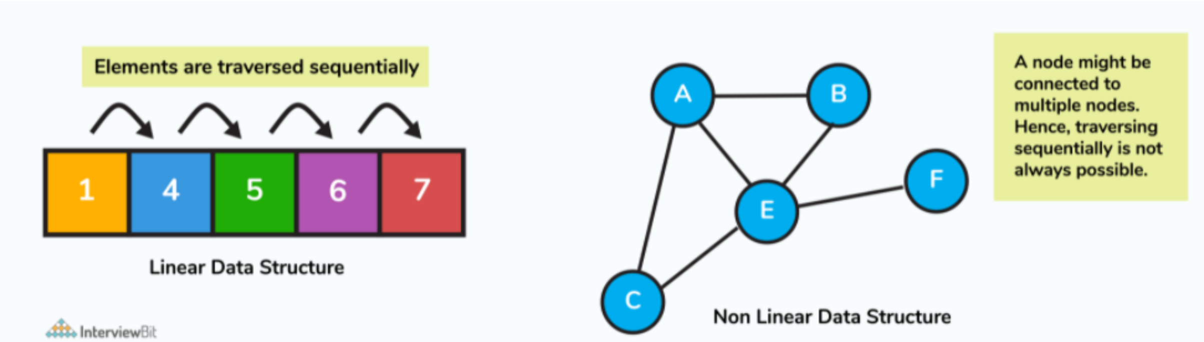
TOPIC 1: ESSENTIALS OF DATA STRUCTURE.

What is a Data Structure?

A data structure is a way of organizing the data so that the data can be used efficiently.

- **Linear:** A data structure is said to be linear if its elements form a sequence or a linear list. Examples: Array. Linked List, Stacks and Queues
- **Non-Linear:** A data structure is said to be non-linear if the traversal of nodes is nonlinear in nature. Example: Graph and Trees.

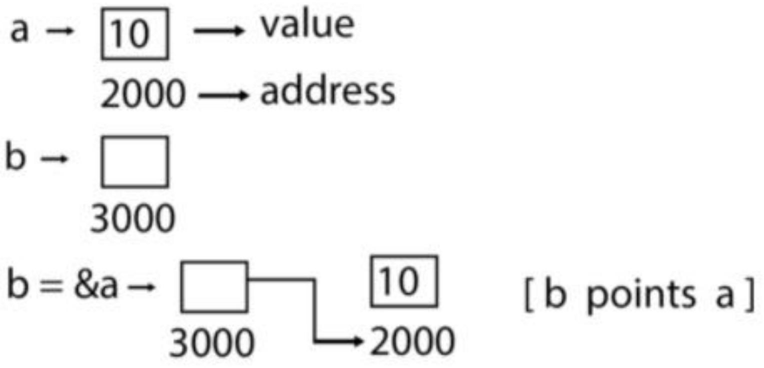



Elements are traversed sequentially
1
4
5
6
7
Linear Data Structure
InterviewBit
A
B
E
F
C
Non Linear Data Structure
A node might be connected to multiple nodes. Hence, traversing sequentially is not always possible.

1.1 Introduction to pointer

A pointer is a variable that refers to the address of a value. It makes the code optimized and makes the performance fast. Whenever a variable is declared inside a program, then the system allocates some memory to a variable. The memory contains some address numbers. The variables that hold this address number is known as the pointer variable.

int *p;



Simple program to understand how pointer works

```
#include <stdio.h>
int main( )
{
int a = 5;
int *b;
b = &a;
 printf ("value of a = %d\n", a);
printf ("value of a = %d\n", *(&a));
printf ("value of a = %d\n", *b);
printf ("address of a = %d\n", b);
printf ("address of b = %u\n", &b);
printf ("value of b = address of a = %u", b);
return 0;
}
```

Output

```
value of a = 5
value of a = 5
address of a = 3010494292
address of b = 3010494296
value of b = address of a = 3010494292
```

%p format specifier

It's purpose is to print a pointer value in an implementation defined format. The corresponding argument must be a void * value.

And %p is used to printing the address of a pointer the addresses are depending by our system bit.

Interview Questions:

What are valid operations on pointers?

<u>Answer:</u>

- Assignment of pointers to the same type of pointers.
- Adding or subtracting a pointer and an integer.
- subtracting or comparing two-pointer.
- incrementing or decrementing the pointers pointing to the elements of an array. When a pointer to an integer is incremented by one, the address is incremented by two. It is done automatically by the compiler.
- Assigning the value 0 to the pointer variable and comparing 0 with the pointer. The pointer having address 0 points to nowhere at all.

What is the usage of the pointer in C?
Answer:

- Accessing array elements: Pointers are used in traversing through an array of integers and strings. The string is an array of characters which is terminated by a null character '\0'.
- Dynamic memory allocation: Pointers are used in allocation and deallocation of memory during the execution of a program.
- Call by Reference: The pointers are used to pass a reference of a variable to other function.
- Data Structures like a tree, graph, linked list, etc.: The pointers are used to construct different data structures like tree, graph, linked list, etc.

What is a NULL pointer in C?
Answer: A pointer that doesn't refer to any address of value but NULL is known as a NULL pointer. When we assign a '0' value to a pointer of any type, then it becomes a Null pointer.

What is dangling pointer in C?
Answer: A **Pointer** in C Programming is used to point the memory location of an existing variable. In case if that particular variable is deleted and the Pointer is still pointing to the same memory location, then that particular pointer variable is called as a **Dangling Pointer Variable.**

What is pointer to pointer in C?
Answer: In case of a pointer to pointer concept, one pointer refers to the address of another pointer. The pointer to pointer is a chain of pointers. Generally, the pointer contains the address of a variable. The pointer to pointer contains the address of a first pointer.

To make pointer generic for which data type needs to be declared?
Answer: void

Can Math Operations Be Performed On A Void Pointer?
Answer: No. Pointer addition and subtraction are based on advancing the pointer by a number of elements. By definition, if you have a void pointer, you don't know what it's pointing to, so you don't know the size of what it's pointing to. If you want pointer arithmetic to work on raw addresses, use character pointers.

What Is Indirection?
Answer: If you declare a variable, its name is a direct reference to its value. If you have a pointer to a variable or any other object in memory, you have an indirect reference to its value.

How Are Pointer Variables Initialized?
Answer: Pointer variable are initialized by one of the following two ways:

1. Static memory allocation
2. Dynamic memory allocation

How is the null pointer different from a void pointer?

Answer: A pointer is initialized as NULL when its value isn't known at the time of declaration. Generally, NULL pointers do not point to a valid location. Unlike NULL pointers, void pointers are general-purpose pointers that do not have any data type associated with them. Void pointers can contain the address of any type of variable. So, the data type that a void pointer points to can be anything.

In what data structures are pointers applied?

Answer: Pointers that are used in the linked list have various applications in the data structure. Data structures that make use of this concept include the Stack, Queue, Linked List and Binary Tree

1.2 Pointer and Array

There may be a situation when we want to maintain an array, which can store pointers to an int or char or any other data type available. It declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, holds a pointer to an int value

```
int *ptr[MAX];  // declaration of an array of pointers
```

```
#include <stdio.h>
const int MAX = 3;
int main () {
   int  var[] = {10, 100, 200};
   int i, *ptr[MAX];
    for ( i = 0; i < MAX; i++) {
      ptr[i] = &var[i]; /* assign the address of integer. */
   }
   for ( i = 0; i < MAX; i++) {
      printf("Value of var[%d] = %d\n", i, *ptr[i] );
   }
    return 0;
}
```

```
Output: Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

Example:

```
int arr[5] = { 1, 2, 3, 4, 5 };
int *ptr = arr;
```

ptr that points to the 0th element of the array

Consider a compiler where int takes 4 bytes, char takes 1 byte and pointer takes 4 bytes.

```
#include <stdio.h>

int main()
{
    int arri[] = {1, 2 ,3};
    int *ptri = arri;

    char arrc[] = {1, 2 ,3};
    char *ptrc = arrc;

    printf("sizeof arri[] = %d ", sizeof(arri));
    printf("sizeof ptri = %d ", sizeof(ptri));

    printf("sizeof arrc[] = %d ", sizeof(arrc));
    printf("sizeof ptrc = %d ", sizeof(ptrc));

    return 0;
}
OutPut:
sizeof arri[] = 12 sizeof ptri = 4 sizeof arrc[] = 3 sizeof ptrc = 4
```

Increment: It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.
For Example:
If an integer pointer that stores **address 1000** is incremented, then it will increment by 2(**size of an int**) and the new address it will points to **1002**. While if a float type pointer is incremented then it will increment by 4(**size of a float**) and the new address will be **1004**.
Decrement: It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.
For Example:
If an integer pointer that stores **address 1000** is decremented, then it will decrement by 2(**size of an int**) and the new address it will points to **998**. While if a float type pointer is decremented then it will decrement by 4(**size of a float**) and the new address will be **996**.

Assume that float takes 4 bytes, predict the output of following program.

```
#include <stdio.h>

int main()
{
   float arr[5] = {12.5, 10.0, 13.5, 90.5, 0.5};
   float *ptr1 = &arr[0];
   float *ptr2 = ptr1 + 3;

   printf("%f ", *ptr2);
   printf("%d", ptr2 - ptr1);

  return 0;
}
Output: 90.500000   3
```

SHORT QUESTION ANSWER

Q) What does it mean when a pointer is used in an if statement?

It is good habits to check the pointer in if condition before using it. It prevents code crashing. A pointer can be used in an if, while, for, or do/while statement, or in any conditional expression.

```
if ( p )
{
/*Run when valid address */
}
else
{
/*When NULL pointer*/
}
```

Differentiate Between Arrays And Pointers?

Answer: Pointers are used to manipulate data using the address. Pointers use * operator to access the data pointed to by them Arrays use subscripted variables to access and manipulate data. Array variables can be equivalently written using pointer expression.

What is an array of pointers?

Answer: If the elements of an array are addresses, such an array is called an array of pointers.

Represent a two-dimensional array using pointer?

Address of a[I][j] Value of a[I][j]
&a[I][j]
or
a[I] + j
or
*(a+I) + j
*&a[I][j] or a[I][j]
or
*(a[I] + j)
or
*(* (a+I) +j)

Difference between an array of pointers and a pointer to an array

Array of pointers

- Declaration is: data_type *array_name[size];
- Size represents the row size.
- The space for columns may be dynamically

Pointers to an array

- Declaration is data_type (*array_name)[size];
- Size represents the column size.

Are the expressions *ptr ++ and ++ *ptr same?

Answer: No, *ptr ++ increments pointer and not the value pointed by it. Whereas ++ *ptr increments the value being pointed to by ptr.

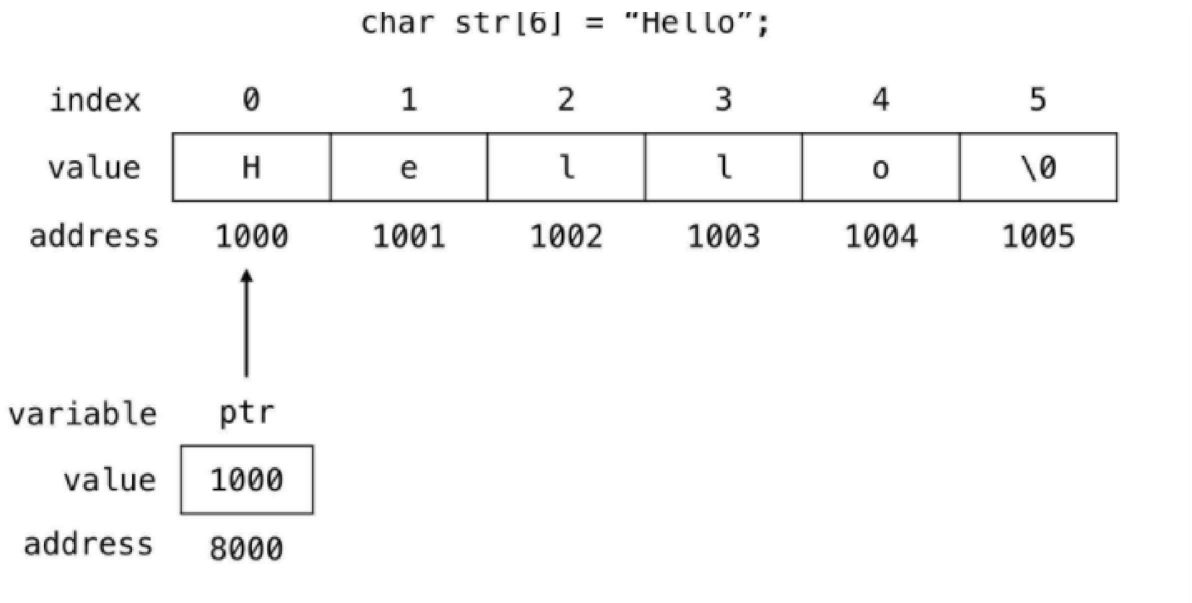
```
#include <stdio.h>
int main(void)
{
int aiData[5] = {100,200,300,400,500};
int *piData = &aiData;
++*piData;
printf("%d %d %d \n ”, aiData[0], aiData[1], *piData);
printf("%d”, *(piData+2));

return 0;
}
```

Output: 101 , 200 , 101
300

1.3 Pointer and String

A String is a sequence of characters stored in an array. A string always ends with null ('\0') character. Simply a group of characters forms a string and a group of strings form a sentence.

```
char *ptr = str;
```



The pointer variable ptr is allocated memory address 8000 and it holds the address of the string variable str i.e., 1000.

```
#include <stdio.h>  // pointer to string
int main()
{
char *cities[] = {"Iran", "Iraq"};
int i;
for(i = 0; i < 2; i++)
printf("%s\n", cities[i]);
return 0;
}
```

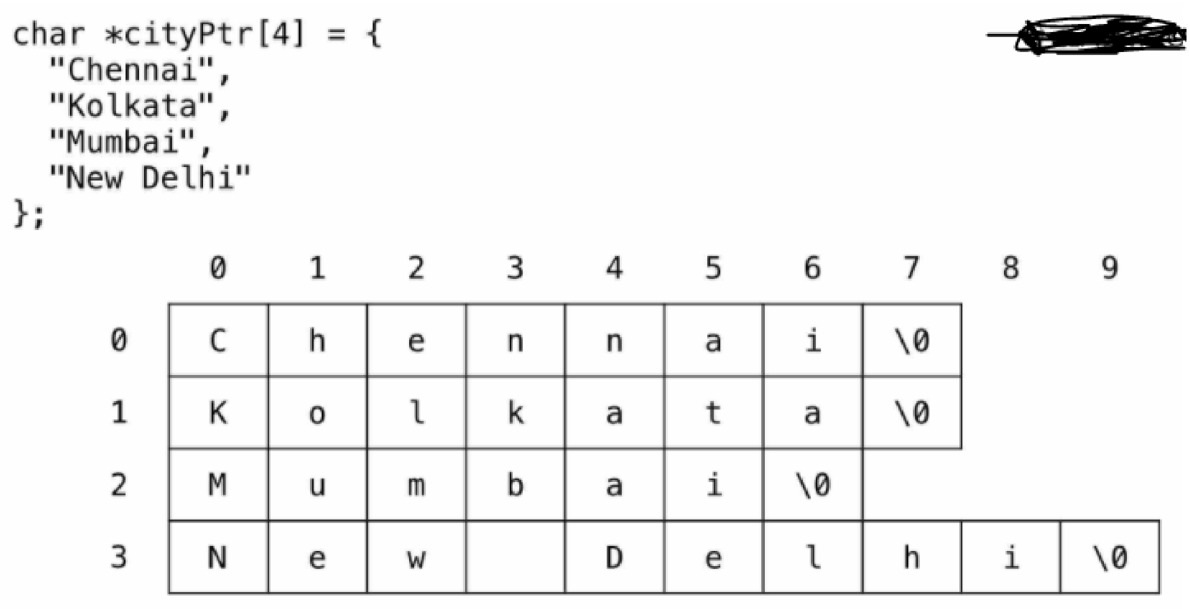
Output :

```
Iran
Iraq
```

```
// pointer to string
#include <stdio.h>
#include <string.h>
void function(char**);
int main()
{
char *str = "Pointer-to-string";
int i, j = strlen(str);
for(i = 0; i < j; i++)
printf("%c", *str++);
return 0;
}
```

Output: Pointer-to-string

We can represent the array of pointers as follows.

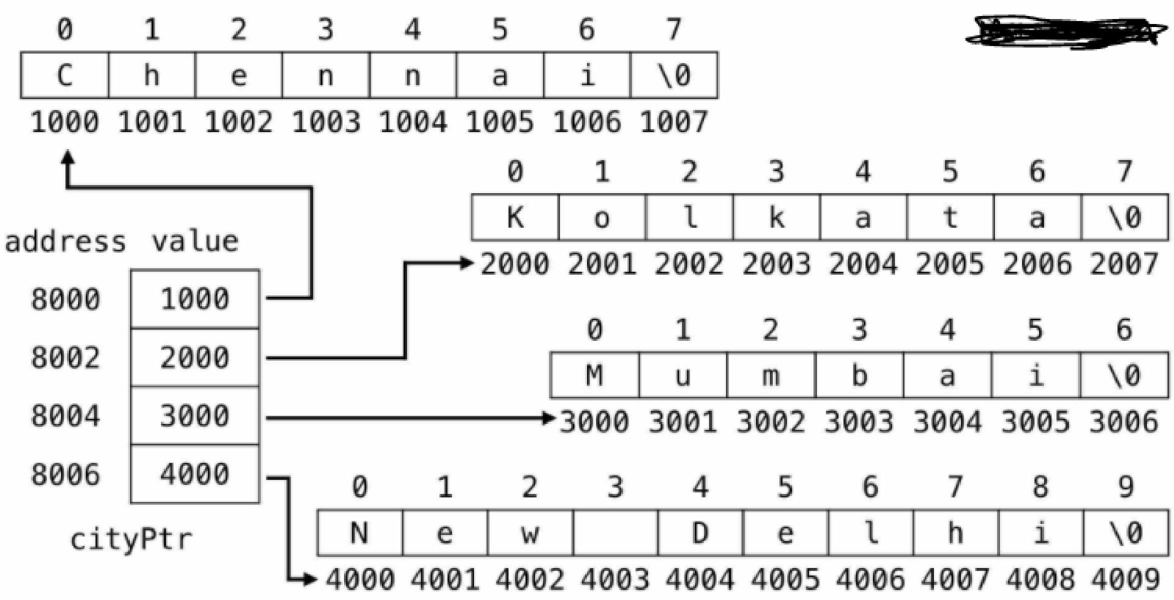


[redacted]

```
char *cityPtr[4] = {
  "Chennai",
  "Kolkata",
  "Mumbai",
  "New Delhi"
};
```

	0	1	2	3	4	5	6	7	8	9
0	C	h	e	n	n	a	i	\0		
1	K	o	l	k	a	t	a	\0		
2	M	u	m	b	a	i	\0			
3	N	e	w		D	e	l	h	i	\0

The above array of pointers can be represented in memory as follows.



[redacted]

0	1	2	3	4	5	6	7
C	h	e	n	n	a	i	\0
1000	1001	1002	1003	1004	1005	1006	1007

0	1	2	3	4	5	6	7
K	o	l	k	a	t	a	\0
2000	2001	2002	2003	2004	2005	2006	2007

address	value
8000	1000
8002	2000
8004	3000
8006	4000

cityPtr

0	1	2	3	4	5	6
M	u	m	b	a	i	\0
3000	3001	3002	3003	3004	3005	3006

0	1	2	3	4	5	6	7	8	9
N	e	w		D	e	l	h	i	\0
4000	4001	4002	4003	4004	4005	4006	4007	4008	4009

To print the cities:

```
int r, c;
  // print cities
  for (r = 0; r < 4; r++) {
    c = 0;
    while(*(cityPtr[r] + c) != '\0') {
      printf("%c", *(cityPtr[r] + c));
      c++;
    }
    printf("\n");
    }
```

What are the advantages of using array of pointers to string instead of an array of strings?

1. i) Efficient use of memory.
2. ii) Easier to exchange the strings by moving their pointers while sorting.

State the equivalent pointer expression which refer the given array element a[i][j][k][l]?

Ans. Above pointer expression can be written as *(*(*(*(a+i)+j)+k)+l).

1.4 Pointer and Structure.

```
struct name {
    member1;
    member2;
    .
    .
};

int main()
{
    struct name *ptr;
}
```

Here, ptr is a pointer to struct.

Access Members of a structure using pointers

```
#include <stdio.h>
struct person
{
   int age;
   float weight;
};
int main()
{
  struct person = person1;      //person1 is an instance of structure person
    struct person *personPtr;   // personPtr is the  structure pointers , stores address of person struct
    personPtr = &person1;

    printf("Enter age: ");
    scanf("%d", &personPtr->age);

    printf("Enter weight: ");
    scanf("%f", &personPtr->weight);

    printf("Displaying:\n");
    printf("Age: %d\n", personPtr->age);
    printf("weight: %f", personPtr->weight);
}
```

In this example, the address of person1 is stored in the personPtr pointer using personPtr = &person1;.
Now, you can access the members of person1 using the personPtr pointer.
By the way,
personPtr->age is equivalent to (*personPtr).age
personPtr->weight is equivalent to (*personPtr).weight

Output:

```
Enter age: 12
Enter weight: 34
Displaying:
Age: 12
weight: 34.000000
```

Interview Questions:

1.5 Algorithm Complexity

1.6 Asymptotic Notation.

1.7 Recursion

What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.
Example: add N natural numbers:

approach(2) – Recursive adding

f(n) = 1	***n=1***
f(n) = n + f(n-1)	***n>1***

What is base condition in recursion?
In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
int fact(int n) // recursive program for factorial
{
   if (n < = 1) // base case
      return 1;
   else
```

```
    return n*fact(n-1);      //recursive case
}
```

What is difference between tailed and non-tailed recursion?

A recursive function is tail recursive when recursive call is the last thing executed by the function.

How memory is allocated to different function calls in recursion?

When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copy of local variables is created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

What are the disadvantages of recursive programming over iterative programming?

The recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

What are the advantages of recursive programming over iterative programming?

Recursion provides a clean and simple way to write code.

Quiz Questions for Topic 1

```
void fun(int *ptr)
{
    *ptr = 30;
}

int main()
{
  int y = 20;
  fun(&y);
  printf("%d", y);

  return 0;
}
```

(A) 20
(B) 30
(C) Compiler error
(D) Runtime error

Answer: B

What will be output of following program?

```
#include<stdio.h>
int main() {
        int a = 10;
        void *p = &a;
        int *ptr = p;
        printf("%u",*ptr);
        return 0;
}
```

A. **11**
B. **10**
C. **Garbage value**
D. **None of the above**

Answer: B

Predict output of following program

```
#include <stdio.h>

int fun(int n)
{
    if (n == 4)
       return n;
    else return 2*fun(n+1);
}

int main()
{
   printf("%d ", fun(2));
   return 0;
}
```

(A) 4 (B) 8 (C) 16 (D) Runtime error

Answer: C

What is the output of the following program?

```
#include<stdio.h>
main()
```

```
{
   int x = 65, *p = &x;
  void *q=p;
   char *r=q;
   printf("%c",*r);
}
```

A - Garbage character.

B - A

C - 65

D - Compile error

Answer: B

What is the output of the following program?

```
#include<stdio.h>
main()
{
   int a[] = {1,2}, *p = a;
   printf("%d", p[1]);
}
```

A - 1

B - 2

C - Compile error

D - Runtime error

Answer: B

Consider the following recursive function fun(x, y). What is the value of fun(4, 3)

```
int fun(int x, int y)
{
  if (x == 0)
    return y;
  return fun(x - 1,  x + y);
}
```

(A) 13 (B) 12 (C) 9 (D) 10

Answer: A

What does the following function print for n = 25?

```
void fun(int n)
{
  if (n == 0)
    return;
  printf("%d", n%2);
  fun(n/2);
}
```

(A) 11001 (B) 10011 (C) 11111 (D) 00000

Answer: B

What does the following function do?

```
int fun(int x, int y)
{
    if (y == 0)   return 0;
    return (x + fun(x, y-1));
}
```

(A) x+y (B) x+x*y (C) x*y (D) x^y

Answer: C

Assume that float takes 4 bytes, predict the output of following program.

```
#include <stdio.h>

int main()
{
    float arr[5] = {12.5, 10.0, 13.5, 90.5, 0.5};
    float *ptr1 = &arr[0];
    float *ptr2 = ptr1 + 3;

    printf("%f ", *ptr2);
    printf("%d", ptr2 - ptr1);

   return 0;
}
```

A. 90.500000   3

B. 90.500000 12

C. 10.000000 12

D. 0.5000000   3

Answer: A

What does fun2() do in general?

```
int fun(int x, int y)
{
    if (y == 0)   return 0;
    return (x + fun(x, y-1));
}

int fun2(int a, int b)
{
    if (b == 0) return 1;
    return fun(a, fun2(a, b-1));
}
```

(A) y^x (B) x+x*y (C) x*y (D) x^y

Answer: D

```
#include<stdio.h>
int main()
{
    int arr[] = {10, 20, 30, 40, 50, 60};
    int *ptr1 = arr;
    int *ptr2 = arr + 5;
    printf("Number of elements between two pointer are: %d.",
                                (ptr2 - ptr1));
    printf("Number of bytes between two pointers are: %d",
                              (char*)ptr2 - (char*) ptr1);
    return 0;
}
```

Assume that an int variable takes 4 bytes and a char variable takes 1 byte

A. **Numbe**r of elements between two pointer are: 5. Number of bytes between two pointers are: 20
B. Number of elements between two pointer are: 20. Number of bytes between two pointers are: 20
C. Number of elements between two pointer are: 5. Number of bytes between two pointers are: 5
D. Compiler Error

Answer: A

Output of following program?

```
#include<stdio.h>
void print(int n)
{
    if (n > 4000)
          return;
    printf("%d ", n);
    print(2*n);
    printf("%d ", n);
}
int main()
{
    print(1000);
    getchar();
    return 0;
}
```

A. 1000 2000 4000 4000 2000 1000

B. 1000 2000 4000 2000 1000

C. 1000 2000 4000 2000

D. None of the above

Answer: A

What is the output of program?

```
#include<stdio.h>
int main()
{
   int a;
   char *x;
   x = (char *) &a;
   a = 512;
   x[0] = 1;
   x[1] = 2;
   printf("%d \n",a);
   return 0;
}
```

A. Machine dependent

B. 513

C. 258

D. Compiler Error

Answer: B

```
int main()
{
 char *ptr = "GeeksQuiz";
 printf("%cn", *&*&*ptr);
 return 0;
}
```

A. Compiler Error

B. Garbage Value

C. Runtime Error
D. Gn
Answer: D

Q. The reason for using pointers in a C program is
A. Pointers allow different functions to share and modify their local variables.
B. To pass large structures so that complete copy of the structure can be avoided.
C. Pointers enable complex “linked" data structures like linked lists and binary trees.
D. All of the above
Answer: D

Predict the output of following program

```
#include <stdio.h>
int f(int n)
{
   if(n <= 1)
      return 1;
   if(n%2 == 0)
      return f(n/2);
   return f(n/2) + f(n/2+1);
}
int main()
{
   printf("%d", f(11));
   return 0;
}
```

(A) Stack Overflow, (B) 3 (C) 4 (D) 5
Answer: D

What is time complexity of fun()?

```
int fun(int n)
{
  int count = 0;
  for (int i = n; i > 0; i /= 2)
     for (int j = 0; j < i; j++)
        count += 1;
  return count;
}
```

(A) O(n^2)
(B) O(nLogn)
(C) O(n)
(D) O(nLognLogn)
Answer: C

What is the time complexity of fun()?

```
int fun(int n)
{
  int count = 0;
  for (int i = 0; i < n; i++)
     for (int j = i; j > 0; j--)
        count = count + 1;
  return count;
}
```

(A) Theta(n)
(B) Theta(n^2)
(C) Theta(n*Logn)
(D) Theta(nLognLogn)

Answer: B

Which of the following C code snippet is not valid?
(A) char* p = “string1”; printf(“%c”, *++p);
(B) char q[] = “string1”; printf(“%c”, *++q);
(C) char* r = “string1”; printf(“%c”, r[1]);
(D) None of the above
Answer: B

What does the following fragment of C-program print?

```
char c[] = "GEEK2018";
char *p =c;
printf("%c,%c", *p,*(p+p[3]-p[1]));
```

(A) G, 1
(B) G, K
(C) GEEK2018
(D) None of the above
Hint: ASCII value
Answer: A

Q. What is the output?

```
char p[20];
char *s = "string";
int length = strlen(s);
int i;
for (i = 0; i < length; i++)
     p[i] = s[length — i];
printf("%s",p);
```

The output of the program is:
(A) gnirts
(B) gnirt
(C) string
(D) no output is printed
Answer: D

Q. What does the following fragment of C-program print?

```
char c[] = "GATE2011";
char *p =c;
printf("%s", p + p[3] - p[1]) ;
```

(A) GATE2011
(B) E2011
(C) 2011
(D) 011
Answer: C

What is the output of following code:

```
main()
{
char *p;
p="Hello";
printf("%c\n",*&*p);
}
```

A. Address Of p
B. Hello
C. H
D. None
Answer: C

```
int y[4] = {6, 7, 8, 9};
int *ptr = y + 2; printf("%d\n", ptr[ 1 ] );
```

What is printed when the sample code above is executed?

A. 6
B. 7
C. 8
D. 9
Answer: D

```
main()
{
    char *p;
    int *q;
    long *r;
    p=q=r=0;
    p++;
    q++;
    r++;
    printf("%p...%p...%p",p,q,r);
}
```

A. Compliation Error
B. 0001...0002...0004
C. 0000002…000004…0000003
D. Runtime Error
Answer: B

CHAPTER- 2: Linked List

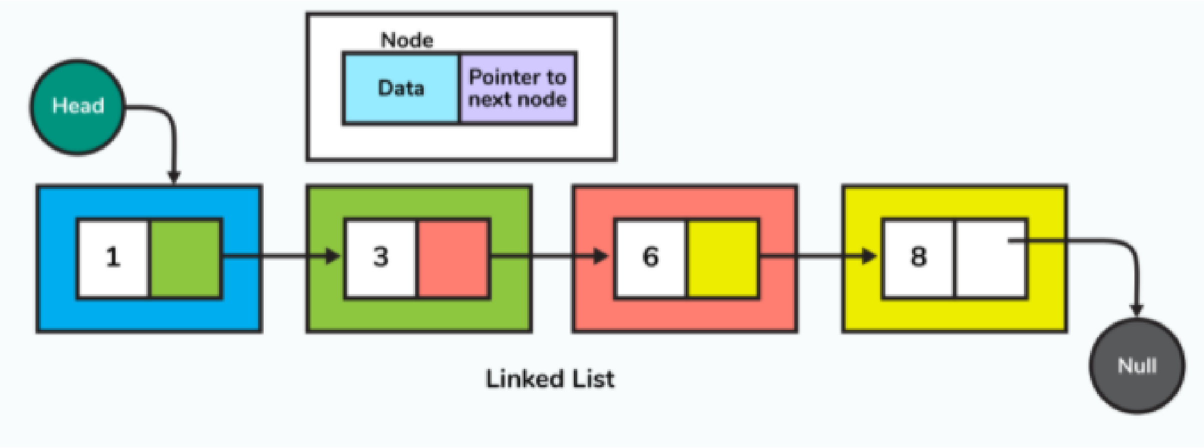
What is a linked list?

Linked List is the collection of randomly stored data objects called nodes. In Linked List, each node is linked to its adjacent node through a pointer. A node contains two fields, i.e. Data Field and Link Field. Each node element has two parts:

a data field

a reference (or pointer) to the next node.

The first node in a linked list is called the head and the last node in the list has the pointer to NULL. Null in the reference field indicates that the node is the last node. When the list is empty, the head is a null reference.



Interview Questions:

Are linked lists of linear or non-linear type?

Answer:

A linked list is considered both linear and non-linear data structure depending upon the situation.

- On the basis of data storage, it is considered as a non-linear data structure.
- On the basis of the access strategy, it is considered as a linear data-structure.

How are linked lists more efficient than arrays?

Asnwer:

- The size of a linked list can be incremented at runtime which is impossible in the case of the array.
- The List is not required to be contiguously present in the main memory, if the contiguous space is not available, the nodes can be stored anywhere in the memory connected through the links.
- The List is dynamically stored in the main memory and grows as per the program demand while the array is statically stored in the main memory, size of which must be declared at compile time.
- The number of elements in the linked list are limited to the available memory space while the number of elements in the array is limited to the size of an array.

Write the syntax in C to create a node in the singly linked list.

```
struct node
{
   int data;
   struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node));
```

What is a doubly-linked list (DLL)? What are its applications.

Answer:

The doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. In a doubly linked list, a node consists of three parts:

- node data
- pointer to the next node in sequence (next pointer)
- pointer to the previous node (previous pointer).

Applications of DLL are:

A music playlist with next song and previous song navigation options.

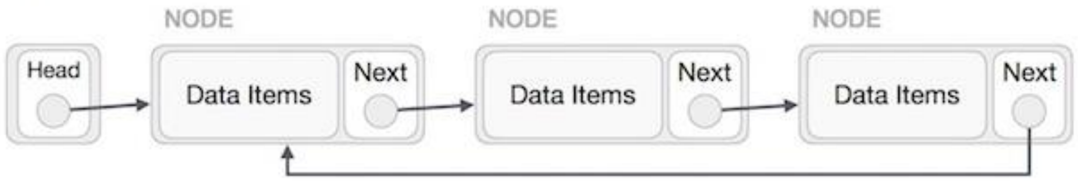
The browser cache with BACK-FORWARD visited pages

The undo and redo functionality on platforms such as word, paint etc, where you can reverse the node to get to the previous page.

Circular Linked List

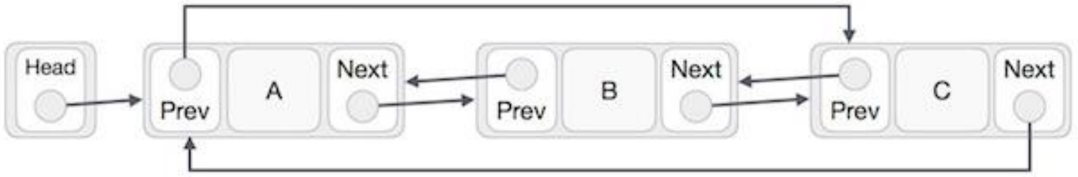
Singly Linked List as Circular

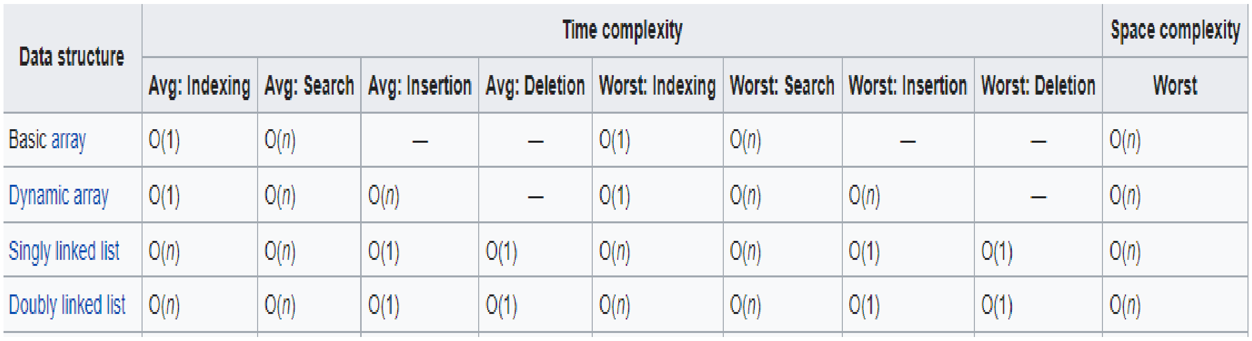
In singly linked list, the next pointer of the last node points to the first node.



Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.





Data structure	Time complexity								Space complexity
	Avg: Indexing	Avg: Search	Avg: Insertion	Avg: Deletion	Worst: Indexing	Worst: Search	Worst: Insertion	Worst: Deletion	Worst
Basic array	O(1)	O(*n*)	—	—	O(1)	O(*n*)	—	—	O(*n*)
Dynamic array	O(1)	O(*n*)	O(*n*)	—	O(1)	O(*n*)	O(*n*)	—	O(*n*)
Singly linked list	O(*n*)	O(*n*)	O(1)	O(1)	O(*n*)	O(*n*)	O(1)	O(1)	O(*n*)
Doubly linked list	O(*n*)	O(*n*)	O(1)	O(1)	O(*n*)	O(*n*)	O(1)	O(1)	O(*n*)

<u>How is an Array different from Linked List?</u>

- The size of the arrays is fixed, Linked Lists are Dynamic in size.
- Inserting and deleting a new element in an array of elements is expensive, Whereas both insertion and deletion can easily be done in Linked Lists.
- Random access is not allowed in Linked Listed.
- Extra memory space for a pointer is required with each element of the Linked list.
- Arrays have better cache locality that can make a pretty big difference in performance

Quiz Questions for Topic 2

Which of the following statement is true?
i) Using singly linked lists and circular list, it is not possible to traverse the list backwards.
ii) To find the predecessor, it is required to traverse the list from the first node in case of singly linked list.
A) i-only
B) ii-only
C) Both i and ii
D) None of both

Answer: A

The advantage of …………….. is that they solve the problem if sequential storage representation. But disadvantage in that is they are sequential lists.
A) Lists
B) Linked Lists
C) Trees
D) Queues
Answer: B

In a circular linked list

a) Components are all linked together in some sequential manner.
b) There is no beginning and no end.
c) Components are arranged hierarchically.
d) Forward and backward traversal within the list is permitted.

Answer: B

A linear collection of data elements where the linear node is given by means of pointer is called?

a) Linked list
b) Node list
c) Primitive list
d) None
Answer:A
Which of the following operations is performed more efficiently by doubly linked list than by singly linked list?
a) Deleting a node whose location in given
b) Searching of an unsorted list for a given item
c) Inverting a node after the node with given location
d) Traversing a list to process each node
Answer: A
Consider an implementation of unsorted singly linked list. Suppose it has its representation with a head and tail pointer. Given the representation, which of the following operation can be implemented in O(1) time?

i) Insertion at the front of the linked list
ii) Insertion at the end of the linked list
iii) Deletion of the front node of the linked list
iv) Deletion of the last node of the linked list
a) I and II
b) I and III
c) I,II and III
d) I,II and IV
Answer: C
Consider an implementation of unsorted singly linked list. Suppose it has its representation with a head pointer only. Given the representation, which of the following operation can be implemented in O(1) time?

i) Insertion at the front of the linked list
ii) Insertion at the end of the linked list
iii) Deletion of the front node of the linked list
iv) Deletion of the last node of the linked list

a) I and II
b) I and III
c) I,II and III
d) I,II and IV
Answer: B
In linked list each node contain minimum of two fields. One field is data field to store the data second field is?

a) Pointer to character
b) Pointer to integer
c) Pointer to node
d) Node
Answer:C
What would be the asymptotic time complexity to add a node at the end of singly linked list, if the pointer is initially pointing to the head of the list?

a) O(1)
b) O(n)
c) θ (n)
d) θ (1)
Answer:C
What would be the asymptotic time complexity to add an element in the linked list?

a) O(1)
b) O(n)
c) O(n^2)
d) None
Answer: B

What would be the asymptotic time complexity to find an element in the linked list?

a) O(1)
b) O(n)
c) O(n^2)
d) None
Answer: B
What would be the asymptotic time complexity to insert an element at the second position in the linked list?

a) O(1)
b) O(n)
c) O(n^2)
d) None
Answer: A
The concatenation of two list can performed in O(1) time. Which of the following variation of linked list can be used?

a) Singly linked list
b) Doubly linked list
c) Circular doubly linked list
d) Array implementation of list
Answer: C

Consider the following definition in c programming language

```
struct node
{
int data;
struct node * next;
}
typedef struct node NODE;
NODE *ptr;
```

Which of the following c code is used to create new node?

a) ptr=(NODE*)malloc(sizeof(NODE));
b) ptr=(NODE*)malloc(NODE);
c) ptr=(NODE*)malloc(sizeof(NODE*));
d) ptr=(NODE)malloc(sizeof(NODE));
Answer: A
A variant of linked list in which last node of the list points to the first node of the list is?
a) Singly linked list
b) Doubly linked list

c) Circular linked list
d) Multiply linked list
Answer: C
In doubly linked lists, traversal can be performed?

a) Only in forward direction
b) Only in reverse direction
c) In both directions
d) None
Answer: C
What kind of linked list is best to answer question like “What is the item at position n?”

a) Singly linked list
b) Doubly linked list
c) Circular linked list
d) Array implementation of linked list
Answer: D
20. A variation of linked list is circular linked list, in which the last node in the list points to first node of the list. One problem with this type of list is?

a) It waste memory space since the pointer head already points to the first node and thus the list node does not need to point to the first node.
b) It is not possible to add a node at the end of the list.
c) It is difficult to traverse the list as the pointer of the last node is now not NULL
d) All of above
Answer: C
In circular linked list, insertion of node requires modification of?

a) One pointer
b) Two pointer
c) Three pointer
d) None
Answer: B

In worst case, the number of comparison need to search a singly linked list of length n for a given element is

a) log n
b) n/2
c) $\log_2 n$-1
d) n
Answer: D

consider the function f defined here:

```
struct item
{
int data;
struct item * next;
};
int f (struct item *p)
{
return( (p==NULL) || ((p->next==NULL)||(p->data<=p->next->data) && (p->next)));
}
```

For a given linked list p, the function f returns 1 if and only if

a) the list is empty or has exactly one element
b) the element in the list are sorted in non-decreasing order of data value
c) the element in the list are sorted in non-increasing order of data value
d) not all element in the list have the same data value

Answer: B

The time required to delete a node x from a doubly linked list having n nodes is

a. O (n)
b. O (log n)
c. O (1)
d. O (n log n)

Answer:: C

Linked lists are best suited

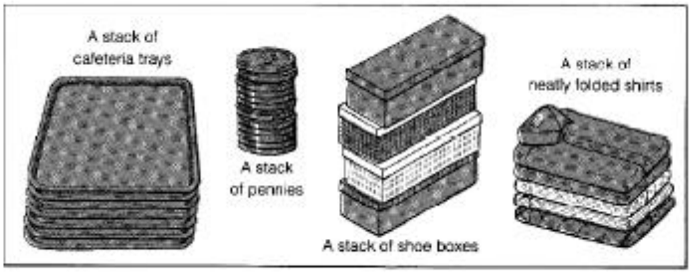
A. for relatively permanent collections of data
B. for the size of the structure and the data in the structure are constantly changing
C. for both of above situation
D. for none of above situation

Answer B

Chapter 3: STACK

STACK

It is an ordered group of homogeneous items of elements. Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack). The last element to be added is the first to be removed (LIFO: Last In, First Out).



one end, called ***TOP*** of the stack. The elements are removed in reverse order of that in which they were inserted into the stack.

2.2 STACK OPERATIONS

These are two basic operations associated with stack:

• Push() is the term used to insert/add an element into a stack.

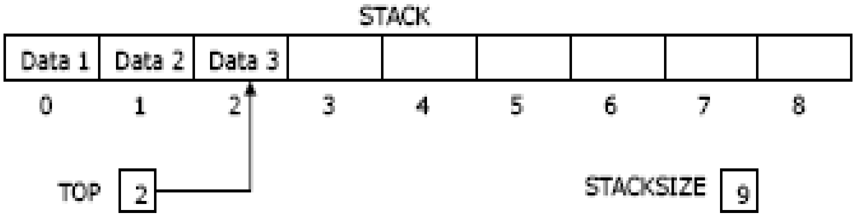
. Pop() is the term used to delete/remove an element from a stack.

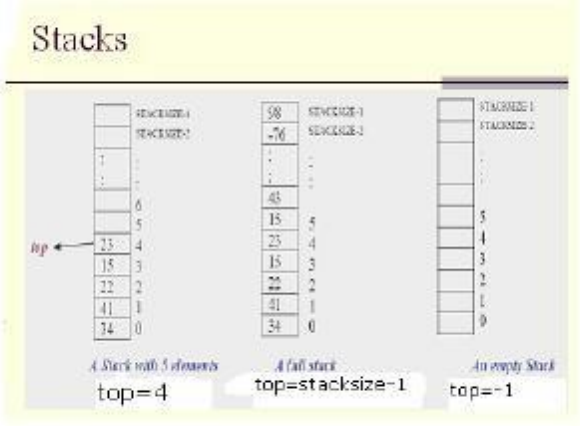
Other names for stacks are piles and push-down lists.

There are two ways to represent Stack in memory. One is using array and other is using linked list.

Array representation of stacks:

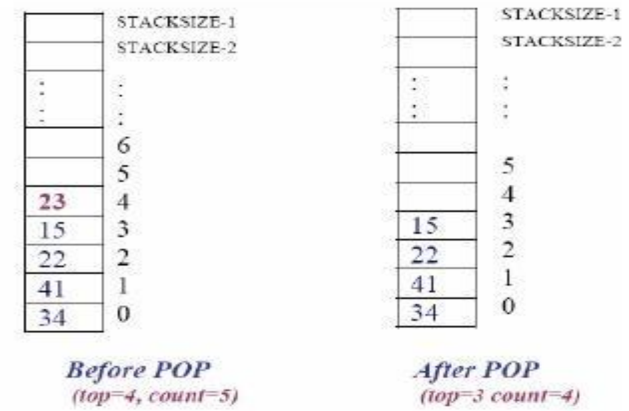
Usually, the stacks are represented in the computer by a linear array. In the following algorithms/procedures of pushing and popping an item from the stacks, we have considered, a linear array STACK, a variable TOP which contain the location of the top element of the stack; and a variable STACKSIZE which gives the maximum number of elements that can be hold by the stack



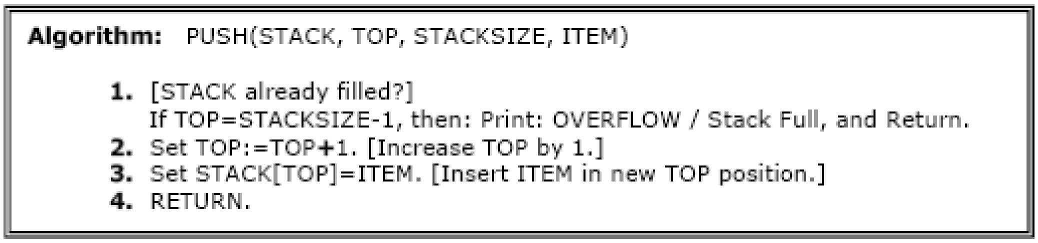


Push Operation

Push an item onto the top of the stack (***insert an item***)

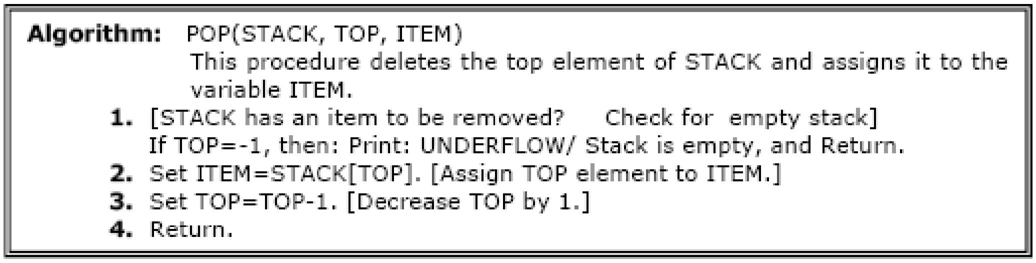


Algorithm for PUSH:



```
Algorithm:   PUSH(STACK, TOP, STACKSIZE, ITEM)

        1. [STACK already filled?]
           If TOP=STACKSIZE-1, then: Print: OVERFLOW / Stack Full, and Return.
        2. Set TOP:=TOP+1. [Increase TOP by 1.]
        3. Set STACK[TOP]=ITEM. [Insert ITEM in new TOP position.]
        4. RETURN.
```

Algorithm for POP:

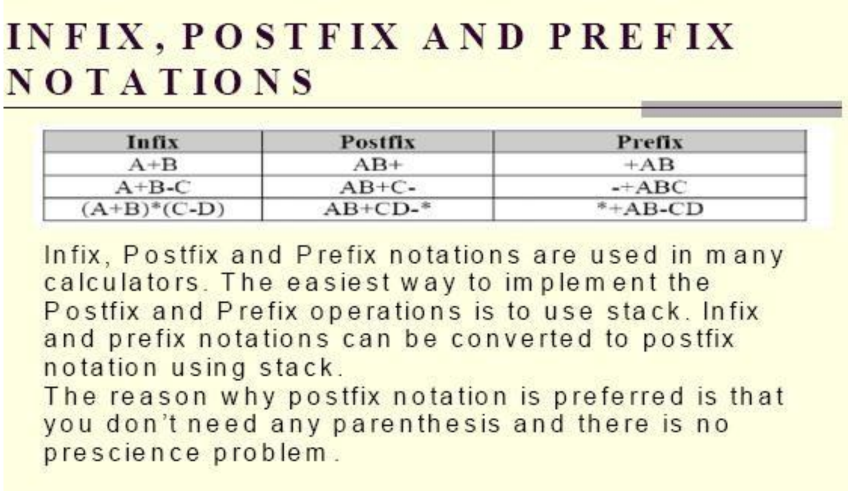


```
Algorithm:   POP(STACK, TOP, ITEM)
             This procedure deletes the top element of STACK and assigns it to the
             variable ITEM.
        1. [STACK has an item to be removed?    Check for  empty stack]
           If TOP=-1, then: Print: UNDERFLOW/ Stack is empty, and Return.
        2. Set ITEM=STACK[TOP]. [Assign TOP element to ITEM.]
        3. Set TOP=TOP-1. [Decrease TOP by 1.]
        4. Return.
```

Here are the minimal operations we'd need for an abstract stack (and their typical names):

o Push: Places an element/value on *top* of the stack.
o Pop: Removes value/element from *top* of the stack.
o IsEmpty: Reports whether the stack is Empty or not.
o IsFull: Reports whether the stack is Full or not.

APPLICATION OF THE STACK (ARITHMETIC EXPRESSIONS)

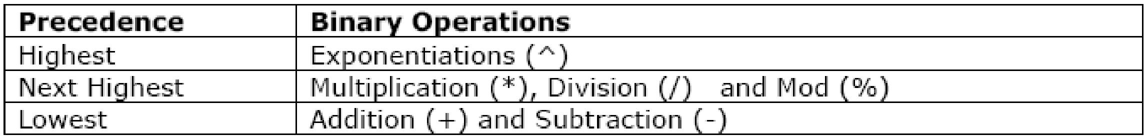


INFIX, POSTFIX AND PREFIX NOTATIONS

Infix	Postfix	Prefix
A+B	AB+	+AB
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD

Infix, Postfix and Prefix notations are used in many calculators. The easiest way to implement the Postfix and Prefix operations is to use stack. Infix and prefix notations can be converted to postfix notation using stack.
The reason why postfix notation is preferred is that you don't need any parenthesis and there is no prescience problem.

Stacks are used by compilers to help in the process of converting infix to postfix arithmetic expressions and also evaluating arithmetic expressions. Arithmetic expressions consisting variables, constants, arithmetic operators and parentheses. Humans generally write expressions in which the operator is written between the operands (**3 + 4**, for example). This is called infix notation. Computers "prefer" postfix notation in which the operator is written to the right of two operands. The preceding infix expression would appear in postfix notation as **3 4 +**. To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation, and then evaluate the postfix version of the expression. We use the following three levels of precedence for the five binary operations.



Precedence	Binary Operations
Highest	Exponentiations (^)
Next Highest	Multiplication (*), Division (/) and Mod (%)
Lowest	Addition (+) and Subtraction (-)

For example:
(66 + 2) * 5 – 567 / 42
to
postfix
66 22 + 5 * 567 42 / –

Transforming Infix Expression into Postfix Expression:

The following algorithm transforms the infix expression **Q** into its equivalent postfix expression

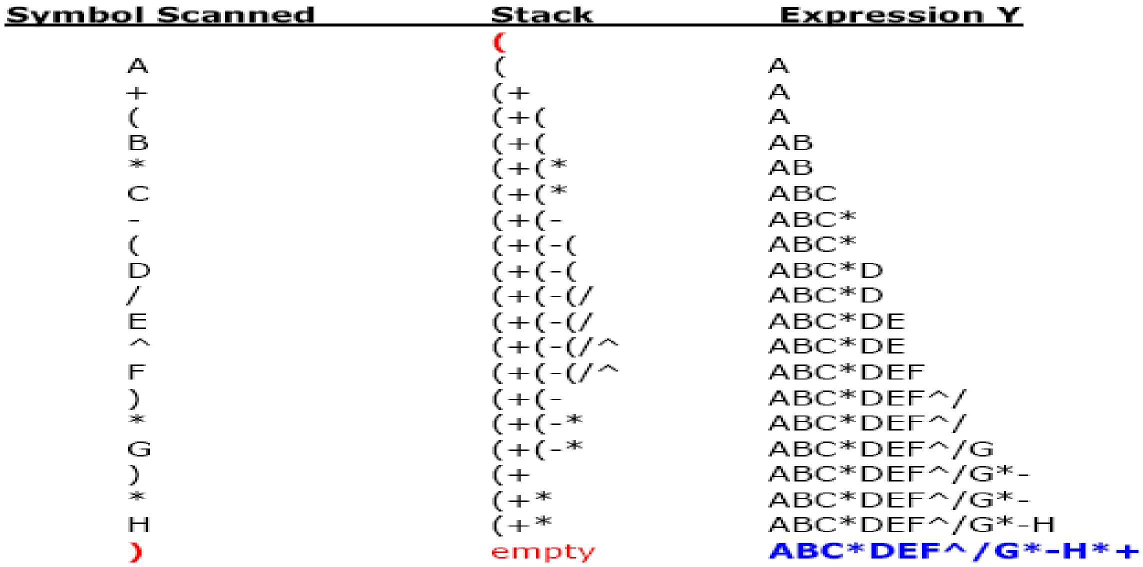
P. It uses a stack to temporary hold the operators and left parenthesis. The postfix expression will be constructed from left to right using operands from **Q** and operators popped from STACK.

Convert **Q**: **A+(B * C – (D / E ^ F) * G) * H** into postfix form showing stack status .

Now add **")"** at the end of expression

A+(B * C – (D / E ^ F) * G) * H)

and also Push a **"("** on Stack.



Symbol Scanned	Stack	Expression Y
	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
(	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(-*	ABC*DEF^/
G	(+(-*	ABC*DEF^/G
)	(+	ABC*DEF^/G*-
*	(+*	ABC*DEF^/G*-
H	(+*	ABC*DEF^/G*-H
)	empty	**ABC*DEF^/G*-H*+**

EVALUATION OF POSTFIX EXPRESSION

If **P** is an arithmetic expression written in postfix notation. This algorithm uses STACK to hold operands, and evaluate. **For example:**

Following is an infix arithmetic expression

(5 + 2) * 3 – 9 / 3

And its postfix is:

5 2 + 3 * 9 3 / –

Now add **"$"** at the end of expression as a sentinel.

Scanned Elements	Stack	Action to do
5	5	Pushed on stack
2	5, 2	Pushed on Stack
+	7	Remove the two top elements and calculate 5 + 2 and push the result on stack
3	7, 3	Pushed on Stack
*	21	Remove the two top elements and calculate 7 * 3 and push the result on stack
8	21, 8	Pushed on Stack
4	21, 8, 4	Pushed on Stack
/	21, 2	Remove the two top elements and calculate 8 / 2 and push the result on stack
-	19	Remove the two top elements and calculate 21 - 2 and push the result on stack
$	**19**	**Sentinel $ encouter ,** Result is on top of the STACK

2.6 RECURSION

Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". This makes it sound very similar to a loop because it repeats the same code, and in some ways, it *is* similar to looping. On the other hand, recursion makes it easier to express ideas in which the result of the recursive call is necessary to complete the task. Of course, it must be possible for the "process" to sometimes be completed without the recursive call. One simple example is the idea of building a wall that is ten feet high; if I want to build a ten-foot-high wall, and then I will first build a 9-foot-high wall, and then add an extra foot of bricks. Conceptually, this is like saying the "build wall" function takes a height and if that height is greater than one, first calls itself to build a lower wall, and then adds one a foot of bricks.

A simple example of recursion would be:

```
void recurse()
{
      recurse(); /* Function calls itself */
}

int main()
{
      recurse(); /* Sets off the recursion */
      return 0;
}
```

This program will not continue forever, however. The computer keeps function calls on a stack and once too many are called without ending, the program will crash. Why not write a program to see how many times the function is called before the program terminates?

A quick example:

```
void count_to_ten ( int count )
{
      /* we only keep counting if we have a value less than ten if ( count < 10 )
            {
                  count_to_ten( count + 1 );
            }
}
int main()
{
   count_to_ten ( 0 );
}
```

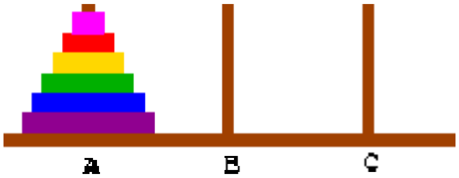
Simulation of Recursion

Tower of Hanoi Problem

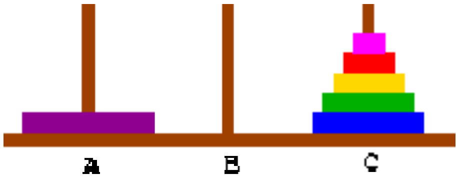
Using recursion often involves a key insight that makes everything simpler. Often the insight is determining what data exactly we are recursing on - we ask, what is the essential feature of the problem that should change as we call ourselves? In the case of isAJew, the feature is the person in question: At the top level, we are asking about a person; a level deeper, we ask about the person's mother; in the next level, the grandmother; and so on.

In our Towers of Hanoi solution, we recurse on the largest disk to be moved. That is, we will write a recursive function that takes as a parameter the disk that is the largest disk in the tower we want to move. Our function will also take three parameters indicating from which peg the tower should be moved (*source*), to which peg it should go (*dest*), and the other peg, which we can use temporarily to make this happen (*spare*).

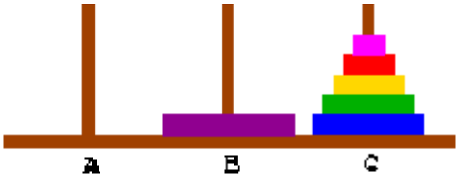
At the top level, we will want to move the entire tower, so we want to move disks 5 and smaller from peg A to peg B. We can break this into three basic steps.



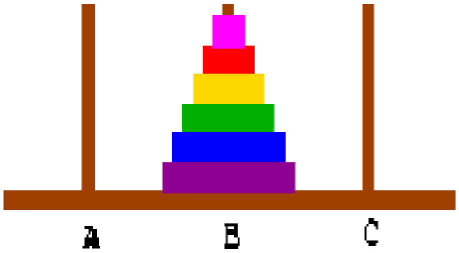
1. Move disks 4 and smaller from peg A (*source*) to peg C (*spare*), using peg B (*dest*) as a spare. How do we do this? By recursively using the same procedure. After finishing this, we'll have all the disks smaller than disk 4 on peg C. (Bear with me if this doesn't make sense for the moment - we'll do an example soon.)



2. Now, with all the smaller disks on the spare peg, we can move disk 5 from peg A (*source*) to peg B (*dest*).



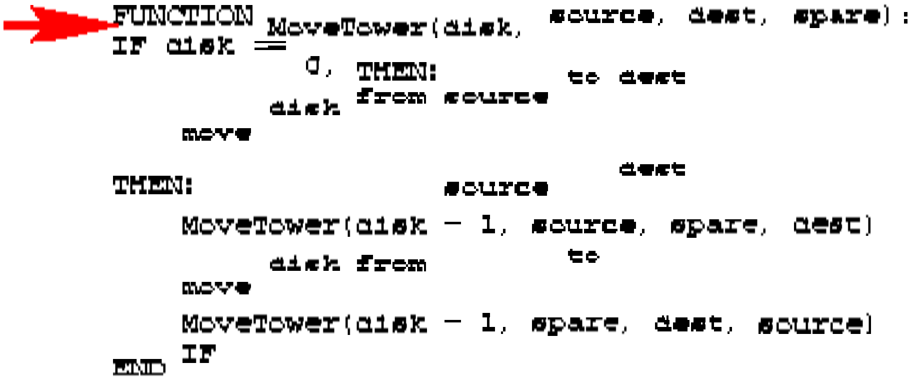
3. Finally, we want disks 4 and smaller moved from peg C (*spare*) to peg B (*dest*). We do this recursively using the same procedure again. After we finish, we'll have disks 5 and smaller all on *dest*.



In pseudocode, this looks like the following. At the top level, we'll call MoveTower with *disk*=5, *source*=A, *dest*=B, and *spare*=C.

```
FUNCTION MoveTower(disk, source, dest, spare): IF
disk == 0, THEN:
   move disk from source to dest
ELSE:
   MoveTower(disk - 1, source, spare, dest)   // Step 1 above
   move disk from source to dest              // Step 2 above
   MoveTower(disk - 1, spare, dest, source)   // Step 3 above
END IF
```

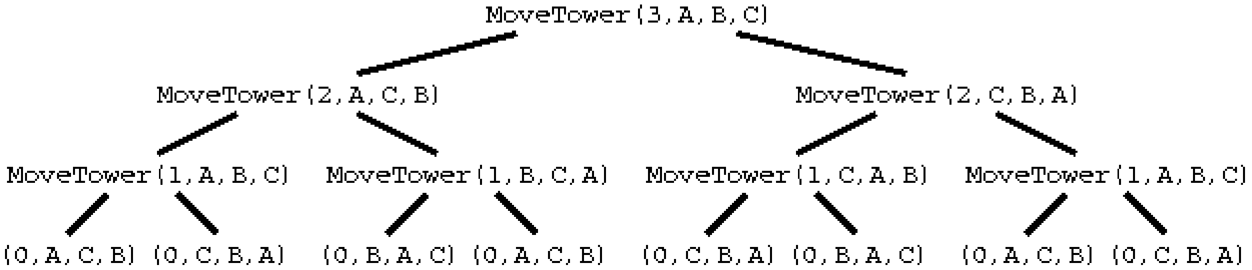
Note that the pseudocode adds a base case: When *disk* is 0, the smallest disk. In this case we don't need to worry about smaller disks, so we can just move the disk directly. In the other cases, we follow the three-step recursive procedure we already described for disk 5.



```
FUNCTION MoveTower(disk,  source,  dest,  spare):
IF disk ==
            0,  THEN:          to dest
          disk  from source
     move
                                 dest
THEN:               source
     MoveTower(disk - 1, source, spare, dest)
          disk from         to
     move
     MoveTower(disk - 1, spare, dest, source)
END  IF
```

The *call stack* in the display above represents where we are in the recursion. It keeps track of the different levels going on. The current level is at the bottom in the display. When we make a new recursive call, we add a new level to the call stack representing this recursive call. When we finish with the current level, we remove it from the call stack (this is called *popping the stack*) and continue with where we left off in the level that is now current.

Another way to visualize what happens when you run MoveTower is called a *call tree*. This is a graphic representation of all the calls. Here is a call tree for MoveTower(3,A,B,C).



We call each function call in the call tree a *node*. The nodes connected just below any node *n* represent the function calls made by the function call for *n*. Just below the top, for example, are MoveTower(2,A,C,B) and MoveTower(2,C,B,A), since these are the two function calls that MoveTower(3,A,B,C) makes. At the bottom are many nodes without any nodes connected below them - these represent base case

ESSENTIAL QUIZ 1

1) Stack is useful for implementing
(a) Radix sort
(b) Recursion
(c) Breadth first
(d) Depth first

2)The postfix equivalent of the prefix *+ab – cd is
(a) ab + cd- *
(b) abcd +- *
(c) ab + cd* -
(d) ab+- cd*

3) Process of inserting an element in stack is called ____________
a) Create
b) Push
c) Evaluation
d) Pop

4) Process of removing an element from stack is called __________
a) Create
b) Push
c) Evaluation
d) Pop
5) In a stack, if a user tries to remove an element from an empty stack it is called _________
a) Underflow
b) Empty collection
c) Overflow
d) Garbage Collection
6) Pushing an element into stack already having five elements and stack size of 5, then stack becomes ___________
a) Overflow
b) Crash
c) Underflow
d) User flow
7) Entries in a stack are "ordered". What is the meaning of this statement?
a) A collection of stacks is sortable
b) Stack entries may be compared with the '<' operation
c) The entries are stored in a linked list
d) There is a Sequential entry that is one by one
Explanation: In stack data structure, elements are added one by one using push operation. Stack follows LIFO Principle i.e. Last In First Out(LIFO).
8) . Which of the following is not the application of stack?
a) A parentheses balancing program

b) Tracking of local variables at run time

c) Compiler Syntax Analyzer

d) Data Transfer between two asynchronous process.

9) Consider the usual algorithm for determining whether a sequence of parentheses is balanced. The maximum number of parentheses that appear on the stack AT ANY ONE TIME when the algorithm analyzes: (()(())(()))?

a) 1

b) 2

c) 3

d) 4 or more

Explanation: In the entire parenthesis balancing method when the incoming token is a left parenthesis it is pushed into stack. A right parenthesis makes pop operation to delete the elements in stack till we get left parenthesis as top most element. 3 elements are there in stack before right parentheses comes. Therefore, maximum number of elements in stack at run time is 3.

10) What is the value of the postfix expression 6 3 2 4 + – *?

a) 1

b) 40

c) 74

d) -18

11) Here is an infix expression: 4 + 3*(6*3-12). Suppose that we are using the usual stack algorithm to convert the expression from infix to postfix notation. The maximum number of symbols that will appear on the stack AT ONE TIME during the conversion of this expression?

a) 1

b) 2

c) 3

d) 4

12) The postfix form of the expression (A+ B)*(C*D- E)*F / G is?

a) AB+ CD*E – FG /**

b) AB + CD* E – F **G /

c) AB + CD* E – *F *G /

d) AB + CDE * – * F *G /

13) The data structure required to check whether an expression contains a balanced parenthesis is?

a) Stack

b) Queue

c) Array

d) Tree

14) What data structure would you mostly likely see in non recursive implementation of a recursive algorithm?

a) Linked List

b) Stack

c) Queue

d) Tree

15) The process of accessing data stored in a serial access memory is similar to manipulating data on a ________

a) Heap

b) Binary Tree
c) Array
d) Stack
16) The postfix form of A*B+C/D is?
a) *AB/CD+
b) AB*CD/+
c) A*BC+/D
d) ABCD+/*
17) Which data structure is needed to convert infix notation to postfix notation?
a) Branch
b) Tree
c) Queue
d) Stack
18) The prefix form of A-B/ (C * D ^ E) is?
a) -/*^ACBDE
b) -ABCD*^DE
c) -A/B*C^DE
d) -A/BC*^DE
19) What is the result of the following operation?
Top (Push (S, X))
a) X
b) X+S
c) S
d) XS
20) The prefix form of an infix expression (p + q) – (r * t) is?
a) + pq – *rt
b) – +pqr * t
c) – +pq * rt
d) – + * pqrt

ESSENTIAL QUIZ 2

1) The result of evaluating the postfix expression 5, 4, 6, +, *, 4, 9, 3, /, +, * is?
a) 600
b) 350
c) 650
d) 588
2) Convert the following infix expressions into its equivalent postfix expressions.
(A + B ΛD)/(E – F)+G
a) (A B D Λ + E F – / G +)
b) (A B D +Λ E F – / G +)
c) (A B D Λ + E F/- G +)
d) (A B D E F + Λ / – G +)
3) Convert the following Infix expression to Postfix form using a stack.
x + y * z + (p * q + r) * s, Follow usual precedence rule and assume that the expression is legal.

a) xyz*+pq*r+s*+
b) xyz*+pq*r+s+*
c) xyz+*pq*r+s*+
d) xyzp+**qr+s*+

4) Which of the following statement(s) about stack data structure is/are NOT correct?
a) Linked List are used for implementing Stacks
b) Top of the Stack always contain the new node
c) Stack is the FIFO data structure
d) Null link is present in the last node at the bottom of the stack

5) Consider the following operation performed on a stack of size 5.
Push(1);
Pop();
Push(2);
Push(3);
Pop();
Push(4);
Pop();
Pop();
Push(5);

After the completion of all operation, the number of elements present in stack is?
a)1
b)2
c)3
d) 4

6) Which of the following is not an inherent application of stack?
a) Reversing a string
b) Evaluation of postfix expression
c) Implementation of recursion
d) Job scheduling

7) The type of expression in which operator succeeds its operands is?
a) Infix Expression
b) Prefix Expression
c) Postfix Expression
d) Both Prefix and Postfix Expressions

8) If the elements “A”, “B”, “C” and “D” are placed in a stack and are deleted one at a time, what is the order of removal?
a) ABCD
b) DCBA
c) DCAB
d) ABDC

9) A normal queue, if implemented using an array of size MAX_SIZE, gets full when?
a) Rear = MAX_SIZE – 1
b) Front = (rear + 1)mod MAX_SIZE
c) Front = rear + 1
d) Rear = front

10) Which of the following real-world scenarios would you associate with a stack data structure?
a) piling up of chairs one above the other

b) people standing in a line to be serviced at a counter

c) offer services based on the priority of the customer

d) tatkal Ticket Booking in IRCTC

11) What does ‘stack underflow’ refer to?

a) accessing item from an undefined stack

b) adding items to a full stack

c) removing items from an empty stack

d) index out of bounds exception

12) What is the time complexity of pop() operation when the stack is implemented using an array?

a) O(1)

b) O(n)

c) O(logn)

d) O(nlogn)

13) Which of the following array position will be occupied by a new element being pushed for a stack of size N elements(capacity of stack > N)?

a) S[N-1]

b) S[N]

c) S[1]

d) S[0]

14) What happens when you pop from an empty stack while implementing using the Stack ADT in Java?

a) Undefined error

b) Compiler displays a warning

c) EmptyStackException is thrown

d) NoStackException is thrown

15) . Array implementation of Stack is not dynamic, which of the following statements supports this argument?

a) space allocation for array is fixed and cannot be changed during run-time

b) user unable to give the input for stack operations

c) a runtime exception halts execution

d) improper program compilation

16) Which of the following array element will return the top-of-the-stack-element for a stack of size N elements(capacity of stack > N)?

a) S[N-1]

b) S[N]

c) S[N-2]

d) S[N+1]

17) What is the best case time complexity of deleting a node in a Singly Linked list?

a) O (n)

b) O (n^2)

c) O (nlogn)

d) O (1)

18) **W**hich of the following statements are not correct with respect to Singly Linked List(SLL) and Doubly Linked List(DLL)?

a) Complexity of Insertion and Deletion at known position is O(n) in SLL and O(1) in DLL

b) SLL uses lesser memory per node than DLL

c) DLL has more searching power than SLL

d) Number of node fields in SLL is more than DLL

19) Minimum number of queues to implement stack is ____________

a) 3

b) 4

c) 1
d) 2

20) Which one of the following is an application of Stack Data Structure?
a) Managing function Call
b) Stock Span Problem
c) Arithmetic expression evaluation
d) All of the above

ESSENTIAL QUIZ 3

1) Minimum number of queues to implement stack is ____________
a) 3
b) 4
c) 1
d) 2

2) What should be done when a left parenthesis '(' is encountered?
a) Ignored
b) Placed in the output
c) Placed onto operator stack
d) Nothing

3) When an operand is read, which of the following is done?
a) Ignored
b) Placed in the output
c) Placed onto operator stack
d) Nothing

4) Which of the following is an infix expression?
a) (a+b)*(c+d)
b) ab+c*
c) +ab*c
d) *ab

5) What is the time complexity of an infix to postfix conversion algorithm?
a) O(nlogn)
b) O(n)
c) O(logn)
d) O(nm)

6) Parentheses are simply ignored in the conversion of infix to postfix expression.
a) True
b) False
c) Depends on question
d) None of these

7) It is easier for a computer to process a postfix expression than an infix expression.
a) False
b) True

8) What is the postfix expression for the infix expression?
a)-ab-c
b) -a-bc

c) -abc-
d)-abc
9) What is the postfix expression for the following infix expression?

a/b^c-d

a) abc^/d-
b) abc/^d-
c) ab^c/d-
d) None of these

10) Which of the following statement is incorrect with respect to infix to postfix conversion algorithm?
a) operand is always placed in the output
b) operator is placed in the stack when the stack operator has lower precedence
c) parenthesis is included in the output
d) higher and equal priority operators follow the same condition

11) In infix to postfix conversion algorithm, the operators are associated from?
a) R to L
b) L to R
c) mid to L
d) mid to R

12) What is the corresponding postfix expression for the given infix expression?

a*(b+c)/d

a) ab*+cd/
b) ab+*cd/
c) abc*+/d
d) abc+*d/

13) What would be the Prefix notation and Postfix notation for the given equation?
a) ++ABC and AB+C+
b) AB+C+ and ++ABC
c) ABC++ and AB+C+
d) ABC+ and ABC+

14) What would be the Prefix notation for the given equation?
a|b&c
a) a|&bc
b) &|abc
c) |a&bc
d) ab&|c

15) Out of the following operators (|, *, +, &, $), the one having lowest priority is

____

a) +
b) $
c) |
d) &

16) Which data structure can be used to test a palindrome?
a) Array
b) Linked List
c) stack
d) heap

17) How many stacks are required for applying evaluation of infix expression algorithm?
a) 1
b) 2
c) 3
d) 4

18) How many passes does the evaluation of infix expression algorithm makes through the input?
a) 1
b) 2
c) 3
d) 4

19. Evaluate the following statement using infix evaluation algorithm and choose the correct answer. 1+2*3-2
a) 3
b) 6
c) 5
d) 4

20. Evaluate the following and choose the correct answer.
a/b+c*d where a=4, b=2, c=2, d=1.
a) 1
b) 4
c) 5
d) 2

ADVANCED QUIZ 1

1) **Is it possible to implement 2 stack in an array?**
Condition: None of the stack should indicate an overflow until every slot of an array is used.
a. Only 1 stack can be implemented for the given condition
b. Stacks can not be implemented in array

c. 2 stacks can be implemented for the given condition.
d. 2 stacks can be implemented if the given condition is applied only for 1 stack.
Explanation: Yes, 2 stacks can be implemented for the given condition
Start 1st stack from left (1st position of an array) and 2nd from right (last position say n). Move 1st stack towards right(i.e 1,2,3 ...n) and 2nd towards left (i.e n,n-1,n-2...1).

2. Following is C like pseudo code of a function that takes a Queue as an argument, and uses a stack S to do processing.

```
void fun(Queue *Q)
{
    Stack S;  // Say it creates an empty stack S

    // Run while Q is not empty
    while (!isEmpty(Q))
    {
        // deQueue an item from Q and push the dequeued item to S
        push(&S, deQueue(Q));
    }

    // Run while Stack S is not empty
    while (!isEmpty(&S))
    {
      // Pop an item from S and enqueue the poppped item to Q
      enQueue(Q, pop(&S));
    }
}
```

What does the above function do in general?
a. Not possible to implement.
b Removes the last element fron Q
c. Make the Q empty
d. Reverses the Q
3. An implementation of a queue Q, using two stacks S1 and S2, is given below:

```
void insert(Q, x) {
   push (S1, x);
}

void delete(Q){
   if(stack-empty(S2)) then
      if(stack-empty(S1)) then {
          print("Q is empty");
          return;
      }
      else while (!(stack-empty(S1))){
          x=pop(S1);
          push(S2,x);
      }
   x=pop(S2);
}
```

Let n insert and m (<=n) delete operations be performed in an arbitrary order on an empty queue Let x and y be the number of push and pop operations performed respectively in the process. Which one of the following is true for all m and n?

a) n+m <= x < 2n and 2m <= y <= n+m
b) n+m <= x < 2n and 2m<= y <= 2n
c) 2m <= x < 2n and 2m <= y <= n+m
d) 2m <= x <2n and 2m <= y <= 2n

4. Suppose implementation supports an instruction REVERSE, which reverses the order of elements on the stack, in addition to the PUSH and POP instructions. Which one of the following statements is TRUE with respect to this modified stack?

a) A queue cannot be implemented using this stack
b) A queue can be implemented where ENQUEUE takes a single instruction and DEQUEUE takes a sequence of two instructions.
c) A queue can be implemented where ENQUEUE takes a sequence of three instructions and DEQUEUE takes a single instruction.
d) queue can be implemented where both ENQUEUE and DEQUEUE take a single instruction each.

5. Following is C like pseudo code of a function that takes a number as an argument, and uses a stack S to do processing.

```
void fun(int n)
{
    Stack S;  // Say it creates an empty stack S
    while (n > 0)
    {
      // This line pushes the value of n%2 to stack S
      push(&S, n%2);
      n = n/2;
    }
    // Run while Stack S is not empty
    while (!isEmpty(&S))
      printf("%d ", pop(&S)); // pop an element from S and print it
}
```

a) Prints binary representation of n in reverse order
b) Prints binary representation of n
c) Prints the value of Logn
d) Prints the value of Logn in reverse order

7. Consider the following pseudocode that uses a stack

```
declare a stack of characters
while ( there are more characters in the word to read )
{
   read a character
   push the character on the stack
}
while ( the stack is not empty )
{
   pop a character off the stack
   write the character to the screen
}
```

What is output for input "geeksquiz"?

a) geeksquizgeeksquiz

b) geeksquiz

c) ziuqskeeg

d) none of these

8. Following is an incorrect pseudocode for the algorithm which is supposed to determine whether a sequence of parentheses is balanced:

```
declare a character stack
while ( more input is available)
{
  read a character
  if ( the character is a '(' )
    push it on the stack
  else if ( the character is a ')' and the stack is not empty )
    pop a character off the stack
  else
    print "unbalanced" and exit
 }
 print "balanced"
```

Which of these unbalanced sequences does the above code think is balanced?

a) ((())

b) ((())))

c) ())(()

d) (()())

9. The following postfix expression with single digit operands is evaluated using a stack:

8 2 3 ^ / 2 3 * + 5 1 * -

Note that ^ is the exponentiation operator. The top two elements of the stack after the first * is evaluated are:

a) 6,1

b) 5,7

c) 3,2

d) 7,8

10. Let S be a stack of size n >= 1. Starting with the empty stack, suppose we push the first n natural numbers in sequence, and then perform n pop operations. Assume that Push and Pop operation take X seconds each, and Y seconds elapse between the end of one such stack operation and the start of the next operation. For m >= 1, define the stack-life of m as the time elapsed from the end of Push(m) to the start of the pop operation that removes m from S. The average stack-life of an element of this stack is

a) n(X+Y)

b) 3Y+2X

c) n(X+Y)-X

d) X+2Y

11. A single array A[1..MAXSIZE] is used to implement two stacks. The two stacks grow from opposite ends of the array. Variables top1 and top2 (topl< top 2) point to the location of the

topmost element in each of the stacks. If the space is to be used efficiently, the condition for “stack full” is

a) (top1 = MAXSIZE/2) and (top2 = MAXSIZE/2+1)

b) top1 + top2 = MAXSIZE

c) (top1= MAXSIZE/2) or (top2 = MAXSIZE)

d) top1= top2 -1

12. Assume that the operators +, -, × are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, x , +, -. The postfix expression corresponding to the infix expression a + b × c - d ^ e ^ f is

a) abc × + def ^ ^

b) abc × + de ^ f ^

c) ab + c × d - e ^ f ^

d) - + a × bc ^ ^ def

13. To evaluate an expression without any embedded function calls:

a) One Stack is enough

b) Two stacks required

c) As many stacks as the height of the expression tree are needed

d) A Turing machine is needed in the general case

14. The result evaluating the postfix expression 10 5 + 60 6 / * 8 – is

a) 284

b) 213

c) 142

d) 153

15. A function f defined on stacks of integers satisfies the following properties. f(∅) = 0 and f (push (S, i)) = max (f(S), 0) + i for all stacks S and integers i.
If a stack S contains the integers 2, -3, 2, -1, 2 in order from bottom to top, what is f(S)?

a) 6
b) 4
c) 3
d) 2

16. Consider the following C program:

```
  #include
        #define EOF -1
        void push (int); /* push the argument on the stack */
        int pop  (void); /* pop the top of the stack */
        void flagError ();
        int main ()
       {        int c, m, n, r;
                while ((c = getchar ()) != EOF)
               { if  (isdigit (c) )
                        push (c);
                else if ((c == '+') || (c == '*'))
                {       m = pop ();
                        n = pop ();
                        r = (c == '+') ? n + m : n*m;
                        push (r);
                 }
                 else if (c != ' ')
                        flagError ();
            }
            printf("% c", pop ());
}
```

What is the output of the program for the following input ? 5 2 * 3 3 2 + * +

a) 15
b) 25
c) 30
d) 150

17. Suppose a stack is to be implemented with a linked list instead of an array. What would be the effect on the time complexity of the push and pop operations of the stack implemented using linked list (Assuming stack is implemented efficiently)?
a) O(1) for insertion and O(n) for deletion
b) O(n) for insertion and O(n) for deletion
c) O(n) for insertion and O(1) for deletion
d) O(1) for insertion and O(1) for deletion

18. Consider n elements that are equally distributed in k stacks. In each stack, elements of it are arranged in ascending order (min is at the top in each of the stack and then increasing downwards). Given a queue of size n in which we have to put all n elements in increasing order. What will be the time complexity of the best-known algorithm?
a) O(nlogk)
b) O(klogn)
c) O(nk)
d) O(k^2)

19. A priority queue Q is used to implement a stack S that stores characters. PUSH(C) is implemented as INSERT(Q, C, K) where K is an appropriate integer key chosen by the implementation. POP is implemented as DELETEMIN(Q). For a sequence of operations, the keys chosen are in:
a) Non-increasing order
b) Non-decreasing order
c) strictly increasing order
d)strictly decreasing order

20. Consider the following statement:

i. First-in-first out types of computations are efficiently supported by STACKS.
ii. Implementing LISTS on linked lists is more efficient than implementing LISTS on an array for almost all the basic LIST operations.
iii. Implementing QUEUES on a circular array is more efficient than implementing QUEUES on a linear array with two indices.
iv. Last-in-first-out type of computations are efficiently supported by QUEUES.

Which of the following is correct?

a) (ii) and (iii) are true
b) (i) only
c) (i) and (iv) are true
d) (ii) and (iv) are true

ADVANCED QUIZ 2

1) Which of the following permutation can be obtained in the same order using a stack assuming that input is the sequence 5, 6, 7, 8, 9 in that order?

a) 7,8,9,5,6
b) 5,9,6,7,8
c) 7,8,9,6,5
d) none of these

2) The seven elements A, B, C, D, E, F and G are pushed onto a stack in reverse order, i.e., starting from G. The stack is popped five times and each element is inserted into a queue.Two elements are deleted from the queue and pushed back onto the stack. Now, one element is popped from the stack. The popped item is _________.

a) A
b) B
c) F
d) G

3. If the sequence of operations - push (1), push (2), pop, push (1), push (2), pop, pop, pop, push (2), pop are performed on a stack, the sequence of popped out values

a) 2,2,1,1,2
b) 2,2,1,2,2
c)2,1,1,2,2
d) 2,1,2,2,2

4. The five items: A, B, C, D, and E are pushed in a stack, one after other starting from A. The stack is popped four items and each element is inserted in a queue. The two elements are deleted from the queue and pushed back on the stack. Now one item is popped from the stack. The popped item is

a) A
b) B
c) C
d) D

5. Consider the following operations performed on a stack of size 5 : Push (a); Pop() ; Push(b); Push(c); Pop(); Push(d); Pop();Pop(); Push (e) Which of the following statements is correct?

a) Underflow occurs
b) Overflow occurs
c) Stack operation are performed smoothly
d) None of these

6. Which of the following is not an inherent application of stack?
a) Implementation of recursion
b) Job Scheduling
c) Paranthesis balance Check
d) Reverse of a string

7. Stack A has the entries a, b, c (with a on top). Stack B is empty. An entry popped out of stack A can be printed immediately or pushed to stack B. An entry popped out of the stack B can be only be printed. In this arrangement, which of the following permutations of a, b, c are not possible?
a) b c a
b) b a c
c) c a b
d) a b c

8. Convert the following infix expression into its equivalent post fix expression (A + B^ D) / (E – F) + G
a) ABD^ + EF – / G+
b) ABD + ^EF – / G+
c) ABD + ^EF / – G+
d) None of these

9. Consider the following sequence of operations on an empty stack.
Push(54);push(52);pop();push(55);push(62);s=pop();
Consider the following sequence of operations on an empty queue.
enqueue(21);enqueue(24);dequeue();enqueue(28);enqueue(32);q=dequeue();
The value of s+q is ____________.

a) **86**
b) **68**
c) **24**
d) **94**

10. An implementation of a queue Q, using two stacks S1 and S2, is given below:

```
void insert(Q, x) {
  push (S1, x);
}

void delete(Q){
  if(stack-empty(S2)) then
    if(stack-empty(S1)) then {
       print(“Q is empty”);
       return;
    }
    else while (!(stack-empty(S1))){
       x=pop(S1);
       push(S2,x);
    }
  x=pop(S2);
}
```

Let n insert and m (<=n) delete operations be performed in an arbitrary order on an empty queue Let x and y be the number of push and pop operations performed respectively in the process. Which one of the following is true for all m and n?

e) n+m <= x < 2n and 2m <= y <= n+m
f) n+m <= x < 2n and 2m<= y <= 2n
g) 2m <= x < 2n and 2m <= y <= n+m
h) 2m <= x <2n and 2m <= y <= 2n

11. The best data structure to check whether an arithmetic expression has balanced parentheses is a

a) Stack
b) Queue
c) Array
d) Heap

12. What is the corresponding postfix expression for the given infix expression?

a+(b*c(d/e^f)*g)*h)

a) ab*cdef/^*g-h+
b) abc*def^/g*-h*+
c) abcd*^ed/g*-h*+
d) abc*de^fg/*-*h+

13. What is the correct postfix expression for the following expression?

a+b*(c^d-e)^(f+g*h)-i

a) abcd^e-fgh*+^*+i-
b) abcde^-fgh*+^*+i-
c) ab+cd^e-fgh*+^*i-
d) None of these

14. Find the Prefix for following Postfix expression: ABC/-AK/L-*

a) *-A/BC-/AKL
b) -A/*BC-/AKL
c) *--/A/BCAKL
d) none of these

15. What would be the Prefix notation for the given equation?

(a+(b/c)*(d^e)-f)

a) -+a*/^bcdef
b) -+a*/bc^def
c) -+a*b/c^def
d) -a+*/bc^def

16. Select the appropriate code for the recursive Tower of Hanoi problem.(n is the number of disks)

```
a) public void solve(int n, String start, String auxiliary, String end)
{
    if (n == 1)
    {
      System.out.println(start + " -> " + end);
    }
    else
    {
      solve(n - 1, start, end, auxiliary);
      System.out.println(start + " -> " + end);
      solve(n - 1, auxiliary, start, end);
    }}
b) public void solve(int n, String start, String auxiliary, String end)
{
    if (n == 1)
    {
      System.out.println(start + " -> " + end);
    }
    else
    {
      solve(n - 1, auxiliary, start, end);
      System.out.println(start + " -> " + end);
    }
}
c) public void solve(int n, String start, String auxiliary, String end)
{
    if (n == 1)
    {
      System.out.println(start + " -> " + end);
    }
    else
    {
      System.out.println(start + " -> " + end);
          solve(n - 1, auxiliary, start, end);
    }
}
d) public void solve(int n, String start, String auxiliary, String end)
{
    if (n == 1)
    {
      System.out.println(start + " -> " + end);
    }
    else
    {
      solve(n - 1, start, end, auxiliary);
      System.out.println(start + " -> " + end);
    }
}
```

17) Select the appropriate code which reverses a word.

```
a) public String reverse(String input)
{
        for (int i = 0; i < input.length(); i++)
        {
       stk.push(input.charAt(i));
```

```
    }
        String rev = "";
        while (!stk.isEmpty())
        {
      rev = rev + stk.peek();
    }
        return rev;
}
b) public String reverse(String input)
{
        for (int i = 0; i < input.length(); i++)
        {
      stk.push(input.charAt(i));
    }
        String rev = "";
        while (!stk.isEmpty())
        {
      rev = rev + stk.pop();
    }
        return rev;
}
c) public String reverse(String input)
{
        for (int i = 0; i < input.length(); i++)
        {
      stk.push(input.charAt(i));
    }
        String rev = "";
        while (!stk.isEmpty())
        {
      rev = rev + stk.pop();
    }
}
d) public String reverse(String input)
{
        for (int i = 0; i < input.length(); i++)
        {
      stk.push(input.charAt(i));
    }
        String rev = "";
        while (!stk.isEmpty())
        {
      rev = rev + stk.pop();
          stk.pop();
    }
        return rev;
}
```

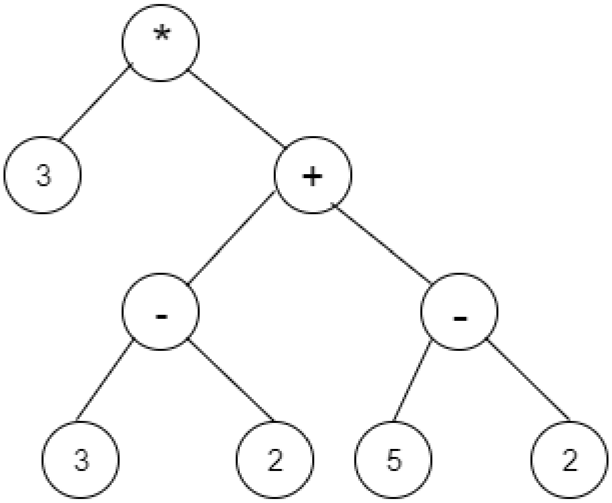
18) Which of the following statement is incorrect with respect to evaluation of infix expression algorithm?

a) Operand is pushed on to the stack

b) If the precedence of operator is higher, pop two operands and evaluate

c) If the precedence of operator is lower, pop two operands and evaluate
d) The result is pushed on to the operand stack

19. From the given expression tree, identify the infix expression, evaluate it and choose the correct result.



a) 5
b) 10
c) 12
d) 16

20. Find the output of the following prefix expression.

***+2-2 1/-4 2+-5 3 1**

a) 2
b) 12
c) 10
d) 4

QUESTION ANSWERS STACK

1. What is a Stack ?

A stack is a non-primitive linear data structure and is an ordered collection of homogeneous data elements.The other name of stack is Last-in -First-out list.

One of the most useful concepts and frequently used data structure of variable size for problem solving is the stack.

2. Write down the algorithm for solving Towers of Hanoi problem?

Push parameters and return address on stack.
If the stopping value has been reached then pop the stack to return to previous level else move all except the final disc from starting to intermediate needle.
Move final discs from start to destination needle.
Move remaining discs from intermediate to destination needle.

Return to previous level by popping stack.

3. What are the two operations of Stack?

PUSH

POP

4. What are the postfix and prefix forms of the expression?

A+B*(C-D)/(P-R)

Postfix form: ABCD-*PR-/+

Prefix form: +A/*B-CD-PR

5. Explain the usage of stack in recursive algorithm implementation?

In recursive algorithms, stack data structures are used to store the return address when a recursive call is encountered and also to store the values of all the parameters essential to the current state of the procedure.

6. What are applications of stack?

- Conversion of expression
- Evaluation of expression
- Parentheses matching
- Recursion

7. Define recursion?

It is a technique and it can be defined as any function that calls itself is called recursion. There are some applications which are suitable for only recursion such as, tower of Hanoi, binary tree traversals etc, can be implementing very easily and efficiently.

8. Which data structures are used for BFS and DFS of a graph?

- Queue is used for BFS
- Stack is used for DFS. DFS can also be implemented using recursion (Note that recursion also uses function call stack).

9. What is the difference between an Array and Stack?

Stack Data Structure:

- Size of the stack keeps on changing as we insert and delete the element
- Stack can store elements of different data type

Array Data Structure:

- Size of the array is fixed at the time of declaration itself
- Array stores elements of similar data type.

10. Which data structure is ideal to perform recursion operation and why?

Stack is the most ideal for recursion operation. This is mainly because of its LIFO (**Last In First Out**) property, it remembers the elements & their positions, so it exactly knows which one to return when a function is called.

11. What are some of the most important applications of a Stack?

Some of the important applications are given below. Check them out to know the detailed code & explanation.

- Balanced parenthesis checker
- Redundant braces
- Infix to postfix using a stack
- Infix to prefix using a stack

12. Convert the below given expression to its equivalent Prefix And Postfix notations. ((A + B) * C - (D - E) ^ (F + G))

Prefix Notation: ^ * +ABC DE + FG Postfix Notation: AB + C * DE FG + ^

13. How many stacks are required to implement a Queue.

2 stacks S1 and S2 ARE REQUIRED.
For Enqueue, perform push on S1.
For Dequeue, if S2 is empty pop all the elements from S1 and push it to S2. The last element you popped from S1 is an element to be dequeued. If S2 is not empty, then pop the top element in it.

14. Is it possible to implement 2 stacks in an array?

Condition: None of the stack should indicate an overflow until every slot of an array is used.
Yes, 2 stacks can be implemented for the given condition
Start 1st stack from left (1st position of an array) and 2nd from right (last position say n). Move 1st stack towards right(i.e 1,2,3 ...n) and 2nd towards left (i.e n,n-1,n-2...1).

15. Explain why Stack is a recursive data structure

A **stack** is a **recursive** data structure, so it's:

- a stack is either empty or
- it consists of a top and the rest which is a stack by itself;

16. Why and when should I use Stack or Queue data structures instead of Arrays/Lists?

Because they help manage your data in more a *particular* way than arrays and lists. It means that when you're debugging a problem, you won't have to wonder if someone randomly inserted an element into the middle of your list, messing up some invariants.
Arrays and lists are random access. They are very flexible and also easily *corruptible*. If you want to manage your data as FIFO or LIFO it's best to use those, already implemented, collections.
More practically you should:

- Use a queue when you want to get things out in the order that you put them in (FIFO)
- Use a stack when you want to get things out in the reverse order than you put them in (LIFO)
- Use a list when you want to get anything out, regardless of when you put them in (and when you don't want them to automatically be removed).

17. How to implement Linked List Using Stack?

You can simulate a linked list by using two stacks. One stack is the "list," and the other is used for temporary storage.

- To **add** an item at the head, simply push the item onto the stack.
- To **remove** from the head, pop from the stack.
- To **insert** into the middle somewhere, pop items from the "list" stack and push them onto the temporary stack until you get to your insertion point. Push the new item onto the "list" stack, then pop from the temporary stack and push back onto the "list" stack. Deletion of an arbitrary node is similar.

18. Explain the Concept of a Stack. How can you Differentiate it from a Queue?

A stack is a type of linear structure which is used to access components that support the *LIFO (Last In First Out)* method. Push and Pop are key stack functions. Unlike a queue, the arrays and linked lists are used to enforce a stack.
The element most recently added is removed first in a stack. However, in the event of a queue, the element least recently added is removed first.

19. Explain the Steps Involved in the Insertion and Deletion of an Element in the Stack.
Algorithms of Stack operations :

Algorithms of *PUSH* operations-

```
Step 1: Start
Step 2: Checks if the stack is full if(top==(SIZE-1))
Step 3: If the stack is full, Give a message and exit printf("\nStack Overflow");
Step 4: If the stack is not full, increment top to point next empty space.
top=top+1;
Step 5: Add data element to the stack location, where top is pointing.
printf("\nEnter the item to be pushed in stack:"); stack[top]=item;
Step 6: Stop
```

Algorithms of *POP* operations :

```
Step 1: Start
Step 2: Checks if the stack is empty if(top==-1)
Step 3: If the stack is empty, Give a message and exit printf("\nStack Underflow");
Step 4: If the stack is not empty, Print the element at which top is pointing.
printf("\nPopped element is : %d\n",stack[top]);
Step 5: Decrement top to point previous location .
top=top-1;
Step 6: Stop
```

20. Write the stack overflow condition.

Overflow occurs when **top = Maxsize -1**

21. What is a postfix expression?

An expression in which operators follow the operands is known as postfix expression. The main benefit of this form is that there is no need to group sub-expressions in parentheses or to consider operator precedence.

The expression "a + b" will be represented as "ab+" in postfix notation.

22. What are the drawbacks of array implementation of Stack?

- **Memory Wastage:** The space of the array, which is used to store stack elements can be wasted because it is fixed prior to its usage.
- **Array Size:** There might be situations in which, we may need to extend the stack to insert more elements if we use an array to implement stack, It will almost be impossible to extend the array size, therefore deciding the correct array size is always a problem in array implementation of stack.

23. WAP to reverse a queue using a stack.

```
/* Function to check if the queue is empty */
int empty()
{
if(st.front == -1)
return 1;
else
return 0;
}

/* Function to insert elements in a queue */
void enqueue(int num)
{
if(st.front == -1)
st.front++;
st.rear++;
st.s[st.rear] = num;
```

```
}

/* Function to delete elements from the queue */
int dequeue()
{
int x;
x = st.s[st.front];
if(st.front==st.rear)
st.front=st.rear=-1;
else
st.front++;
return x;
}

/* Function to display queue elements */
void display()
{
int i;
if(empty())
printf("\nEMPTY QUEUE\n");
else
{
printf("\nQUEUE ELEMENTS : ");
for(i = st.front ; i <= st.rear ; i++)
printf("%d ",st.s[i]);
}
printf("\n");
}

/* Function to reverse a queue using a stack */
void reverse_queue_using_stack()
{
while(!(st.front == st.rear))
{stack1.push(dequeue());}
stack1.push(dequeue());
printf("\nREVERSED QUEUE : ");
while(!stack1.empty())
{
printf("%d ",stack1.top());   // Print the top element of the stack
stack1.pop();
}
printf("\n");
exit(0);
}

/* Main function */
int main()
{
int num,choice;
st.front = st.rear = -1;
printf("\nREVERSING A QUEUE USING STACKS\n");
printf("\n1.ENQUEUE\n2.DEQUEUE\n3.DISPLAY\n4.REVERSE\n5.EXIT\n");
while(1)
{
printf("\nEnter the choice : ");
scanf("%d",&choice);
switch (choice)
{
case 1:
if(full())
{
printf("\nQUEUE IS FULL\n");
}
else
{
printf("\nEnter data : ");
scanf("%d",&num);
enqueue(num);
}
break;
case 2:
if (empty())
{
printf("\nEMPTY QUEUE\n");
}
else
printf("\nDequeued Element : %d",dequeue());
break;
case 3:
display();
break;
case 4: reverse_queue_using_stack();
break;
default: exit(0);
}}
return 0;
}
```

24. Solve the arithmetic expression 4 + 3*(6*3-12).

When we perform the conversion from infix to postfix expression +, *, (, * symbols are placed inside the stack. A maximum of 4 symbols are identified during the entire conversion. So, answer is 4.

25. Find the postfix form of the expression (A+ B)*(C*D- E)*F / G is?

(((A+ B)*(C*D- E)*F) / G) is converted to postfix expression as
=> (AB+(*(C*D- E)*F)/ G)
=> (AB+CD*E-*F) / G
=> (AB+CD*E-*F * G/).
Thus, Postfix expression is AB+CD*E-*F*G/

CHAPTER 4: QUEUE

QUEUE

1. It is an ordered collection of items from which items may be deleted from one end (called front end) and inserted from other end (called rear end). First item in is first item out and therefore, it is also known as FIFO structure.
2. **The ADT.**

```
ADT Queue is :
    objects    :  a finite ordered list of zero or more elements.
    functions :
    for all queue є Queue;  item є element;  maxQSize є positive integer
    Queue CreateQ(maxQSize)                  ::= create an empty queue  whose
                                                  maximum size is maxQSize

     Boolean IsFull(stack, maxQSize)         ::= if(number of elements in queue
                                                      == maxQSize)
                                                      return TRUE
                                                  else  return FALSE
     insert(queue, item)                     ::= if(IsFull(queue)) queue Full
                                                  else insert item into queue
                                                       and return
     remove (queue)                          ::=if(!empty (queue) return item at
                                                        the front
                                                      return TRUE
                                                 else  return error
end Queue
```

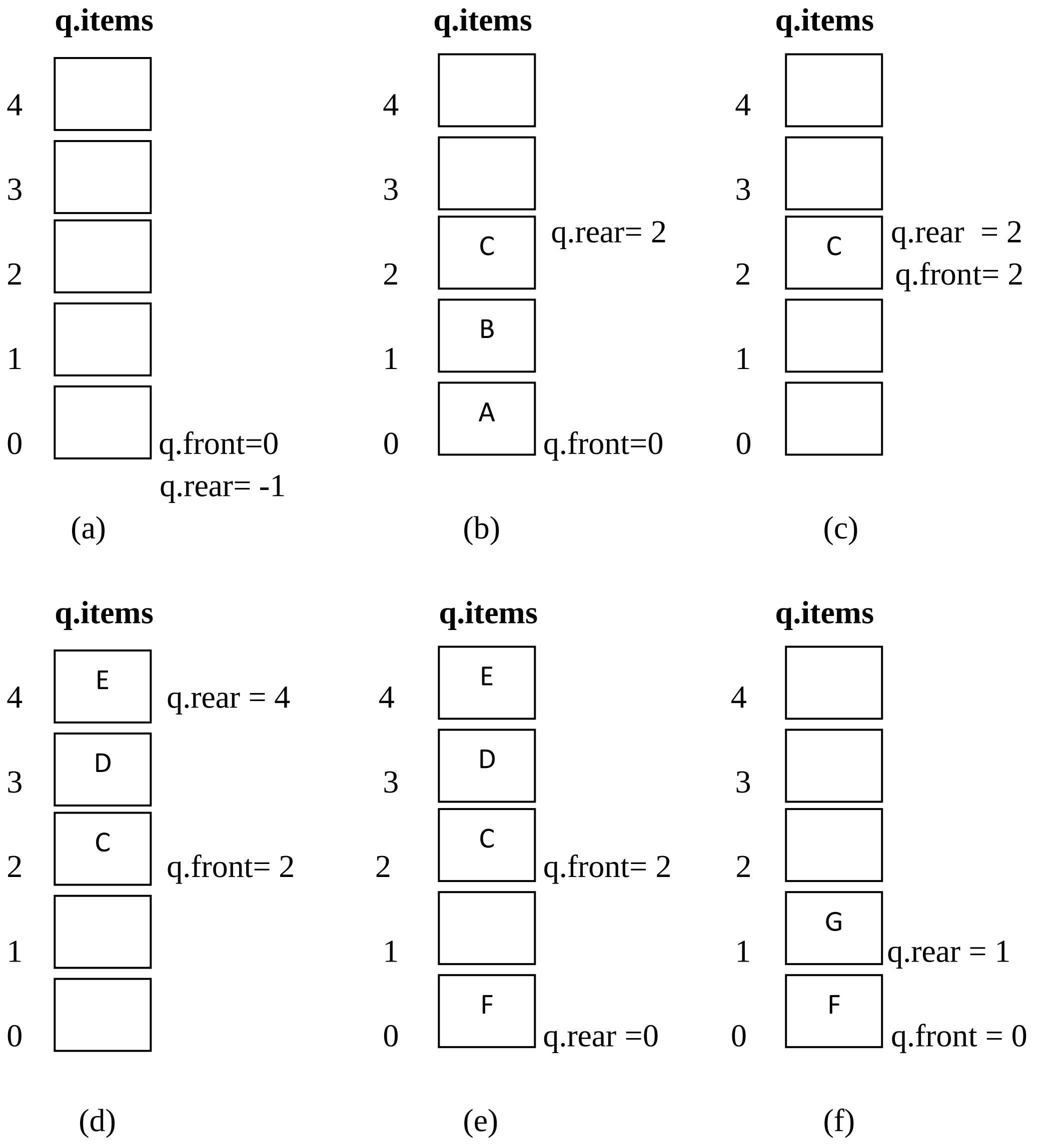
3. **Implementation of Queue in C.**

```
#define MAXQ 100
struct queue {
        int items[MAXQ] ;/* assuming that items in queue are integers */
        int front, rear ;
}q ;
q.rear =-1;
q.front=0;
```

With this definition of a queue, the functions to insert and remove, conditions for empty queue and full queue and number of elements and the will be as follows: -

(a) **insert item x :** q.items[++q.rear] = x;
(b) **remove item x :** x = q.items[q.front ++];
(c) **Empty condition :** q.rear < q.front
(d) **Full condition :** Apparently when q.rear = MAXQ-1
(e) **Number of elements :** q.rear – q.front +1

A major problem in this representation is that a queue may appear to be full while there may be empty positions available as shown in the diagram below assuming that **MAXQ=5** :



In case as shown in diagram (d), the queue is full, but there are only 3 items in the queue and positions 0 and 1 are empty. The next element F can’t be inserted.

4. **Circular Queue.** A solution to this problem is to consider a circular queue such that the next position after MAXQ -1 is 0. A circular queue may be defined as :

```
#define MAXQ 100
struct queue {
        int items[MAXQ] ;/* assuming that items in queue are integers */
        int front, rear ;
}q;
q.rear = q.front= MAXQ - 1;
```

In this representation, the conditions for empty and full queue will have to be modified because there is no rear or front. q.rear may be ahead or behind q.front as shown in diagrams (d), (e) and (f)

With this definition of a queue, the functions to insert and remove, conditions for empty queue and full queue and number of elements and the will be as follows: -

(a) **Empty Queue :**

```
empty( struct queue *pq) { /* pq is a pointer to circular queue */
        return ((pq->front = = pq->rear)?TRUE : FALSE);
}
```

(b) **Remove Item x :**

```
remove( struct queue *pq) { /* pq is a pointer to circular queue */
        if (empty (&pq)) {
           printf(" queue underflow");
           exit (1);
        }
        if (pq->front = = MAXQ -1)
           pq->front = 0;
        else
            (pq->front)++;
        return (pq->items[pq->front]);
}
```

(c) **:** q.rear < q.front

(d) **Full condition :** Apparently when q.rear = MAXQ-1

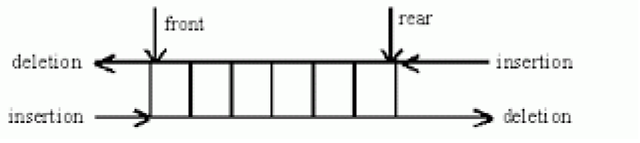
(e) **Number of elements :** q.rear – q.front +1

DEQUEUE (OR) DEQUE (DOUBLE ENDED QUEUE)

DeQueue is a data structure in which elements may be added to or deleted from the front or the rear.

Like an ordinary queue, a double-ended queue is a data structure it supports the following operations: enq_front, enq_back, deq_front, deq_back, and empty. Dequeue can be behave like a queue by using only enq_front and deq_front , and behaves like a stack by using only enq_front and deq_rear. .

The DeQueue is represented as follows.



DeQueue can be represented in two ways they are

1) Input restricted DeQueue 2) output restricted DeQueue

The out put restricted Dequeue allows deletions from only one end and input restricted Dequeue allow insertions at only one end.

The DeQueue can be constructed in two ways they are

2) Using array 2. using linked list

Algorithm to add an element into DeQueue :

Assumptions: pointer f,r and initial values are -1,-1

Q[] is an array

max represent the size of a queue

enq_front

```
step1. Start
step2. Check the queue is full or not as if (f <>
step3. If false update the pointer f as f= f-1
step4. Insert the element at pointer f as Q[f] = element
step5. Stop
```

enq_back

```
step1. Start
step2. Check the queue is full or not as if (r == max-1) if yes queue is full
step3. If false update the pointer r as r= r+1
step4. Insert the element at pointer r as Q[r] = element
```

```
step5. Stop
```

Algorithm to delete an element from the DeQueue

deq_front

```
step1. Start
step2. Check the queue is empty or not as if (f == r) if yes queue is empty
step3. If false update pointer f as f = f+1 and delete element at position f as element = Q[f]
step4. If ( f== r) reset pointer f and r as f=r=-1
step5. Stop
```

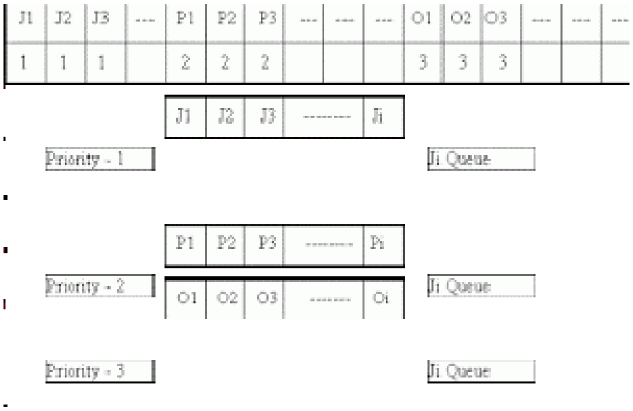
deq_back

```
step1. Start
step2. Check the queue is empty or not as if (f == r) if yes queue is empty
step3. If false delete element at position r as element = Q[r]
step4. Update pointer r as r = r-1
step5. If (f == r ) reset pointer f and r as f = r= -1
step6. Stop
```

2.9 PRIORITY QUEUE

Priority queue is a linear data structure. It is having a list of items in which each item has associated priority. It works on a principle add an element to the queue with an associated priority and remove the element from the queue that has the highest priority. In general different items may have different priorities. In this queue highest or the lowest priority item are inserted in random order. It is possible to delete an element from a priority queue in order of their priorities starting with the highest priority.

While priority queues are often implemented with heaps, they are conceptually distinct from heaps. A priority queue is an abstract concept like "a list" or "a map"; just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or a variety of other methods such as an unordered array.



Operations on Priority Queue:

A priority queue must at least support the following operations:

- insert_with_priority: add an element to the queue with an associated priority.
- pull_highest_priority_element: remove the element from the queue that has the highest priority, and return it.

  This is also known as "pop_element(Off)", "get_maximum_element" or "get_front(most)_element".

  Some conventions reverse the order of priorities, considering lower values to be higher priority, so this may also be known as "get_minimum_element", and is often referred to as "get-min" in the literature.

  This may instead be specified as separate "peek_at_highest_priority_element" and "delete_element" functions, which can be combined to produce "pull_highest_priority_element".

In addition, peek (in this context often called find-max or find-min), which returns the highest-priority element but does not modify the queue, is very frequently implemented, and nearly always executes in O(1) time. This operation and its O(1) performance is crucial to many applications of priority queues.

More advanced implementations may support more complicated operations, such as pull_lowest_priority_element, inspecting the first few highest- or lowest-priority elements, clearing the queue, clearing subsets of the queue, performing a batch insert, merging two or more queues into one, incrementing priority of any element, etc.

Similarity to Queue:

One can imagine a priority queue as a modified queue, but when one would get the next element off the queue, the highest-priority element is retrieved first.

- stack – elements are pulled in last-in first-out-order (e.g., a stack of papers)
- queue – elements are pulled in first-in first-out-order (e.g., a line in a cafeteria)

Stacks and queues may be modeled as particular kinds of priority queues. In a stack, the priority of each inserted element is monotonically increasing; thus, the last element inserted is always the first retrieved. In a queue, the priority of each inserted element is monotonically decreasing; thus, the first element inserted is always the first retrieved.

ESSENTIAL QUIZ 1

1) Which of the following is useful in traversing a given graph by breadth first search?

(a) Stack

(b) Set

(c) List

(d) Queue

2) A linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as ______________

a) Queue

b) Stack

c) Tree

d) Linked list

3) A queue follows ___________

a) FIFO (First In First Out) principle

b) LIFO (Last In First Out) principle

c) Ordered array

d) Linear tree

4) Circular Queue is also known as _________

a) Ring Buffer

b) Square Buffer

c) Rectangle Buffer

d) Curve Buffer

5) If the elements "A", "B", "C" and "D" are placed in a queue and are deleted one at a time, in what order will they be removed?

a) ABCD

b) DCBA

c) DCAB

d) ABDC

6) A data structure in which elements can be inserted or deleted at/from both ends but not in the middle is?

a) Queue

b) Circular queue

c) Dequeue

d) Priority queue

7) Queues serve major role in _______________

a) Simulation of recursion

b) Simulation of arbitrary linked list

c) Simulation of limited resource allocation

d) Simulation of heap sort

8) Which of the following is not the type of queue?

a) Ordinary queue

b) Single ended queue

c) Circular queue

d) Priority queue

9) 7. A normal queue, if implemented using an array of size MAX_SIZE, gets full when?
a) Rear = MAX_SIZE – 1
b) Front = (rear + 1)mod MAX_SIZE
c) Front = rear + 1
d) Rear = front
10) In a circular queue, how do you increment the rear end of the queue?
a) rear++
b) (rear+1) % CAPACITY
c) (rear % CAPACITY)+1
d) rear–-
11) What is the term for inserting into a full queue known as?
a) overflow
b) underflow
c) null pointer exception
d) program won't be compiled
12) What is the time complexity of enqueue operation?
a) O(logn)
b) O(nlogn)
c) O(n)
d) O(1)
13) What is the need for a circular queue?
a) effective usage of memory
b) easier computations
c) to delete elements based on priority
d) implement LIFO principle in queues
14) What is the space complexity of a linear queue having n elements?
a) O(n)
b) O(nlogn)
c) O(logn)
d) O(1)
15) In linked list implementation of queue, if only front pointer is maintained, which of the following operation take worst case linear time?
a) Insertion
b) Deletion
c) To empty a queue
d) Both Insertion and To empty a queue
16) In linked list implementation of a queue, where does a new element be inserted?
a) At the head of link list
b) At the center position in the link list
c) At the tail of the link list
d) At any position in the linked list
17) In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into a NONEMPTY queue?
a) Only front pointer
b) Only rear pointer

c) Both front and rear pointer

d) No pointer will be changed

18) In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into EMPTY queue?

a) Only front pointer

b) Only rear pointer

c) Both front and rear pointer

d) No pointer will be changed

19) In case of insertion into a linked queue, a node borrowed from the __________ list is inserted in the queue.

a) AVAIL

b) FRONT

c) REAR

d) NULL

20) In linked list implementation of a queue, from where is the item deleted?

a) At the head of link list

b) At the centre position in the link list

c) At the tail of the link list

d) Node before the tail

ESSENTIAL QUIZ 2

9) Minimum number of queues to implement stack is ___________
   a) 3
   b) 4
   c) 1
   d) 2

10) In linked list implementation of a queue, the important condition for a queue to be empty is?
   a) FRONT is null
   b) REAR is null
   c) LINK is empty
   d) FRONT==REAR-1

11) The essential condition which is checked before insertion in a linked queue is?
   a) Underflow
   b) Overflow
   c) Front value
   d) Rear value

12) The essential condition which is checked before deletion in a linked queue is?
   a) Underflow
   b) Overflow
   c) Front value
   d) Rear value

13) Which of the following is true about linked list implementation of queue?
   a) In push operation, if new nodes are inserted at the beginning of linked list, then in

pop operation, nodes must be removed from end

b) In push operation, if new nodes are inserted at the beginning, then in pop operation, nodes must be removed from the beginning

c) In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from end

d) In push operation, if new nodes are inserted at the end, then in pop operation, nodes must be removed from beginning

14) With what data structure can a priority queue be implemented?

a) Array

b) List

c) Heap

d) Tree

15) Which of the following is not an application of priority queue?

a) Huffman codes

b) Interrupt handling in operating system

c) Undo operation in text editors

d) Bayesian spam filter

16) What is the time complexity to insert a node based on key in a priority queue?

a) O(nlogn)

b) O(logn)

c) O(n)

d) O(n^2)

17) Which of the following is not an advantage of a priority queue?

a) Easy to implement

b) Processes with different priority can be efficiently handled

c) Applications with differing requirements

d) Easy to delete elements in any case

18) What is the time complexity to insert a node based on position in a priority queue?

a) O(nlogn)

b) O(logn)

c) O(n)

d) O(n^2)

19) Which one of the following is an application of Queue Data Structure?

a) When a resource is shared among multiple consumers.

b) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes

c) Load Balancing.

d) All of the above

20) How many stack are needed to implement a stack? Consider the situation where no other data structure like arrays, linked list is available to you.

a)1

b) 2

c) 3

d) 4

13) In linked list implementation of queue, if only front pointer is maintained, which of the following operation take worst case linear time?

A. Insertion
B. Deletion
C. To empty a queue
D. Both Insertion and to empty a queue

14) If the MAX_SIZE is the size of the array used in the implementation of circular queue. How is rear manipulated while inserting an element in the queue?

A. rear=(rear%1)+MAX_SIZE
B. rear=rear%(MAX_SIZE+1)
C. rear=(rear+1)%MAX_SIZE
D. rear=rear+(1%MAX_SIZE)

15) Queues serve major role in

A. Simulation of recursion
B. Simulation of arbitrary linked list
C. Simulation of limited resource allocation
D. All of the mentioned

16) ……… of the queue added a new nodes
a) Front
b) middle
c) back
d) Both A and B
17) **Let the following circular queue can accommodate maximum six elements with the following data**

front = 2 rear = 4
queue = _______; L, M, N, ___, ___
What will happen after ADD O operation takes place?
a) front = 2 rear = 5
queue = ______; L, M, N, O, ___
b) front = 3 rear = 5
queue = L, M, N, O, ___
c) front = 3 rear = 4
queue = ______; L, M, N, O, ___
d) front = 2 rear = 4
queue = L, M, N, O, ___
18) A circular queue is implemented using an array of size 10. The array index starts with 0, front is 6, and rear is 9. The insertion of next element takes place at the array index.

a) 0
b) 7
c) 9
d) 10
19) If the MAX_SIZE is the size of the array used in the implementation of circular queue, array index start with 0, front point to the first element in the queue, and rear point to the last element in the queue. Which of the following condition specify that circular queue is

EMPTY?

a) Front=rear=0
b) Front= rear=-1
c) Front=rear+1
d) Front=(rear+1)%MAX_SIZE

20) In linked list implementation of a queue, front and rear pointers are tracked. Which of these pointers will change during an insertion into a NONEMPTY queue?

a) Only front pointer
b) Only rear pointer
c) Both front and rear pointer
d) None of the front and rear pointer

ESSENTIAL QUIZ 3

1) What kind of a data structure does a queue is?
   a) Linear
   b) Non-Linear
   c) Both a and b
   d) None
2) In Queues, we can insert an element at ___ end and can delete an element at _____ end.
   a) REAR, FRONT
   b) FRONT, REAR
   c) TOP, BOTTOM
   d) BOTTOM, TOP
3) In DEQUEUEs, insertion is performed at ___ end whereas the deletion is performed at __ end.
   a) REAR, FRONT
   b) FRONT, REAR
   c) Both a and b
   d) None
4) Consider P,Q,R and S are the four elements in a queue. If we delete an element at a time then on which order they will get deleted?
   a) PQRS
   b) SRQP
   c) PSQR
   d) SRQP
5) A circular queue is implemented using an array of size 10. The array index starts with 0, front is 6, and rear is 9. The insertion of next element takes place at the array index of__.
   a) 0
   b) 7
   c) 9
   d) 10
6) What is the worst case time complexity of a sequence of n queue operations on an initially empty queue?
   a) θ (n)
   b) θ (n + k)
   c) θ (nk)
   d) θ (n2)

7) The implementation of Radix sort can be done with the help of _____.

a) Linked list

b) Stack

c) Queue

d) Possible with all the above.

8) In Queue, ENQUEUE means_____ whereas DEQUEUE refers_____.

a) an insertion operation, a deletion operation.

b) End of the queue, defining a queue.

c) Both A and B.

d) None of the above are true.

9) Dequeue is a data structure in which elements can be inserted or deleted at/from both the ends but not in the middle.

a) True

b) False

10) Both front and rear pointer will change during an insertion into a NONEMPTY queue.

a. False

b. True

11) A normal queue, if implemented using an array of size MAX_SIZE, gets full when **Rear=MAX_SIZE-1.**

a. True

b. False

12) If an array of size MAX_SIZE is used to implement a circular queue. Front, Rear, and count are tracked. Suppose front is 0 and rear is MAX_SIZE -1. MAX_SIZE elements are present in the queue.

a. True

b. False

13) What is not a disadvantage of priority scheduling in operating systems?

a) A low priority process might have to wait indefinitely for the CPU

b) If the system crashes, the low priority systems may be lost permanently

c) Interrupt handling

d) Indefinite blocking

14) Which of the following is not an advantage of a priority queue?

a) Easy to implement

b) Processes with different priority can be efficiently handled

c) Applications with differing requirements

d) Easy to delete elements in any case

15) What is the time complexity to insert a node based on position in a priority queue?

a) O(nlogn)

b) O(logn)

c) O(n)

d) $O(n^2)$

16) In linked list implementation of a queue, the important condition for a queue to be empty is?

a) FRONT is null

b) REAR is null

c) LINK is empty

d) FRONT==REAR-1

17) In a circular queue, how do you increment the rear end of the queue?

a) rear++

b) (rear+1) % CAPACITY

c) (rear % CAPACITY)+1

d) rear–

18) What does the following Java code do?

```
public Object function()
{
   if(isEmpty())
   return -999;
   else
   {
                  Object high;
                  high = q[front];
                  return high;
   }
}
```

a) Dequeue

b) Enqueue

c) Return the front element

d) Return the last element

19) What is the need for a circular queue?

a) effective usage of memory

b) easier computations

c) to delete elements based on priority

d) implement LIFO principle in queues

20) What is the space complexity of a linear queue having n elements?

a) O(n)

b) O(nlogn)

c) O(logn)

d) O(1)

ADVANCED QUIZ 1

1. Is it possible to implement a queue using Linked List ?. Enqueue & Dequeue should be O(1).

a. Not possible to implement.
b Only Enqueue is possible at O(1).
c. Only Dequeue is possible at O(1).
d. Both Enqueue and Dequeue is possible at O(1)

EXPLANATION: Have two pointers H pointing to the Head and T pointing to the Tail of the linked list. Perform enqueue at T and perform dequeue at H. Update the pointers after each operations accordingly.

2. Following is C like pseudo code of a function that takes a Queue as an argument, and uses a stack S to do processing.

```
void fun(Queue *Q)
{
    Stack S;  // Say it creates an empty stack S

    // Run while Q is not empty
    while (!isEmpty(Q))
    {
        // deQueue an item from Q and push the dequeued item to S
        push(&S, deQueue(Q));
    }

    // Run while Stack S is not empty
    while (!isEmpty(&S))
    {
      // Pop an item from S and enqueue the poppped item to Q
      enQueue(Q, pop(&S));
    }
}
```

What does the above function do in general?
a. Not possible to implement.
b Removes the last element fron Q
c. Make the Q empty
d. Reverses the Q

3. An implementation of a queue Q, using two stacks S1 and S2, is given below:

```
void insert(Q, x) {
   push (S1, x);
}

void delete(Q){
   if(stack-empty(S2)) then
      if(stack-empty(S1)) then {
          print("Q is empty");
          return;
      }
      else while (!(stack-empty(S1))){
          x=pop(S1);
          push(S2,x);
      }
   x=pop(S2);
}
```

Let n insert and m (<=n) delete operations be performed in an arbitrary order on an empty queue Let x and y be the number of push and pop operations performed respectively in the process. Which one of the following is true for all m and n?

i) n+m <= x < 2n and 2m <= y <= n+m
j) n+m <= x < 2n and 2m<= y <= 2n
k) 2m <= x < 2n and 2m <= y <= n+m
l) 2m <= x <2n and 2m <= y <= 2n

4. Consider the following operation along with Enqueue and Dequeue operations on queues, where k is a global parameter.

```
MultiDequeue(Q){
  m = k
  while (Q is not empty and m  > 0) {
    Dequeue(Q)
    m = m - 1
  }
}
```

What is the worst case time complexity of a sequence of n MultiDequeue() operations on an initially empty queue?

a) $\Theta(n)$
b) $\Theta(n+k)$
c) $\Theta(nk)$
d) $\Theta(n^2)$

5. Consider the following pseudo code. Assume that IntQueue is an integer queue. What does the function fun do?

```
void fun(int n)
{
   IntQueue q = new IntQueue();
   q.enqueue(0);
   q.enqueue(1);
   for (int i = 0; i < n; i++)
   {
      int a = q.dequeue();
      int b = q.dequeue();
      q.enqueue(b);
      q.enqueue(a + b);
      ptint(a);
   }
}
```

a) Prints number 0 to n-1
b) Prints number n to 0
c) Prints first n Fibonacci numbers
d) Prints first n Fibonacci numbers in reverse order

6. Suppose implementation supports an instruction REVERSE, which reverses the order of elements on the stack, in addition to the PUSH and POP instructions. Which one of the following statements is TRUE with respect to this modified stack?

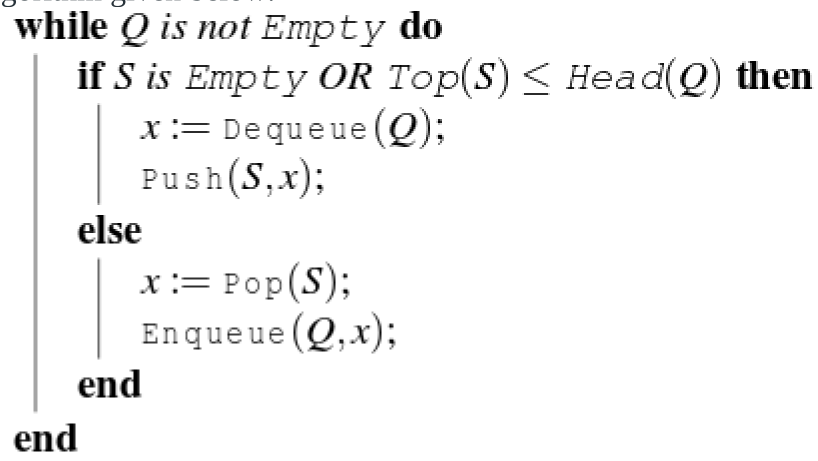
a) A queue cannot be implemented using this stack
b) A queue can be implemented where ENQUEUE takes a single instruction and DEQUEUE takes a sequence of two instructions.
c) A queue can be implemented where ENQUEUE takes a sequence of three instructions and DEQUEUE takes a single instruction.

d) queue can be implemented where both ENQUEUE and DEQUEUE take a single instruction each.

7. A queue is implemented using an array such that ENQUEUE and DEQUEUE operations are performed efficiently. Which one of the following statements is CORRECT (n refers to the number of items in the queue)?

a) Both operations can be performed in O(1) time.
b) At most one operation can be performed in O(1) time but the worst case time for the other operation will be Ω(n)
c) The worst case time complexity for both operations will be Ω(n)
d) Worst case time complexity for both operations will be Ω(log n)

8. Let Q denote a queue containing sixteen numbers and S be an empty stack. Head(Q) returns the element at the head of the queue Q **without** removing it from Q. Similarly Top(S) returns the element at the top of S **without** removing it from S. Consider the algorithm given below.



```
while $Q$ is not $Empty$ do
    if $S$ is $Empty$ OR $Top(S) \leq Head(Q)$ then
        $x :=$ Dequeue$(Q)$;
        Push$(S, x)$;
    else
        $x :=$ Pop$(S)$;
        Enqueue$(Q, x)$;
    end
end
```

The maximum possible number of iterations of the while loop in the algorithm is______

a) 16
b) 32
c) 256
d) 64

9. Suppose you are given an implementation of a queue of integers. The operations that can be performed on the queue are:
i. isEmpty (Q) — returns true if the queue is empty, false otherwise.
ii. delete (Q) — deletes the element at the front of the queue and returns its value.
iii. insert (Q, i) — inserts the integer i at the rear of the queue.
Consider the following function:

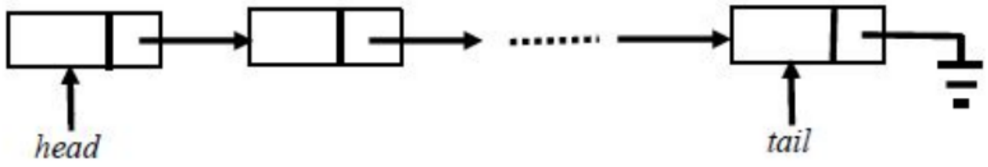
```
void f (queue Q) {
int i ;
if (!isEmpty(Q)) {
  i = delete(Q);
  f(Q);
  insert(Q, i);
 }
   }
```

What operation is performed by the above function f ?

a) Leaves the queue Q unchanged

b) Reverses the order of the elements in the queue Q
c) Deletes the element at the front of the queue Q and inserts it at the rear keeping the other elements in the same order
d) Empties the queue Q

10. Consider the following statements:
    i. First-in-first out types of computations are efficiently supported by STACKS.
    ii. Implementing LISTS on linked lists is more efficient than implementing LISTS on an array for almost all the basic LIST operations.
    iii. Implementing QUEUES on a circular array is more efficient than implementing QUEUES on a linear array with two indices.
    iv. Last-in-first-out type of computations are efficiently supported by QUEUES.
    Which of the following are true?
    a) (i) and (iii) are true
    b) (ii) and (iii) are true
    c) (iii) and (iv) are true
    d) (ii) and (iv) are true
11. Which of the following option is not correct?
    a) If the queue is implemented with a linked list, keeping track of a front pointer, Only rear pointer s will change during an insertion into an non-empty queue.
    b) Queue data structure can be used to implement least recently used (LRU) page fault algorithm and Quick short algorithm.
    c) Queue data structure can be used to implement Quick short algorithm but not least recently used (LRU) page fault algorithm.
    d) Both a and c are correct
12. A queue is implemented using a non-circular singly linked list. The queue has a head pointer and a tail pointer, as shown in the figure. Let n denote the number of nodes in the queue. Let 'enqueue' be implemented by inserting a new node at the head, and 'dequeue' be implemented by deletion of a node from the tail.



Which one of the following is the time complexity of the most time-efficient implementation of 'enqueue' and 'dequeue, respectively, for this data structure?

a) Θ(1), Θ(1)
b) Θ(1), Θ(n)
c) Θ(n), Θ(1)
d) Θ(n), Θ(n)

13. A priority queue is implemented as a max-heap. Initially, it has five elements. The level-order traversal of the heap is as follows: 20, 18, 15, 13, 12 Two new elements ‘10’ and ‘17’ are inserted in the heap in that order. The level-order traversal of the heap after the insertion of the element is:
    a) 20, 18, 17, 15, 13, 12, 10
    b) 20, 18, 17, 12, 13, 10, 15
    c) 20, 18, 17, 10, 12, 13, 15
    d) 20, 18, 17, 13, 12, 10, 15

14. Consider a standard Circular Queue 'q' implementation (which has the same condition for Queue Full and Queue Empty) whose size is 11 and the elements of the queue are q[0], q[1], q[2].....,q[10]. The front and rear pointers are initialized to point at q[2] . In which position will the ninth element be added?

a) q[0]
b) q[1]
c) q[2]
d) q[10]

15. What is the functionality of the following piece of code?

```
public Object delete_key()
{
		if(count == 0)
		{
				System.out.println("Q is empty");
				System.exit(0);
		}
		else
		{
				Node cur = head.getNext();
				Node dup = cur.getNext();
				Object e = cur.getEle();
				head.setNext(dup);
				count--;
				return e;
		}
}
```

a) Delete the second element in the list
b) Return but not delete the second element in the list
c) Delete the first element in the list
d) Return but not delete the first element in the list

16. A priority queue Q is used to implement a stack S that stores characters. PUSH(C) is implemented as INSERT(Q, C, K) where K is an appropriate integer key chosen by the implementation. POP is implemented as DELETEMIN(Q). For a sequence of operations, the keys chosen are in:

a) Non-increasing order
b) Non-decreasing order
c) strictly increasing order
d)strictly decreasing order

17. Consider the following sequence of operations on an empty stack.

```
Push(54);push(52);pop();push(55);push(62);s=pop();
```

Consider the following sequence of operations on an empty queue.

```
enqueue(21);enqueue(24);dequeue();enqueue(28);enqueue(32);q=dequeue();
```

The value of s+q is ____________.

e) **86**
f) **68**
g) **24**
h) **94**

18. Suppose a circular queue of capacity (n – 1) elements is implemented with an array of n elements. Assume that the insertion and deletion operation are carried out using REAR and

FRONT as array index variables, respectively. Initially, REAR = FRONT = 0. The conditions to detect queue full and queue empty are

a) Full: (REAR+1) mod n == FRONT, empty: REAR == FRONT
b) Full: (REAR+1) mod n == FRONT, empty: (FRONT+1) mod n == REAR
c) Full: REAR == FRONT, empty: (REAR+1) mod n == FRONT
d) Full: (FRONT+1) mod n == REAR, empty: REAR == FRONT

19. A Priority-Queue is implemented as a Max-Heap. Initially, it has 5 elements. The level-order traversal of the heap is given below: 10, 8, 5, 3, 2 Two new elements ”1‘ and ”7‘ are inserted in the heap in that order. The level-order traversal of the heap after the insertion of the elements is:

a) 10, 8, 7, 5, 3, 2, 1
b) 10, 8, 7, 3, 2, 1, 5
c) 1 0, 8, 7, 1, 2, 3, 5
d) 10, 8, 7, 3, 2, 1, 5

20. A priority queue can efficiently implement using which of the following data structures? Assume that the number of insert and peek (operation to see the current highest priority item) and extraction (remove the highest priority item) operations are almost same.

a) Array
b) Linked Lists
c) Heap Data Structures like Binary Heap, Fibonacci Heap
d) None of these

ADVANCED QUIZ 2

1) 1. A Double-ended queue supports operations such as adding and removing items from both the sides of the queue. They support four operations like addFront(adding item to top of the queue), addRear(adding item to the bottom of the queue), removeFront(removing item from the top of the queue) and removeRear(removing item from the bottom of the queue). You are given only stacks to implement this data structure. You can implement only push and pop operations. What are the total number of stacks required for this operation?(you can reuse the stack)

a) 1
b) 2
c) 3
d) 4

2) You are asked to perform a queue operation using a stack. Assume the size of the stack is some value ‘n’ and there are ‘m’ number of variables in this stack. The time complexity of performing deQueue operation is (Using only stack operations like push and pop)(Tightly bound).

a) O(m)
b) O(n)
c) O(m*n)
d) Data is insufficient

3) Consider you have an array of some random size. You need to perform dequeue operation. You can perform it using stack operation (push and pop) or using queue operations itself (enQueue and Dequeue). The output is guaranteed to be same. Find some differences?

a) They will have different time complexities

b) The memory used will not be different

c) There are chances that output might be different

d) No differences

4) A double-ended queue supports operations like adding and removing items from both the sides of the queue. They support four operations like addFront(adding item to top of the queue), addRear(adding item to the bottom of the queue), removeFront(removing item from the top of the queue) and removeRear(removing item from the bottom of the queue). You are given only stacks to implement this data structure. You can implement only push and pop operations. What's the time complexity of performing addFront and addRear? (Assume 'm' to be the size of the stack and 'n' to be the number of elements)

a) O(m) and O(n)

b) O(1) and O(n)

c) O(n) and O(1)

d) O(n) and O(m)

5) Why is implementation of stack operations on queues not feasible for a large dataset (Asssume the number of elements in the stack to be n)?

a) Because of its time complexity O(n)

b) Because of its time complexity O(log(n))

c) Extra memory is not required

d) There are no problems

6) Consider yourself to be in a planet where the computational power of chips to be slow. You have an array of size 10.You want to perform enqueue some element into this array. But you can perform only push and pop operations .Push and pop operation both take 1 sec respectively. The total time required to perform enQueue operation is?

a) 20

b) 40

c) 42

d) 43

7) You have two jars, one jar which has 10 rings and the other has none. They are placed one above the other. You want to remove the last ring in the jar. And the second jar is weak and cannot be used to store rings for a long time.

a) Empty the first jar by removing it one by one from the first jar and placing it into the second jar

b) Empty the first jar by removing it one by one from the first jar and placing it into the second jar and empty the second jar by placing all the rings into the first jar one by one

c) There exists no possible way to do this

d) Break the jar and remove the last one

8) Select the function which performs insertion at the front end of the dequeue? a)

```
public void function(Object item)
{
        Node temp = new Node(item,null);
        if(isEmpty())
        {
                temp.setNext(trail);
                head.setNext(temp);
        }
        else
        {
                Node cur = head.getNext();
                temp.setNext(cur);
                head.setNext(temp);
        }
        size++;
}
```

b)

```
public void function(Object item)
{
        Node temp = new Node(item,null);
        if(isEmpty())
        {
                temp.setNext(trail);
                head.setNext(trail);
        }
        else
        {
                Node cur = head.getNext();
                temp.setNext(cur);
                head.setNext(temp);
        }
        size++;
}
```

c)

```
public void function(Object item)
{
        Node temp = new Node(item,null);
        if(isEmpty())
        {
                Node cur = head.getNext();
                temp.setNext(cur);
                head.setNext(temp);
        }
        else
        {
                temp.setNext(trail);
                head.setNext(temp);
        }
        size++;
}
```

d) None of these

9) Which of the following can be used to delete an element from the front end of the queue? a)

```
public Object deleteFront() throws emptyDEQException
{
        if(isEmpty())
                throw new emptyDEQException("Empty");
        else
        {
                Node temp = head.getNext();
                Node cur = temp;
                Object e = temp.getEle();
                head.setNext(cur);
                size--;
                return e;
        }
}
```

b)

```
public Object deleteFront() throws emptyDEQException
{
        if(isEmpty())
                throw new emptyDEQException("Empty");
        else
        {
                Node temp = head.getNext();
                Node cur = temp.getNext();
                Object e = temp.getEle();
                head.setNext(cur);
                size--;
                return e;
        }
}
```

c)

```
public Object deleteFront() throws emptyDEQException
{
        if(isEmpty())
                throw new emptyDEQException("Empty");
        else
        {
                Node temp = head.getNext();
                Node cur = temp.getNext();
                Object e = temp.getEle();
                head.setNext(temp);
                size--;
                return e;
        }
}
```

d)

```
public Object deleteFront() throws emptyDEQException
{
        if(isEmpty())
                throw new emptyDEQException("Empty");
        else
        {
                Node temp = head.getNext();
                Node cur = temp.getNext();
                Object e = temp.getEle();
                temp.setNext(cur);
                size--;
                return e;
        }
}
```

10) What is the time complexity of deleting from the rear end of the dequeue implemented with a singly linked list?

a) O(nlogn)

b) O(logn)

c) O(n)

d) O(n^2)

11) After performing these set of operations, what does the final list look contain?

```
InsertFront(10);
InsertFront(20);
InsertRear(30);
DeleteFront();
InsertRear(40);
InsertRear(10);
DeleteRear();
InsertRear(15);
display();
```

a) 10 30 10 15
b) 20 30 40 15
c) 20 30 40 10
d) 10 30 40 15

12) Select the appropriate code that inserts elements into the list based on the given key value. (head and trail are dummy nodes to mark the end and beginning of the list, they do not contain any priority or element)

```
a) public void insert_key(int key,Object item)
{
        if(key<0)
        {
                Systerm.our.println("invalid");
                System.exit(0);
        }
        else
        {
                Node temp = new Node(key,item,null);
                if(count == 0)
                {
                        head.setNext(temp);
                        temp.setNext(trail);
                }
                else
                {
```

```
            Node dup = head.getNext();
            Node cur = head;
            while((key>dup.getKey()) && (dup!=trail))
            {
                dup = dup.getNext();
                cur = cur.getNext();
            }
            cur.setNext(temp);
            temp.setNext(dup);
        }
        count++;
    }
}
```

b)

```
public void insert_key(int key,Object item)
{
    if(key<0)
    {
        Systerm.our.println("invalid");
        System.exit(0);
    }
    else
    {
        Node temp = new Node(key,item,null);
        if(count == 0)
        {
            head.setNext(temp);
            temp.setNext(trail);
        }
        else
        {
            Node dup = head.getNext();
            Node cur = dup;
            while((key>dup.getKey()) && (dup!=trail))
            {
                dup = dup.getNext();
                cur = cur.getNext();
            }
            cur.setNext(temp);
            temp.setNext(dup);
        }
        count++;
    }
}
```

c)

```
public void insert_key(int key,Object item)
{   if(key<0)
    {
        Systerm.our.println("invalid");
        System.exit(0);
    }
    else
    {
        Node temp = new Node(key,item,null);
        if(count == 0)
        {
            head.setNext(temp);
            temp.setNext(trail);
        }
        else
        {
            Node dup = head.getNext();
            Node cur = head;
            while((key>dup.getKey()) && (dup!=trail))
            {
                dup = dup.getNext();
                cur = cur.getNext();
            }
            cur.setNext(dup);
            temp.setNext(cur);
        }
        count++;
    }
}
```

d)

```
public void insert_key(int key,Object item)
{
    if(key<0)
    {
        Systerm.our.println("invalid");
        System.exit(0);
    }
    else
    {
        Node temp = new Node(key,item,null);
        if(count == 0)
        {
            head.setNext(temp);
            temp.setNext(trail);
        }
        else
        {
            Node dup = head.getNext();
            Node cur = head;
            while((key>dup.getKey()) && (dup!=trail))
            {
                dup = cur
                cur = cur.getNext();
            }
            cur.setNext(dup);
            temp.setNext(cur);
        }
        count++;
    }
}
```

13) What is the functionality of the following piece of code?

```
public Object delete_key()
{
        if(count == 0)
        {
                System.out.println("Q is empty");
                System.exit(0);
        }
        else
        {
                Node cur = head.getNext();
                Node dup = cur.getNext();
                Object e = cur.getEle();
                head.setNext(dup);
                count--;
                return e;
        }
}
```

a) Delete the second element in the list
b) Return but not delete the second element in the list
c) Delete the first element in the list
d) Return but not delete the first element in the list

14) Which of the following represents a dequeue operation? (count is the number of elements in the queue)

a)

```
public Object dequeue()
{
        if(count == 0)
        {
                System.out.println("Queue underflow");
                return 0;
        }
        else
        {
                Object ele = q[front];
                q[front] = null;
                front = (front+1)%CAPACITY;
                count--;
                return ele;
        }
}
```

b)

```
public Object dequeue()
{
        if(count == 0)
        {
                System.out.println("Queue underflow");
                return 0;
        }
        else
        {
                Object ele = q[front];
                front = (front+1)%CAPACITY;
                q[front] = null;
                count--;
                return ele;
        }
}
```

c)

```
public Object dequeue()
{
        if(count == 0)
        {
                System.out.println("Queue underflow");
                return 0;
        }
        else
        {
                front = (front+1)%CAPACITY;
                Object ele = q[front];
                q[front] = null;
                count--;
                return ele;
        }
}
```

d)

```
public Object dequeue()
{
        if(count == 0)
        {
                System.out.println("Queue underflow");
                return 0;
        }
        else
        {
                Object ele = q[front];
                q[front] = null;
                front = (front+1)%CAPACITY;
                return ele;
                count--;
        }
}
```

15) Which of the following best describes the growth of a linear queue at runtime? (Q is the original queue, size() returns the number of elements in the queue)

a)

```
private void expand()
{
        int length = size();
        int[] newQ = new int[length<<1];
        for(int i=front; i<=rear; i++)
        {
                newQ[i-front] = Q[i%CAPACITY];
        }
        Q = newQ;
        front = 0;
        rear = size()-1;
}
```

b)

```
private void expand()
{
        int length = size();
        int[] newQ = new int[length<<1];
        for(int i=front; i<=rear; i++)
        {
                newQ[i-front] = Q[i%CAPACITY];
        }
        Q = newQ;
}
```

c)

```
private void expand()
{
        int length = size();
        int[] newQ = new int[length<<1];
        for(int i=front; i<=rear; i++)
        {
                newQ[i-front] = Q[i];
        }
        Q = newQ;
        front = 0;
        rear = size()-1;
}
```

d)

```
private void expand()
{
        int length = size();
        int[] newQ = new int[length*2];
        for(int i=front; i<=rear; i++)
        {
                newQ[i-front] = Q[i%CAPACITY];
        }
        Q = newQ;
}
```

16) What is the output of the following Java code?

```
public class CircularQueue
{
        protected static final int CAPACITY = 100;
        protected int size,front,rear;
        protected Object q[];
        int count = 0;

        public CircularQueue()
```

```
    {
        this(CAPACITY);
    }
    public CircularQueue (int n)
    {
        size = n;
        front = 0;
        rear = 0;
        q = new Object[size];
    }


    public void enqueue(Object item)
    {
        if(count == size)
        {
            System.out.println("Queue overflow");
                return;
        }
        else
        {
            q[rear] = item;
            rear = (rear+1)%size;
            count++;
        }
    }
    public Object dequeue()
    {
        if(count == 0)
        {
            System.out.println("Queue underflow");
            return 0;
        }
        else
        {
            Object ele = q[front];
            q[front] = null;
            front = (front+1)%size;
            count--;
            return ele;
        }
    }
    public Object frontElement()
    {
        if(count == 0)
        return -999;
        else
        {
            Object high;
            high = q[front];
            return high;
        }
    }
    public Object rearElement()
    {
        if(count == 0)
        return -999;
        else
        {
            Object low;
            rear = (rear-1)%size;
            low = q[rear];
            rear = (rear+1)%size;
            return low;
        }
    }
}
public class CircularQueueDemo
{
    public static void main(String args[])
    {
        Object var;
        CircularQueue myQ = new CircularQueue();
        myQ.enqueue(10);
        myQ.enqueue(3);
        var = myQ.rearElement();
        myQ.dequeue();
        myQ.enqueue(6);
        var = mQ.frontElement();
        System.out.println(var+" "+var);
    }
}
```

a) 3 3
b) 3 6
c) 6 6
d) 10 6

17) What does the following Java code do?

```
public Object function()
{
    if(isEmpty())
    return -999;
    else
    {
        Object high;
        high = q[front];
        return high;
    }
}
```

a) Dequeue
b) Enqueue
c) Return the front element
d) Return the last element

18) Given only a single array of size 10 and no other memory is available. Which of the following operation is not feasible to implement (Given only push and pop operation)?

a) Push
b) Pop
c) Enqueue
d) Returntop
Explanation: To perform Enqueue using just push and pop operations, there is a need of another array of same size. But as there is no extra available memory, the given operation is not feasible.

QUESTION ANSWER QUEUE

1. **What is a Queue?**

A Queue is an ordered collection of items from which items may be deleted at one end called the front of the queue and into which terms may be inserted at the other end called rear of the queue. Queue is called as First –in-First-Out (FIFO).

2. **What is a Priority Queue?**

Priority queue is a data structure in which the intrinsic ordering of the elements does determine the results of its basic operations. Ascending and Descending priority queue are the two types of Priority queue.

3. **Write down the operations that can be done with queue data structure?**

Queue is a first - in -first out list. The operations that can be done with queue are insert and remove.

4. **What is a circular queue?**

The queue, which wraps around upon reaching the end of the array is called as circular queue.

5. **Give few examples for data structures?**

- Stacks
- Queue
- Linked list
- Trees
- Graphs

6. **List out Applications of queue**

Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
Computer systems must often provide a “h processes, two programs, or even two systems. This holding area is usually called a “buffer” and is often implemented as a queue.

7. **How do you test for an empty queue?**

To test for an empty queue, we have to check whether READ=HEAD where REAR is a pointer pointing to the last node in a queue and HEAD is a pointer that pointer to the dummy header.

In the case of array implementation of queue, the condition to be checked for an empty queue is READ<FRONT.

8. Which data structures are used for BFS and DFS of a graph?

- Queue is used for BFS
- Stack is used for DFS. DFS can also be implemented using recursion (Note that recursion also uses function call stack).

9.Which Data Structure Should be used for implementing LRU cache?

We use two data structures to implement an LRU Cache.

***Queue** which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size). The most recently used pages will be near rear end and least recently pages will be near front end.

***Hash** with page number as key and address of the corresponding queue node as value.

10. What are the minimum number of Queues needed to implement the priority queue?

Two queues are needed. One queue is used to store the data elements, and another is used for storing priorities.

11. Is it possible to implement a queue using Linked List? Enqueue & Dequeue should be O (1).

Both Enqueue and Dequeue is possible at O (1)

EXPLANATION: Have two pointers H pointing to the Head and T pointing to the Tail of the linked list. Perform enqueue at T and perform dequeue at H. Update the pointers after each operation accordingly.

12. How many stacks are required to implement a Queue.

2 stacks S1 and S2 are required.

For Enqueue, perform push on S1.

For Dequeue, if S2 is empty pop all the elements from S1 and push it to S2. The last element you popped from S1 is an element to be dequeued. If S2 is not empty, then pop the top element in it.

13. Why and when should I use Stack or Queue data structures instead of Arrays/Lists?

Because they help manage your data in more a *particular* way than arrays and lists. It means that when you're debugging a problem, you won't have to wonder if someone randomly inserted an element into the middle of your list, messing up some invariants.

Arrays and lists are random access. They are very flexible and also easily *corruptible*. If you want to manage your data as FIFO or LIFO it's best to use those, already implemented, collections.

More practically you should:

- Use a queue when you want to get things out in the order that you put them in (FIFO)
- Use a stack when you want to get things out in the reverse order than you put them in (LIFO)
- Use a list when you want to get anything out, regardless of when you put them in (and when you don't want them to automatically be removed).

14. Explain the Concept of a Queue. How can you Differentiate it from a Stack?

Answer– A queue is a type of linear structure which is used to access components that support the *FIFO (First In First Out)* method. Dequeue, enqueue, front, and rear are key queue functions. Unlike a stack, the arrays and linked lists are used to enforce a queue.

The element most recently added is removed first in a stack. However, in the event of a queue, the element least recently added is removed first.

15. List some applications of queue data structure.

The Applications of the queue is given as follows:

- Queues are widely used as waiting lists for a single shared resource like a printer, disk, CPU.
- Queues are used in the asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
- Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
- Queues are used to maintain the playlist in media players to add and remove the songs from the play-list.
- Queues are used in operating systems for handling interrupts.

16. What are the drawbacks of array implementation of Queue?

- **Memory Wastage:** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.
- **Array Size:** There might be situations in which, we may need to extend the queue to insert more elements if we use an array to implement queue, It will almost be impossible to extend the array size, therefore deciding the correct array size is always a problem in array implementation of queue.

17. What are the scenarios in which an element can be inserted into the circular queue?

- If (rear + 1)%maxsize = front, the queue is full. In that case, overflow occurs and therefore, insertion can not be performed in the queue.
- If rear != max - 1, the rear will be incremented to the mod(maxsize) and the new value will be inserted at the rear end of the queue.
- If front != 0 and rear = max - 1, it means that queue is not full therefore, set the value of rear to 0 and insert the new element there.

18.Write algorithm for ENQUEUE operation.

```
newNode -> data = data
newNode -> next = NULL
if ( REAR == NULL)
FRONT = REAR = newNode
end if
else
REAR -> next = newNode
REAR = newNode
end Algorithm_Enqueue
```

19.Write algorithm for DEQUEUE operation.

```
if(FRONT == NULL)
print "EMPTY QUEUE" and exit.
end if
end Algorithm_Dequeue
else
temp = FRONT
```

```
FRONT = FRONT -> NEXT
free(temp)
end else
end Algorithm_Dequeue
```

20. Write priority queue operations.
A typical priority queue supports the following operations.
1) insert(item, priority): Inserts an item with given priority.
2) getHighestPriority(): Returns the highest priority item.
3) deleteHighestPriority(): Removes the highest priority item.

21. What are the applications of Priority Queue:
1) CPU Scheduling
2) Graph algorithms like Dijkstra’s shortest path algorithm, Prim’s Minimum Spanning Tree, etc
3) All queue applications where priority is involved.

22. WAP to reverse a queue using a stack.

```
/* Function to check if the queue is empty */
int empty()
{
if(st.front == -1)
return 1;
else
return 0;
}

/* Function to insert elements in a queue */
void enqueue(int num)
{
if(st.front == -1)
st.front++;
st.rear++;
st.s[st.rear] = num;
}

/* Function to delete elements from the queue */
int dequeue()
{
int x;
x = st.s[st.front];
if(st.front==st.rear)
st.front=st.rear=-1;
else
st.front++;
return x;
}

/* Function to display queue elements */
```

```
void display()
{
int i;
if(empty())
printf("\nEMPTY QUEUE\n");
else
{
printf("\nQUEUE ELEMENTS : ");
for(i = st.front ; i <= st.rear ; i++)
printf("%d ",st.s[i]);
}
printf("\n");
}

/* Function to reverse a queue using a stack */
void reverse_queue_using_stack()
{
while(!(st.front == st.rear))
{stack1.push(dequeue());}
stack1.push(dequeue());
printf("\nREVERSED QUEUE : ");
while(!stack1.empty())
{
printf("%d ",stack1.top());   // Print the top element of the stack
stack1.pop();
}
printf("\n");
exit(0);
}

/* Main function */
int main()
{
int num,choice;
st.front = st.rear = -1;
printf("\nREVERSING A QUEUE USING STACKS\n");
printf("\n1.ENQUEUE\n2.DEQUEUE\n3.DISPLAY\n4.REVERSE\n5.EXIT\n");
while(1)
{
printf("\nEnter the choice : ");
scanf("%d",&choice);
switch (choice)
{
case 1:
if(full())
{
printf("\nQUEUE IS FULL\n");
}
else
{
printf("\nEnter data : ");
scanf("%d",&num);
enqueue(num);
}
break;
case 2:
if (empty())
{
printf("\nEMPTY QUEUE\n");
}
else
printf("\nDequeued Element : %d",dequeue());
break;
case 3:
display();
break;
case 4: reverse_queue_using_stack();
break;
default: exit(0);
}}
return 0;
}
```

23. What are the various operations of Deque.

Mainly the following four basic operations are performed on deque.
insetFront(): Adds an item at the front of the Deque.
insertLast(): Adds an item at the rear of the Deque.
deleteFront(): Deletes an item from front of the Deque.
deleteLast(): Deletes an item from rear of the Deque.

24. **What are the various applications of Deque.**
Applications of deque

- Palindrome checker.
- A-steal job scheduling algorithm

25. What is the time complexity to insert a node based on key in a priority queue?
O(n), because in the worst case, you might have to traverse the entire list.

CHAPTER 6: GRAPH

Graphs

- Data sometimes contains a relationship between pairs of elements which is not necessarily hierarchical in nature, e.g. an airline flights only between the cities connected by lines. This data structure is called Graph.
- A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices.
- A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.
- A graph may be either undirected or directed. Intuitively, an undirected edge models a "twoway" or "duplex" connection between its endpoints, while a directed edge is a one-way connection, and is typically drawn as an arrow.
- A directed edge is often called an arc. Mathematically, an undirected edge is an unordered pair of vertices, and an arc is an ordered pair. The maximum number of edges in an undirected graph without a self-loop isn(n - 1)/2 while a directed graph can have at most n 2 edges
- Graphs can be classified by whether or not their edges have weights. In Weighted graph, edges have a weight.

Graph Terminology

- **Adjacent Vertices**:- Vertex v_1 is said to be adjacent to a vertex v_2 if there is an edge(v_1, v_2) or (v_2, v_1).
- **Path**:- A path from vertex w is a sequence of vertices, each adjacent to the next.
- **Cycle**:- A cycle is a path in which first and last vertices are the same.
- **Connected graph**:- A graph is called connected if there exists a path from any vertex to any other vertex. These are graphs which are unconnected.
- **Degree**:- The number of edges incident on a vertex determine its degree.

- **Complete graph**:- A graph G is said to be complete if all its nodes are fully connected. That is, there is a path from one node to every other node in the graph. A complete graph has n(n–1)/2 edges, where n is the number of nodes in G.
- **Weighted graph:-** A graph is said to be weighted graph if every edge in the graph is assigned some weight or value.
- **Tree**:- A graph is a tree if it has two properties :
    - It is connected, and
    - There are no cycles in the graph.

Graph Representation

There are two standard ways of maintaining a graph G in the memory of a computer.

- The sequential representation (Adjacency Matrix)
- The linked representation (Adjacency List)

Adjacency Matrix Representation

- An adjacency matrix is one of the two common ways to represent a graph. The adjacency matrix shows which nodes are adjacent to one another.
- Two nodes are adjacent if there is an edge connecting them. In the case of a directed graph, if node j is adjacent to node i, there is an edge from i to j . In other words, if j is adjacent to i, you can get from i to j by traversing one edge.
- For a given graph with n nodes, the adjacency matrix will have dimensions of nxn. For an unweighted graph, the adjacency matrix will be populated with Boolean values.

Adjacency List Representation

- The adjacency list is another common representation of a graph. There are many ways to implement this adjacency representation.
- One way is to have the graph maintain a list of lists, in which the first list is a list of indices corresponding to each node in the graph. Each of these refer to another list that stores a the index of each adjacent node to this one.
- It might also be useful to associate the weight of each link with the adjacent node in this list.

Adjacency Multi-lists

- Adjacency Multi-lists are an edge, rather than vertex based, graph representation. In the Multilist representation of graph structures; these are two parts, a directory of Node information and a set of linked list of edge information.
- There is one entry in the node directory for each node of the graph. The directory entry for node i points to a linked adjacency list for node i. each record of the linked list area appears on two adjacency lists: one for the node at each end of the represented edge.

Graph Traversal

Graph traversal is the problem of visiting all the nodes in a graph in a particular manner, updating and/or checking their values along the way. The order in which the vertices are visited may be important, and may depend upon the particular algorithm.

The two common traversals:

- Breadth-first search
- Depth-first search

Breadth-first search

- Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal.
- A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away.

Applications of Breadth-First Search Algorithm

Breadth-first search can be used to solve many problems such as:

- Finding all connected components in a graph G.

- Finding all nodes within an individual connected component.
- Finding the shortest path between two nodes, u and v, of an unweighted graph.
- Finding the shortest path between two nodes, u and v, of a weighted graph.

Depth First Search

- The depth-first-search algorithm is similar to the standard algorithm for traversing binary trees; it first fully explores one subtree before returning to the current node and then exploring the other subtree.
- Another way to think of depth-first-search is by saying that it is similar to breadth-first search except that it uses a stack instead of a queue.

Applications of Depth-First Search Algorithm

Depth-first search is useful for:

- Finding a path between two specified nodes, u and v, of an unweighted graph.
- Finding a path between two specified nodes, u and v, of a weighted graph.
- Finding whether a graph is connected or not.
- Computing the spanning tree of a connected graph.

Shortest Path algorithm

- The shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.
- This is analogous to the problem of finding the shortest path between two intersections on a road map: the graph's vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of its road segment.
- The Minimal Spanning Tree problem is to select a set of edges so that there is a path between each node. The sum of the edge lengths is to be minimized.
- The Shortest Path Tree problem is to find the set of edges connecting all nodes such that the sum of the edge lengths from the root to each node is minimized.

Applications of Graph

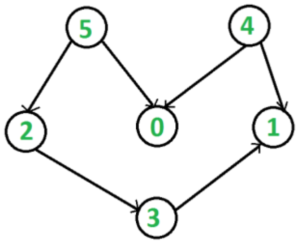
Graphs are constructed for various types of applications such as:

- In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- In maps that draw cities/states/regions as vertices and adjacency relations as edges.
- In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
- Once we have a graph of a particular concept, they can be easily used for finding shortest paths, project planning, etc.
- In flowcharts or control-flow graphs, the statements and conditions in a program are represented as nodes and the flow of control is represented by the edges.

Topological Sorting

Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

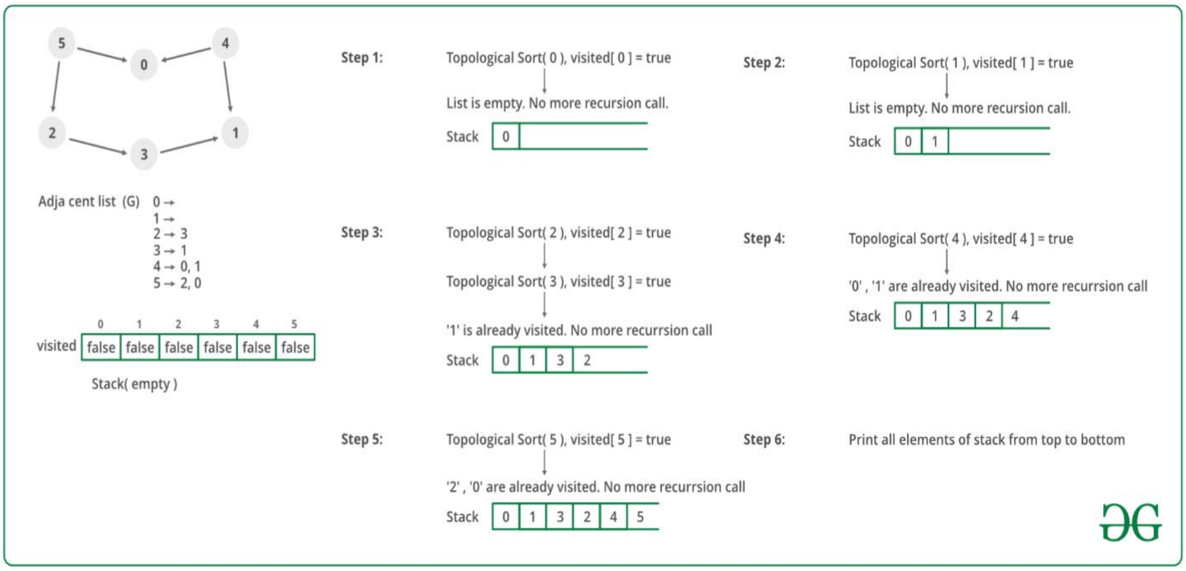
For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).



Algorithm to find Topological Sorting:

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

Below image is an illustration of the above approach:

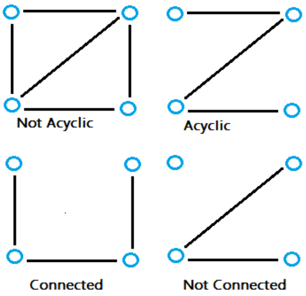

5
0
4
2
3
1
Adja cent list (G) 0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0
0 1 2 3 4 5
visited false false false false false false
Stack(empty)
Step 1:
Topological Sort(0), visited[0] = true
List is empty. No more recursion call.
Stack 0
Step 2:
Topological Sort(1), visited[1] = true
List is empty. No more recursion call.
Stack 0 1
Step 3:
Topological Sort(2), visited[2] = true
Topological Sort(3), visited[3] = true
'1' is already visited. No more recurrsion call
Stack 0 1 3 2
Step 4:
Topological Sort(4), visited[4] = true
'0' , '1' are already visited. No more recurrsion call
Stack 0 1 3 2 4
Step 5:
Topological Sort(5), visited[5] = true
'2' , '0' are already visited. No more recurrsion call
Stack 0 1 3 2 4 5
Step 6:
Print all elements of stack from top to bottom

Minimum Spanning Tree

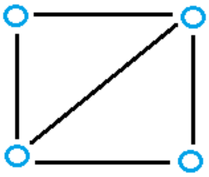
A minimum spanning tree is a special kind of tree that minimizes the lengths (or "weights") of the edges of the tree. An example is a cable company wanting to lay line to multiple neighborhoods; by minimizing the amount of cable laid, the cable company will save money.

A tree has one path joins any two vertices. A spanning tree of a graph is a tree that:

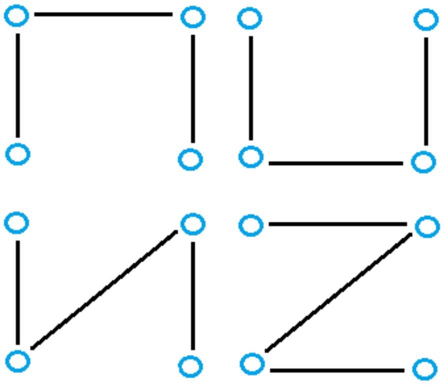
- Contains all the original graph's vertices.
- Reaches out to (spans) all vertices.
- Is acyclic. In other words, the graph doesn't have any nodes which loop back to itself.



Even the simplest of graphs can contain many spanning trees. For example, the following graph:



has many possibilities for spanning trees, including:



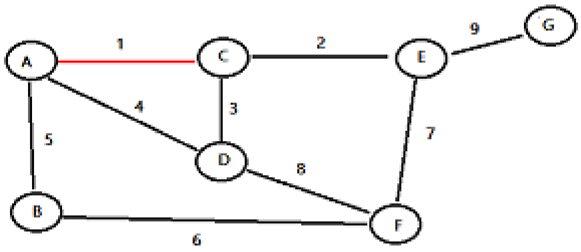
Finding Minimum Spanning Trees

As you can probably imagine, larger graphs have more nodes and many more possibilities for subgraphs. The number of subgraphs can quickly reach into the millions, or billions, making it very difficult (and sometimes impossible) to find the minimum spanning tree. Additionally, the lengths usually have different weights; one 5m long edge might be given a weight of 5, another of the same length might be given a weight of 7.

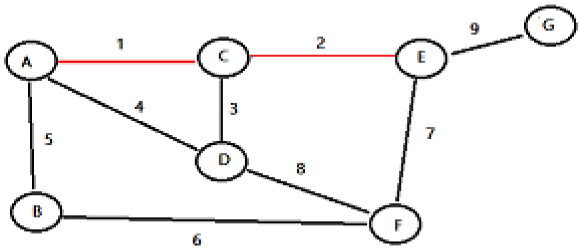
A few popular algorithms for finding this minimum distance include: Kruskal's algorithm, Prim's algorithm and Boruvka's algorithm. These work for simple spanning trees. For more complex graphs, you'll probably need to use software.

Kruskal's algorithm example

Find the edge with the least weight and highlight it. For this example graph, I've highlighted the top edge (from A to C) in red. It has the lowest weight (of 1):



Find the next edge with the lowest weight and highlight it:

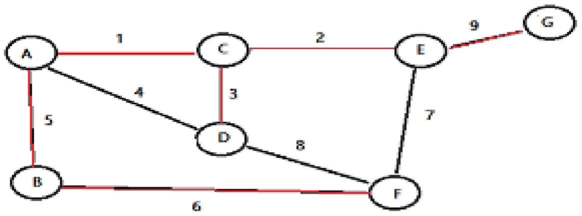


Continue selecting the lowest edges until all nodes are in the same tree.

Notes:

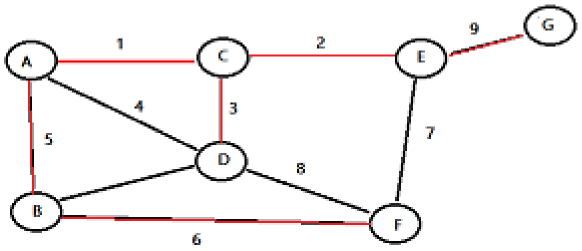
- If you have more than one edge with the same weight, choose an edge with the lowest weight.
- Be careful not to complete a cycle (route one node back to itself). If your choice completes a cycle, discard your choice and move onto the next largest weight.

The finished minimum spanning tree for this example looks like this:



Prim’s Algorithm

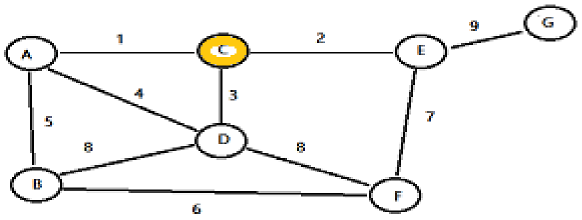
Prim’s algorithm is one way to find a minimum spanning tree (MST).



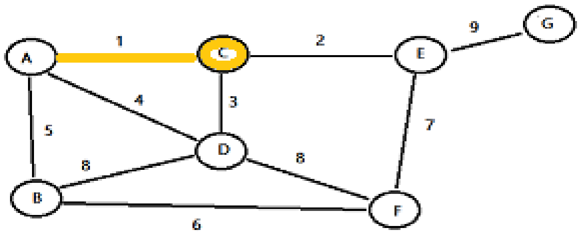
A minimum spanning tree (shown in red) minimizes the edges (weights) of a tree.

How to Run Prim’s Algorithm

Step 1: Choose a random node and highlight it. For this example, I’m choosing node C.

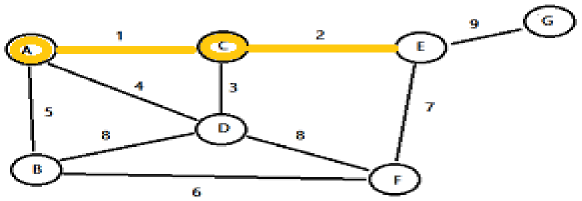


Step 2: Find all of the edges that go to un-highlighted nodes. For this example, node C has three edges with weights 1, 2, and 3. Highlight the edge with the lowest weight. For this example,that’s1.



Step 3: Highlight the node you just reached (in this example, that’s node A).

Step 4: Look at all of the nodes highlighted so far (in this example, that’s A And C). Highlight the edge with lowest weight (in this example, that’s the edge with weight 2).



Note: if you have have more than one edge with the same weight, pick a random one.

Step 5: Highlight the node you just reached.

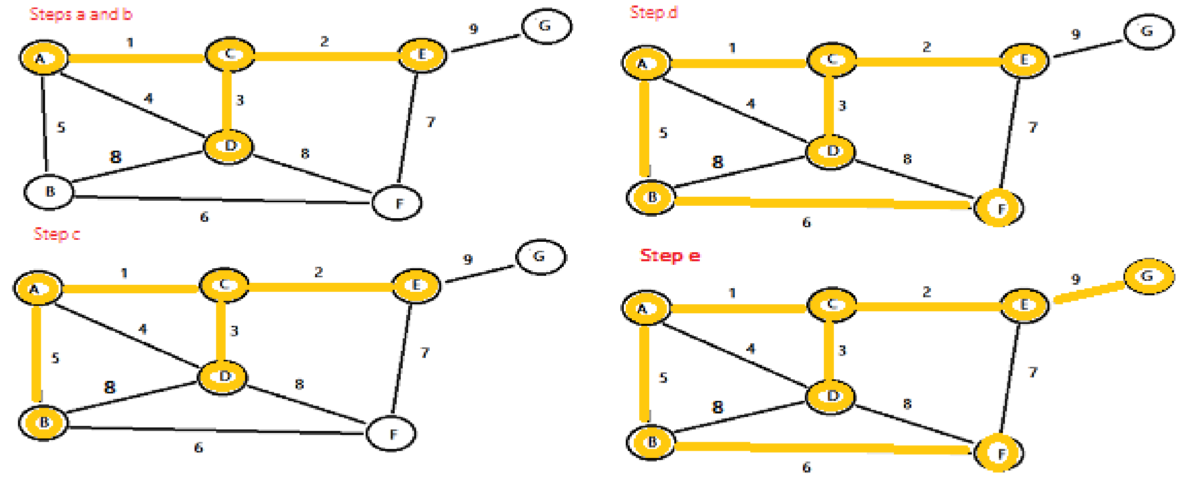
Step 6: Highlight the edge with the lowest weight. Choose from all of the edges that:

- Come from all of the highlighted nodes.
- Reach a node that you haven’t highlighted yet

Step 7: Repeat steps 5 and 6 until you have no more un-highlighted nodes. For this particular example, the specific steps remaining are:

- Highlight node E.
- Highlight edge 3 and then node D.
- Highlight edge 5 and then node B.
- Highlight edge 6 and then node F.
- Highlight edge 9 and then node G.

The finished graph is shown at the bottom right of this image:



Prim's Algorithm Example

Finding shortest paths

Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph *G = (V, E)*, where all the edges are non-negative (i.e., *w(u, v)* ≥ 0 for each edge *(u, v) Є E*).

In the following algorithm, we will use one function ***Extract-Min()***, which extracts the node with the smallest key.

Algorithm: Dijkstra's-Algorithm (G, w, s)

```
for each vertex v Є G.V
  v.d := ∞
  v.∏ := NIL
s.d := 0
S := Φ
Q := G.V
while Q ≠ Φ
  u := Extract-Min (Q)
  S := S U {u}
  for each vertex v Є G.adj[u]
    if v.d > u.d + w(u, v)
      v.d := u.d + w(u, v)
      v.∏ := u
```

Analysis

The complexity of this algorithm is fully dependent on the implementation of Extract-Min function. If extract min function is implemented using linear search, the complexity of this algorithm is $O(V^2 + E)$.

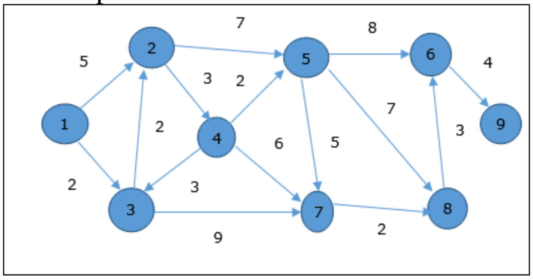
In this algorithm, if we use min-heap on which ***Extract-Min()*** function works to return the node from ***Q*** with the smallest key, the complexity of this algorithm can be reduced further.

Example

Let us consider vertex ***1*** and ***9*** as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by ***0***.

Vertex	Initial	St1 V_1	St2 V_3	St3 V_2	St4 V_4	St5 V_5	St6 V_7	St7 V_8	St8 V_6
1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

Hence, the minimum distance of vertex ***9*** from vertex ***1*** is ***20***. And the path is
1→ 3→ 7→ 8→ 6→ 9
This path is determined based on predecessor information.



Bellman Ford Algorithm

This algorithm solves the single source shortest path problem of a directed graph ***G = (V, E)*** in which the edge weights may be negative. Moreover, this algorithm can be applied to find the shortest path, if there does not exist any negative weighted cycle.

Algorithm: Bellman-Ford-Algorithm (G, w, s)

```
for each vertex v Є G.V
  v.d := ∞
  v.∏ := NIL
s.d := 0
for i = 1 to |G.V| - 1
  for each edge (u, v) Є G.E
    if v.d > u.d + w(u, v)
      v.d := u.d +w(u, v)
      v.∏ := u
for each edge (u, v) Є G.E
  if v.d > u.d + w(u, v)
    return FALSE
return TRUE
```

Analysis

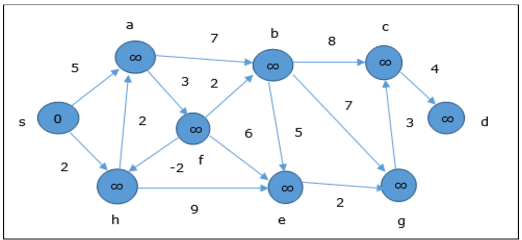
The first **for** loop is used for initialization, which runs in ***O(V)*** times. The next **for** loop runs |***V - 1***| passes over the edges, which takes ***O(E)*** times.

Hence, Bellman-Ford algorithm runs in ***O(V, E)*** time.

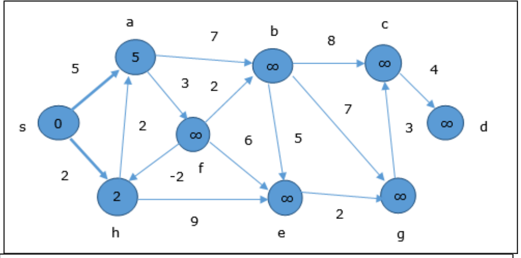
Example

The following example shows how Bellman-Ford algorithm works step by step. This graph has a negative edge but does not have any negative cycle, hence the problem can be solved using this technique.

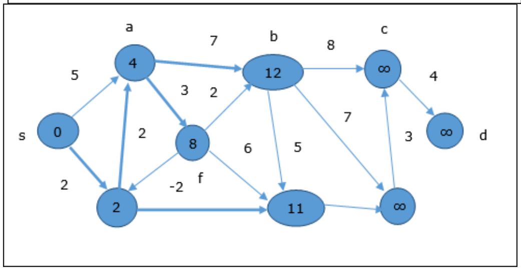
At the time of initialization, all the vertices except the source are marked by ∞ and the source is marked by **0**.



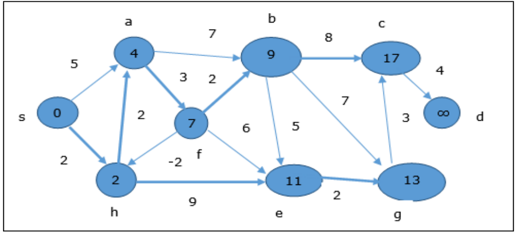
In the first step, all the vertices which are reachable from the source are updated by minimum cost. Hence, vertices ***a*** and ***h*** are updated.



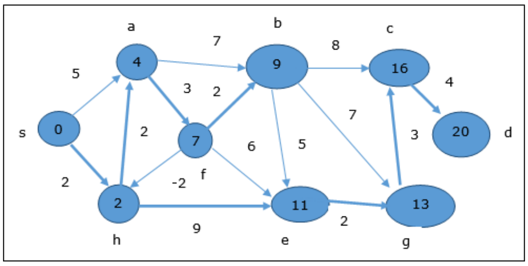
In the next step, vertices ***a, b, f*** and ***e*** are updated.



Following the same logic, in this step vertices ***b, f, c*** and ***g*** are updated.



Here, vertices ***c*** and ***d*** are updated.



Hence, the minimum distance between vertex **s** and vertex **d** is **20.**

Based on the predecessor information, the path is s→ h→ e→ g→ c→ d

Euler Graph-

An Euler graph may be defined as-

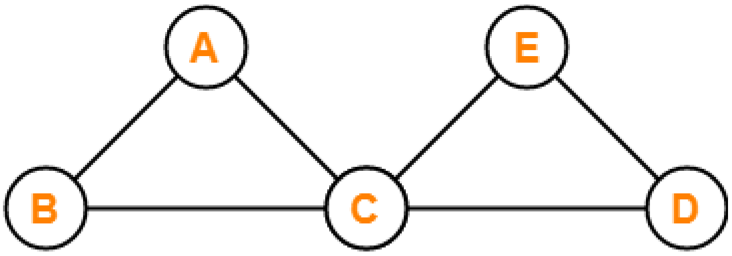
Any connected graph is called as an Euler Graph if and only if all its vertices are of even degree.

OR

An Euler Graph is a connected graph that contains an Euler Circuit.

Euler Graph Example-

The following graph is an example of an Euler graph-



Example of Euler Graph

Here,

- This graph is a connected graph and all its vertices are of even degree.
- Therefore, it is an Euler graph.

Alternatively, the above graph contains an Euler circuit BACEDCB, so it is an Euler graph.

Euler Path-

Euler path is also known as **Euler Trail** or **Euler Walk**.

- If there exists a **Trail** in the connected graph that contains all the edges of the graph, then that trail is called as an Euler trail.

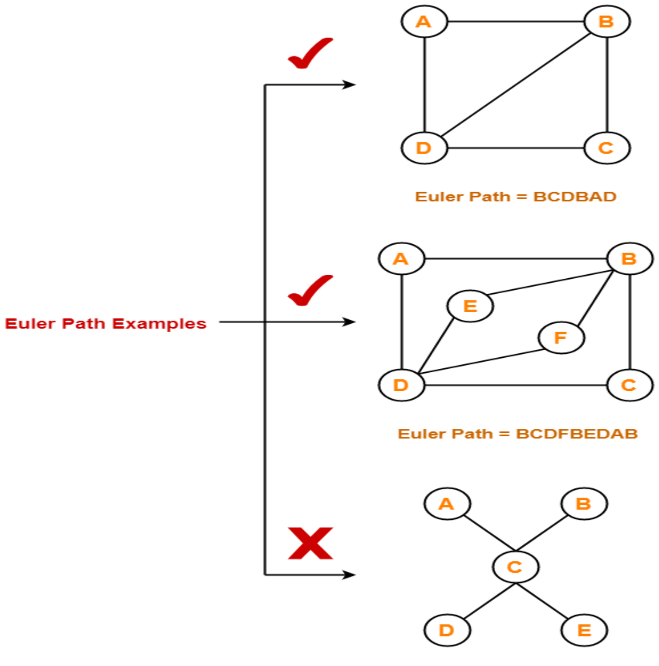
OR

- If there exists a walk in the connected graph that visits every edge of the graph exactly once with or without repeating the vertices, then such a walk is called as an Euler walk.

NOTE: A graph will contain an Euler path if and only if it contains at most two vertices of odd degree.

Euler Path Examples-

Examples of Euler path are as follows-



Euler Path = BCDBAD

Euler Path = BCDFBEDAB

Euler Path Does Not Exist

Euler Circuit-

Euler circuit is also known as **Euler Cycle** or **Euler Tour**.

If there exists a **Circuit** in the connected graph that contains all the edges of the graph, then that circuit is called as an Euler circuit.

OR

If there exists a walk in the connected graph that starts and ends at the same vertex and visits every edge of the graph exactly once with or without repeating the vertices, then such a walk is called as an Euler circuit.

OR

An Euler trail that starts and ends at the same vertex is called as an Euler circuit.

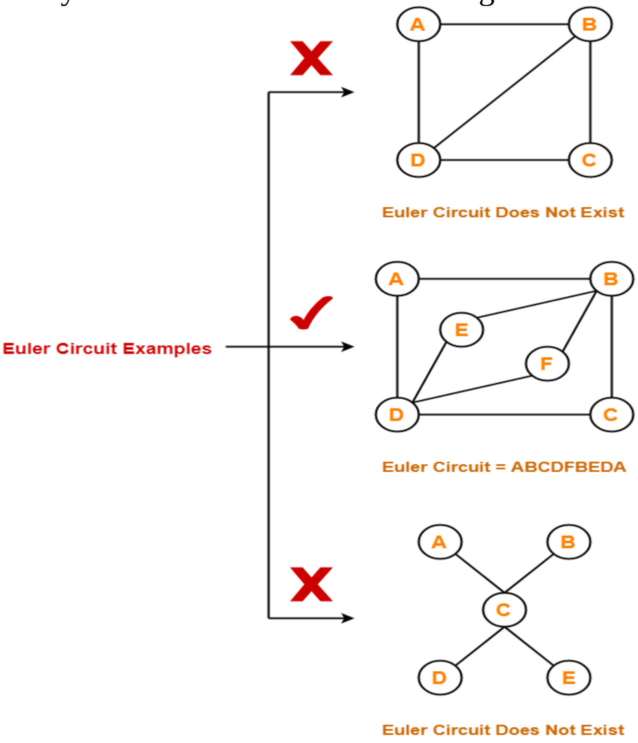
OR

A closed Euler trail is called as an Euler circuit.

NOTE: A graph will contain an Euler circuit if and only if all its vertices are of even degree

Euler Circuit Examples-

Examples of Euler circuit are as follows-


Euler Circuit Examples
A
B
D
C
Euler Circuit Does Not Exist
A
B
E
F
D
C
Euler Circuit = ABCDFBEDA
A
B
C
D
E
Euler Circuit Does Not Exist

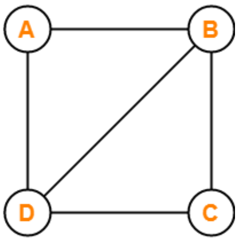
Semi-Euler Graph-

If a connected graph contains an Euler trail but does not contain an Euler circuit, then such a graph is called as a semi-Euler graph.

Thus, for a graph to be a semi-Euler graph, following two conditions must be satisfied-

- Graph must be connected.
- Graph must contain an Euler trail.

Example-



Semi-Euler Graph

Here,

- This graph contains an Euler trail BCDBAD.
- But it does not contain an Euler circuit.
- Therefore, it is a semi-Euler graph.

Important Notes-

Note-01: To check whether any graph is an Euler graph or not, any one of the following two ways may be used-

- If the graph is connected and contains an Euler circuit, then it is an Euler graph.
- If all the vertices of the graph are of even degree, then it is an Euler graph.

Note-02: To check whether any graph contains an Euler circuit or not,

- Just make sure that all its vertices are of even degree.
- If all its vertices are of even degree, then graph contains an Euler circuit otherwise not.

Note-03: To check whether any graph is a semi-Euler graph or not,

- Just make sure that it is connected and contains an Euler trail.

- If the graph is connected and contains an Euler trail, then graph is a semi-Euler graph otherwise not.

Note-04: To check whether any graph contains an Euler trail or not,

- Just make sure that the number of vertices in the graph with odd degree are not more than 2.
- If the number of vertices with odd degree are at most 2, then graph contains an Euler trail otherwise not.

Note-05:

- A graph will definitely contain an Euler trail if it contains an Euler circuit.
- A graph may or may not contain an Euler circuit if it contains an Euler trail.

Note-06:

- An Euler graph is definitely be a semi-Euler graph.
- But a semi-Euler graph may or may not be an Euler graph.

Hamiltonian Graph-

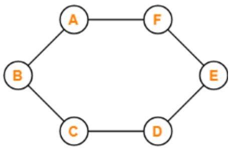
A Hamiltonian graph may be defined as- If there exists a closed walk in the connected graph that visits every vertex of the graph exactly once (except starting vertex) without repeating the edges,then such a graph is called as a Hamiltonian graph.

OR

Any connected graph that contains a Hamiltonian circuit is called as a Hamiltonian Graph.

Hamiltonian Graph Example-

The following graph is an example of a Hamiltonian graph-



Example of Hamiltonian Graph

Here,

- This graph contains a closed walk ABCDEFA.
- It visits every vertex of the graph exactly once except starting vertex.
- The edges are not repeated during the walk.
- Therefore, it is a Hamiltonian graph.

Alternatively, there exists a Hamiltonian circuit ABCDEFA in the above graph, therefore it is a Hamiltonian graph.

Hamiltonian Path-

If there exists a walk in the connected graph that visits every vertex of the graph exactly once without repeating the edges, then such a walk is called as a Hamiltonian path.

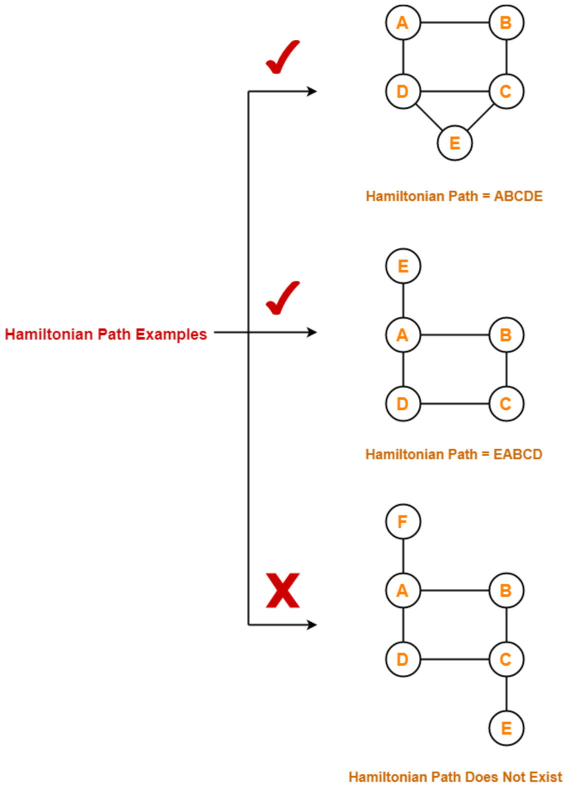
OR

If there exists a **Path** in the connected graph that contains all the vertices of the graph, then such a path is called as a Hamiltonian path.

NOTE: In Hamiltonian path, all the edges may or may not be covered but edges must not repeat.

Hamiltonian Path Examples-

Examples of Hamiltonian path are as follows-



Hamiltonian Circuit-

Hamiltonian circuit is also known as **Hamiltonian Cycle**.

If there exists a walk in the connected graph that visits every vertex of the graph exactly once (except starting vertex) without repeating the edges and returns to the starting vertex, then such a walk is called as a Hamiltonian circuit.

OR

If there exists a **Cycle** in the connected graph that contains all the vertices of the graph, then that cycle is called as a Hamiltonian circuit.

OR

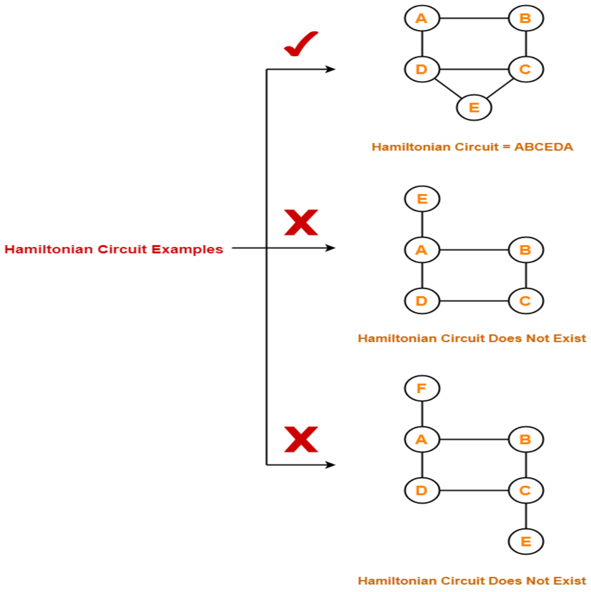
A Hamiltonian path which starts and ends at the same vertex is called as a Hamiltonian circuit.

OR

A closed Hamiltonian path is called as a Hamiltonian circuit.

Hamiltonian Circuit Examples-

Examples of Hamiltonian circuit are as follows-


Hamiltonian Circuit Examples
A
B
D
C
E
Hamiltonian Circuit = ABCEDA
E
A
B
D
C
Hamiltonian Circuit Does Not Exist
F
A
B
D
C
E
Hamiltonian Circuit Does Not Exist

Important Notes-

- Any Hamiltonian circuit can be converted to a Hamiltonian path by removing one of its edges.
- Every graph that contains a Hamiltonian circuit also contains a Hamiltonian path but vice versa is not true.
- There may exist more than one Hamiltonian paths and Hamiltonian circuits in a graph.

<u>QUESTIONS & ANSWERS</u>

1> **Why is the complexity of DFS O(V+E)?**

For a directed graph, the sum of the sizes of the adjacency lists of all the nodes is E. So, the time complexity in this case is O(V) + O(E) = O(V + E).

For an undirected graph, each edge appears twice. Once in the adjacency list of either end of the edge. The time complexity for this case will be O(V) + O (2E) ~ O(V + E).

2> **Why can we not use DFS for finding shortest possible path?**

- In the implementation of DFS, there is no deterministic rule on what order the next children/neighbor to be explored is considered.
- DFS does not guarantee that if node 1 is visited before another node 2 starting from a source vertex. It can not identify what node is closer to the source node.
- DFS just visits the 'deeper' nodes in any order. It can even be farther from source nodes. In the worst case, it might go as far as possible from the source node and then returns to unvisited adjacent nodes of visited nodes.
- Due to this, DFS is not a reliable choice to find the shortest path between the nodes.

3> **Is DFS a complete algorithm?**

- A search algorithm is said to be complete if at least one solution exists then the algorithm is guaranteed to find a solution in a finite amount of time.
- DFS is complete if the search tree is finite, meaning for a given finite search tree, DFS will come up with a solution if it exists.

4> **Is DFS a optimal algorithm?**

- A search algorithm is optimal if it finds a solution, it finds that in the best possible manner.
- DFS is not optimal, meaning the number of steps in reaching the solution, or the cost spent in reaching it is high.

5> **When is it best to use DFS?**

- The usage of DFS heavily depends on the structure of the search tree/graph and the number and location of solutions needed.
  - If it is known that the solution is not far from the root of the tree, a breadth first search (BFS) might be better.
  - If the tree is very deep and solutions are rare, depth first search (DFS) might take an extremely long time, but BFS could be faster.
  - If the tree is very wide, a BFS might need too much memory, so it might be completely impractical. We go for DFS in such cases.
  - If solutions are frequent but located deep in the tree we opt for DFS.

6> **Why do we prefer queues instead of other data structures while implementing BFS?**

- BFS searches for nodes levelwise, i.e. it searches the nodes w.r.t their distance from the root (or source).
- From the above example, we could see that BFS required us to visit the child nodes in order their parents were discovered.
- Whenever we visit a node, we insert all the neighboring nodes into our data structure. If a queue data structure is used, it guarantees that, we get the nodes in order their parents were discovered as queue follows the FIFO (first in first out) flow.

7> **Why is time complexity more in the case of graph being represented as Adjacency Matrix?**

- Every time we want to find what are the edges adjacent to a given node 'U', we have to traverse the whole array AdjacencyMatrix[U], which is of length |V|.
- During BFS, you take a starting node S, which is at level 0. All the adjacent nodes are at level 1. Then, we mark all the adjacent nodes of all vertices at level 1, which don't have a level, to level 2. So, every vertex will belong to one level only and when an element is in a level, we have to check once for its adjacent nodes which

takes O(V) time. Since the level covers V elements over the course of BFS, the total time would beO(V * V) which is O(V2).

- In short, for the case of Adjacency Matrix, to tell which nodes are adjacent to a given vertex, we take O(V) time, irrespective of edges.
- Whereas, when Adjacency List is used, it is immediately available to us and it just takes time complexity proportional to adjacent nodes itself, which upon summation over all nodes V is E. So, BFS when using Adjacency List gives O(V + E).

8> What is 0-1 BFS?

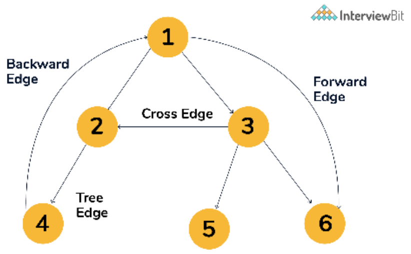
- This type of BFS is used to find shortest distance or path from a source node to a destination node in a graph with edge values 0 or 1.
- When the weights of edges are 0 or 1, the normal BFS techniques provide erroneous results because in normal BFS technique, its assumed that the weight of edges would be equal in the graph.
- In this technique, we will check for the optimal distance condition instead of using bool array to mark visited nodes.
- We use double ended queue to store the node details. While performing BFS, if we encounter a edge having weight = 0, the node is pushed at front of double ended queue and if a edge having weight = 1 is found, the node is pushed at back of double ended queue.
- The above approach is similar to Dijkstra's algorithm where if the shortest distance to node is relaxed by the previous node then only it will be pushed in the queue.

9> Why can't we use normal queue in 0-1 BFS technique?

- The normal queue lacks methods which helps us to perform the below functions necessary for performing 0-1 BFS:
    - Removing Top Element (To get vertex for BFS)
    - Inserting at the beginning of queue
    - Inserting at the end
- All the above operations are supported in Double ended Queue data structure and hence we go for that.

10> What are the classifications of edges in a BFS graph?

For the given graph below, the general types of edges are as follows:



- **Tree Edge:** The edge which is present in the tree obtained after applying DFS on the graph. All the Green edges are tree edges.
- **Forward Edge:** Edge (u, v) such that v is descendant but not part of the DFS tree. Edge from node 1 to node 6 is a forward edge.
- **Back edge**:
    - Edge (u, v) such that v is ancestor of edge u but not part of DFS or BFS tree. Edge from node 4 to node 1 is a back edge.
    - Presence of back edge indicates a cycle in the directed graph.
- **Cross Edge**: It is a edge which connects two nodes such that they do not have any ancestor and a descendant relationship between them. Edge from node 3 to node 2 is a cross edge.

11> What are the types of edges present in BFS of a directed graph?

- A BFS of a directed graph has only Tree Edge, Cross Edge and Back Edge.
- The BFS has NO Forward edges.

- For Edge A->B as forward edge, node B should have been visited before the edge A-B is discovered and this can happen only when B is visited via some other node using more than one edge.
- As BFS finds shortest path from source by using optimal number of edges, when node A is enqueued, edge A-B will have been discovered and would be marked as a tree or cross edge. Hence, forward edges is never possible in BFS.

12> Can BFS be used for finding shortest possible path?

Yes. BFS is mostly used for finding shortest possible path.

13> What is the difference between DFS and BFS? When is DFS and BFS used?

- BFS is less space efficient than DFS as BFS maintains a priority queue of the entire level while DFS just maintains a few pointers at each level by using simple stack.
- If it is known priorly that an answer will likely be found far into a tree (depths of tree), DFS is a better option than BFS.
- BFS is useful when the depth of the tree can vary or when a single answer is needed. For instance, the shortest path in a maze. BFS will perform better here because DFS is most likely to start out on a wrong path, exploring a large portion of the maze before reaching the goal.
- If it is known that the solution is not far from the root of the tree, a breadth first search (BFS) might be better.
- If the tree is very deep and solutions are rare, depth first search (DFS) might take an extremely long time, but BFS could be faster.
- If the tree is very wide, a BFS might need too much memory, so it might be completely impractical. We go for DFS in such cases.
- If solutions are frequent but located deep in the tree we opt for DFS.

14> Is BFS a complete algorithm?

- A search algorithm is said to be complete if at least one solution exists then the algorithm is guaranteed to find a solution in a finite amount of time.
- BFS is complete.

15> Is BFS a optimal algorithm?

- A search algorithm is optimal if it finds a solution, it finds that in the best possible manner.
- BFS is optimal which is why it is being used in cases to find single answer in optimal manner.

16> What is minimum spanning tree problem?

The **minimum** labeling **spanning tree problem** is to find a **spanning tree** with least types of labels if each edge in a graph is associated with a label from a finite label set instead of a weight. A bottleneck edge is the highest weighted edge in a **spanning tree**

17> How many edges does a minimum spanning tree have?

- four edges
- As a **minimum spanning tree** is also a **spanning tree**, these properties will also be true for a **minimum spanning tree**. **vertices**, and each of the **spanning trees** contains four **edges**.

18> Is a minimum spanning tree unique?

Any undirected, connected graph has a **spanning tree**. If the graph has more than one connected component, each component will have a **spanning tree** (and the union of these **trees** will form a **spanning** forest for the graph). The **spanning tree** of G is not **unique**. ... This is called the **minimum spanning tree** (**MST**) of G.

19> Which is better Prims or Kruskal?

Prim's algorithm has a time complexity of O(V^2), V being the number of vertices and can be improved up to O(E + log V) using Fibonacci heaps. ... **Prim's** algorithm runs faster in dense graphs. **Kruskal's** algorithm runs faster in sparse graphs

20> Why do we use minimum spanning tree?

Minimum spanning trees are used for network designs (i.e. telephone or cable networks). They **are** also **used** to find approximate solutions for complex mathematical problems like the Traveling Salesman Problem. Other, diverse applications include: Cluster Analysis.

21> What is the cost of its minimum spanning tree?

The **cost** of a **spanning tree** is the sum of **costs** on **its** edges. An MST of G is a **spanning tree** of G having a **minimum cost**.

22> What is the difference between spanning tree and minimum spanning tree?

If the graph is edge-weighted, we can define the weight of a **spanning tree** as the sum of the weights of all its edges. A **minimum spanning tree** is a **spanning tree** whose weight is the smallest among all possible **spanning trees**.

23> What is maximum spanning tree?

A **maximum spanning tree** is a **spanning tree** of a weighted graph having **maximum** weight. It can be computed by negating the weights for each edge and applying Kruskal's algorithm (Pemmaraju and Skiena, 2003, p. 336). A **maximum spanning tree** can be found in the Wolfram Language using the command FindSpanningTree[g]

24> What is Spanning Tree with example?

Given a graph G=(V,E), a subgraph of G that is connects all of the vertices and is a **tree** is called a **spanning tree** . For **example**, suppose we start with this graph: We can remove edges until we are left with a **tree**: the result is a **spanning tree**. Clearly, a **spanning tree** will have |V|-1 edges, like any other **tree**

25> How do you calculate spanning tree?

If a graph is a complete graph with n vertices, then total number of **spanning trees** is $n^{(n-2)}$ where n is the number of nodes in the graph. In complete graph, the task is equal to counting different labeled **trees** with n nodes for which have Cayley's **formula**.

26> How do you solve Prim's algorithm?

Algorithm

Step 1: Select a starting vertex.
Step 2: Repeat Steps 3 and 4 until there are fringe vertices.
Step 3: Select an edge e connecting the tree vertex and fringe vertex that has minimum weight.
Step 4: Add the selected edge and the vertex to the minimum spanning tree T. [END OF LOOP]
Step 5: EXIT.

27> What is the time complexity of Kruskal's algorithm?

Kruskal's algorithm involves sorting of the edges, which takes O(E logE) **time**, where E is a number of edges in graph and V is the number of vertices. After sorting, all edges are iterated and union-find **algorithm** is applied.

28> What is minimally connected graph?

A **graph** is said to be **minimally connected** if removal of any one edge from it disconnects the **graph**. Clearly, a **minimally connected graph** has no cycles.

29> Is every minimum spanning tree of G also a skinny spanning tree?

We define the jumbo edge of T to be the edge of T with the largest weight. A **spanning tree** To of **G** is a **skinny spanning tree** if there is no **spanning tree of G** whose jumbo edge has a smaller weight. Give a counterexample showing that not **every skinny spanning tree of G** must be a **minimum spanning tree** of the same graph.

30> Can a graph have two minimum spanning tree?

For a connected undirected **graph** G = (V,E), a **spanning tree** is a **tree** T = (V,E) with E ⊆ E. Note that a **graph can have** many **spanning trees**, but all **have** |V | vertices and |V | − 1 edges. Example 18.2. A **graph** on the left, and **two** possible **spanning trees**.

31> How do you find multiple minimum spanning trees?

You can do this by following method:

Find the edges in MST having same weight as some other edge not in MST. ...

Remove edge (a , b) from the graph and run MST again.
Repeat until you **find** no other such edge or MST with equal overall weight.

32> is Kruskal greedy?

Kruskal's Algorithm: An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a **Greedy** Algorithm.

33> What is Dijkstra shortest path algorithm?

Dijkstra's algorithm. **Dijkstra's algorithm** to find the **shortest path** between a and b. It picks the unvisited vertex with the lowest **distance**, calculates the **distance** through it to each unvisited neighbor, and updates the neighbor's **distance** if smaller. Mark visited (set to red) when done with neighbors.

34> Does Dijkstra give minimum spanning tree?

Strictly, the answer is no. **Dijkstra's** algorithm finds the shortest path between 2 vertices on a graph. However, a very small change to the algorithm produces another algorithm which **does** efficiently produce an MST.

35> What is single source shortest path algorithm?

The **single source shortest path algorithm** (for arbitrary weight positive or negative) is also known Bellman-Ford **algorithm** is used to find minimum distance from **source** vertex to any other vertex. ... At first it finds those distances which have only one edge in the **path**

36> What is the other name of Dijkstra algorithm?

Dijkstra's algorithm (or **Dijkstra's** Shortest Path First **algorithm**, SPF **algorithm**) is an **algorithm** for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.

MCQ’S

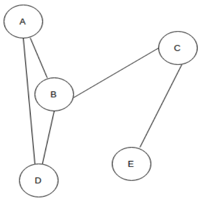
1> **Which of the following statements for a simple graph is correct?**

a) Every path is a trail
b) Every trail is a path
c) Every trail is a path as well as every path is a trail
d) Path and trail have no relation

Answer: a

Explanation: In a walk if the vertices are distinct it is called a path, whereas if the edges are distinct it is called a trail.

2> **In the given graph identify the cut vertices.**

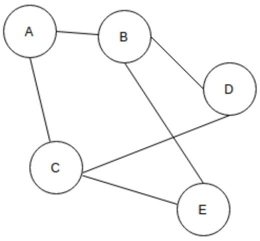


a) B and E
b) C and D
c) A and E
d) C and B

Answer: d

Explanation: After removing either B or C, the graph becomes disconnected.

3> **For the given graph(G), which of the following statements is true?**



a) G is a complete graph
b) G is not a connected graph
c) The vertex connectivity of the graph is 2
d) The edge connectivity of the graph is

Answer: c

Explanation: After removing vertices B and C, the graph becomes disconnected

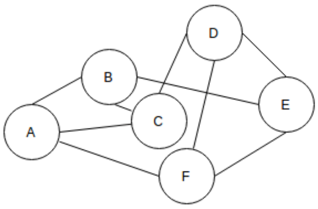
4> **What is the number of edges present in a complete graph having n vertices?**

a)(n*(n+1))/2
b)(n*(n-1))/2
c)n
d)Information given is insufficient

Answer:b

Explanation: Number of ways in which every vertex can be connected to each other is nC2.

5. The given Graph is regular.



a) True
b) Fals

Answer:A

Explanation: In a regular graph, degrees of all the vertices are equal. In the given graph the degree of every vertex is 3.

6. In a simple graph, the number of edges is equal to twice the sum of the degrees of the vertices.

a) True

b) False

Answer: b

Explanation: The sum of the degrees of the vertices is equal to twice the number of edges.

7. A connected planar graph having 6 vertices, 7 edges contains _____________ regions.

a) 1 b) 3 c) 1 d) 11

Answer: b

Explanation: By euler's formula the relation between vertices(n), edges(q) and regions(r) is given by n-q+r=2.

8. If a simple graph G, contains n vertices and m edges, the number of edges in the Graph G'(Complement of G) is ___________

a) (n*n-n-2*m)/2 b) (n*n+n+2*m)/2

c) (n*n-n-2*m)/2 d) (n*n-n+2*m)/2

Answer: a

Explanation: The union of G and G' would be a complete graph so, the number of edges in G'= number of edges in the complete form of G(nC2)-edges in G(m).

9. Which of the following properties does a simple graph not hold?

a) Must be connected b) Must be unweighted

c) Must have no loops or multiple edges d) Must have no multiple edges

Answer: a

Explanation: A simple graph maybe connected or disconnected.

10. What is the maximum number of edges in a bipartite graph having 10 vertices?

a) 24 b) 21 c) 25 d) 16

Answer: c

Explanation: Let one set have n vertices another set would contain 10-n vertices.

Total number of edges would be n*(10-n), differentiating with respect to n, would yield the answer.

11. Which of the following is true?

a) A graph may contain no edges and many vertices

b) A graph may contain many edges and no vertices

c) A graph may contain no edges and no vertices

d) A graph may contain no vertices and many edges

Answer: b

Explanation: A graph must contain at least one vertex.

12. For a given graph G having v vertices and e edges which is connected and has no cycles, which of the following statements is true?

a) v=e b) v = e+1 c) v + 1 = e d) v = e-1

Answer: b

Explanation: For any connected graph with no cycles the equation holds true.

13. For which of the following combinations of the degrees of vertices would the connected graph be eulerian?

a) 1,2,3 b) 2,3,4 c) 2,4,5 d) 1,3,5

Answer: a

Explanation: A graph is eulerian if either all of its vertices are even or if only two of its vertices are odd.

14. A graph with all vertices having equal degree is known as a __________

a) Multi Graph b) Regular Graph c) Simple Graph d) Complete Graph

Answer: b

Explanation: The given statement is the definition of regular graphs.

15. Which of the following ways can be used to represent a graph?
a) Adjacency List and Adjacency Matrix
b) Incidence Matrix
c) Adjacency List, Adjacency Matrix as well as Incidence Matrix
d) No way to represent

Answer: c

Explanation: Adjacency Matrix, Adjacency List and Incidence Matrix are used to represent a graph.

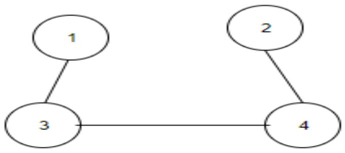
16. The number of elements in the adjacency matrix of a graph having 7 vertices is __________

a) 7 b) 14 c) 36 d) 49

Answer: d

Explanation: There are n*n elements in the adjacency matrix of a graph with n vertices.

17. What would be the number of zeros in the adjacency matrix of the given graph?



a) 10 b) 6
c) 16 d) 0

Answer: b

Explanation: Total number of values in the matrix is 4*4=16, out of which 6 entries are non zero.

18. Adjacency matrix of all graphs are symmetric.
a) False b) True

Answer: a

Explanation: Only undirected graphs produce symmetric adjacency matrices.

19. The time complexity to calculate the number of edges in a graph whose information in stored in form of an adjacency matrix is ____________
a) O(V) b) O(E^2) c) O(E) d) O(V^2)

Answer: d

Explanation: As V entries are 0, a total of V^2-V entries are to be examined.

20. For the adjacency matrix of a directed graph the row sum is the _________ degree and the column sum is the ________ degree.
a) in, out b) out, in c) in, total d) total, out

Answer: b

Explanation: Row number of the matrix represents the tail, while Column number represents the head of the edge.

<u>SET 2</u>

1. What is the maximum number of possible non zero values in an adjacency matrix of a simple graph with n vertices?
a) (n*(n-1))/2 b) (n*(n+1))/2 c) n*(n-1) d) n*(n+1)

Answer: c

Explanation: Out of n*n possible values for a simple graph the diagonal values will always be zero.

2. On which of the following statements does the time complexity of checking if an edge exists between two particular vertices is not, depends?
a) Depends on the number of edges
b) Depends on the number of vertices

c) Is independent of both the number of edges and vertices
d) It depends on both the number of edges and vertices

Answer: c

Explanation: To check if there is an edge between to vertices i and j, it is enough to see if the value of A[i][j] is 1 or 0, here A is the adjacency matrix.

3. In the given connected graph G, what is the value of rad(G) and diam(G)?

a) 2, 3 b) 3, 2 c) 2, 2 d) 3, 3

Answer: a

Explanation: Value of eccentricity for vertices A, C is 2 whereas for F, B, D, E it is 3.

4. Which of these adjacency matrices represents a simple graph?

a) [[1, 0, 0], [0, 1, 0], [0, 1, 1]] b) [[1, 1, 1], [1, 1, 1], [1, 1, 1]]
c) [[0, 0, 1], [0, 0, 0], [0, 0, 1]] d) [[0, 0, 1], [1, 0, 1], [1, 0, 0]]

Answer: d

Explanation: A simple graph must have no-self loops, should be undirected.

5. Given an adjacency matrix A = [[0, 1, 1], [1, 0, 1], [1, 1, 0]], The total no. of ways in which every vertex can walk to itself using 2 edges is ________

a) 2 b) 4 c) 6 d) 8

Answer: c

Explanation: A^2 = [[2, 1, 1], [1, 2, 1], [1, 1, 2]], all the 3 vertices can reach to themselves in 2 ways, hence a total of 3*2, 6 ways.

6. If A[x+3][y+5] represents an adjacency matrix, which of these could be the value of x and y.

a) x=5, y=3 b) x=3, y=5 c) x=3, y=3 d) x=5, y=5

Answer: a

Explanation: All adjacency matrices are square matrices.

7. Two directed graphs(G and H) are isomorphic if and only if A=PBP-1, where P and A are adjacency matrices of G and H respectively.

a) True b) False

Answer: a

Explanation: This is a property of isomorphic graphs.

8. Space complexity for an adjacency list of an undirected graph having large values of V (vertices) and E (edges) is ___________

a) O(E) b) O(V*V) c) O(E+V) d) O(V)

Answer: c

Explanation: In an adjacency list for every vertex there is a linked list which have the values of the edges to which it is connected.

9. For some sparse graph an adjacency list is more space efficient against an adjacency matrix.

a) True b) False

Answer: a

Explanation: Space complexity for adjacency matrix is always O(V*V) while space complexity for adjacency list in this case would be O(V).

10. Time complexity to find if there is an edge between 2 particular vertices is

a) O(V) b) O(E) c) O(1) d) O(V+E)

Answer: a

Explanation: The maximum edges a vertex can have is V-1.

11. For the given conditions, which of the following is in the correct order of increasing space requirement?

i) Undirected, no weight ii) Directed, no weight
iii) Directed, weighted iv) Undirected, weighted

a) ii iii i iv b) i iii ii iv c) iv iii i ii d) i ii iii iv

Answer: a

Explanation: i) takes v+4e, ii) takes v+2e, iii) takes v+3e, iv) takes v +6e space.

12. Space complexity for an adjacency list of an undirected graph having large values of V (vertices) and E (edges) is __________

**a) O(V)		b) O(E*E)		c) O(E)		d) O(E+V)**

Answer: c

Explanation: In an adjacency list for every vertex there is a linked list which have the values of the edges to which it is connected.

13. Complete the given snippet of code for the adjacency list representation of a weighted directed graph.

```
class neighbor
    {
                int vertex, weight;
                ____ next;
        }

        class vertex
    {
                string name;
                                _____ adjlist;
                }

                vertex adjlists[101];
```

a) vertex, vertex

b) neighbor, vertex

c) neighbor, neighbor

d) vertex, neighbor

Answer: c

Explanation: Vertex would have a name and a linked list attached to it.

14. In which case adjacency list is preferred in front of an adjacency matrix?

**a) Dense graph				b) Sparse graph**

**c) Adjacency list is always preferred	d) Complete graph**

Answer: b

Explanation: In case of sparse graph most of the entries in the adjacency matrix would be 0, hence adjacency list would be preferred.

15. To create an adjacency list C++’s map container can be used.

**a) True			b) False**

Answer: a

Explanation: We can create a mapping from string to a vector, where string would be the name of the vertex and vector would contains the name of the vertices to which it is connected.

16. What would be the time complexity of the following function which adds an edge between two vertices i and j, with some weight ‘weigh’ to the graph having V vertices?

```
vector<int> adjacent[15] ;
vector<int> weight[15];

void addEdge(int i,int j,int weigh)
{
        adjacent[a].push_back(i);
        adjacent[b].push_back(j);
        weight[a].push_back(weigh);
        weight[b].push_back(weigh);
}
```

a) O(1)

b) O(V)

c) O(V*V)

d) O(log V)

Answer: a

Explanation: The function win in the constant time as all the four step takes constant time.

17. What would be the time complexity of the BFS traversal of a graph with n vertices and $n^{1.25}$ edges?

**a) O(n)		b) O($n^{1.25}$)		c) O($n^{2.25}$)		d) O(n*n)**

Answer: b

Explanation: The time complexity for BFS is O(|V| + |E|) = O(n + $n^{1.25}$) = O($n^{1.25}$).

18. The number of possible undirected graphs which may have self loops but no multiple edges and have n vertices is ________

a) $2^{((n*(n-1))/2)}$ b) $2^{((n*(n+1))/2)}$ c) $2^{((n-1)*(n-1))/2)}$ d) $2^{((n*n)/2)}$

Answer: d

Explanation: There can be at most, n*n edges in an undirected graph.

19. Given a plane graph, G having 2 connected component, having 6 vertices, 7 edges and 4 regions. What will be the number of connected components?

a) 1 b) 2 c) 3 d) 4

Answer: b

Explanation: Euler's Identity says V – E + R = 1+ number of connected components.

20. Number of vertices with odd degrees in a graph having a eulerian walk is

a) 0 b) Can't be predicted c) 2 d) either 0 or 2

Answer: d

Explanation: If the start and end vertices for the path are same the answer would be 0 otherwise 2.

<u>SET 3</u>

1. Dijkstra's Algorithm will work for both negative and positive weights?

a) True b) False

Answer: b

Explanation: Dijkstra's Algorithm assumes all weights to be non-negative.

2. A graph having an edge from each vertex to every other vertex is called a

a) Tightly Connected b) Strongly Connected

c) Weakly Connected d) Loosely Connected

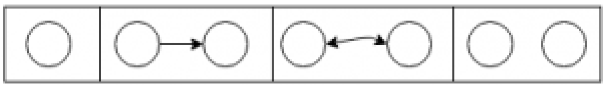
Answer: a

Explanation: This is a part of the nomenclature followed in Graph Theory.

3. What is the number of unlabeled simple directed graph that can be made with 1 or 2 vertices?

a) 2 b) 4 c) 5 d) 9

Answer: b

Explanation:

4. Floyd Warshall Algorithm used to solve the shortest path problem has a time complexity of __________

a) O(V*V) b) O(V*V*V) c) O(E*V) d) O(E*E)

Answer: b

Explanation: The Algorithm uses Dynamic Programming and checks for every possible path.

5. All Graphs have unique representation on paper.

a) True b) False

Answer: b

Explanation: Same Graph may be drawn in different ways on paper.

6. Assuming value of every weight to be greater than 10, in which of the following cases the shortest path of a directed weighted graph from 2 vertices u and v will never change?

a) add all values by 10 b) subtract 10 from all the values

c) multiply all values by 10 d) in both the cases of multiplying and adding by 10

Answer: c

Explanation: In case of addition or subtraction the shortest path may change because the number of edges between different paths may be different, while in case of multiplication path wont change.

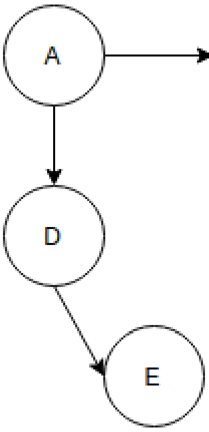
7. What is the maximum possible number of edges in a directed graph with no self loops having 8 vertices?

a) 28 b) 64 c) 256 d) 56

Answer: d

Explanation: If a graph has V vertices than every vertex can be connected to a possible of V-1 vertices.

8. What would be the DFS traversal of the given Graph?



a) ABCED
b) AEDCB
c) EDCBA
d) ADECB

Answer: a

Explanation: In this case two answers are possible including ADEBC.

9. What would be the value of the distance matrix, after the execution of the given code?

```
#include <bits/stdc++.h>
#define INF 1000000
int graph[V][V] = {   {0,   7,  INF, 4},
                      {INF, 0,   13, INF},
                      {INF, INF, 0,   12},
                      {INF, INF, INF, 0}
                  };

int distance[V][V], i, j, k;

for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
                distance[i][j] = graph[i][j];

for (k = 0; k < V; k++)
                for (i = 0; i < V; i++)
                for (j = 0; j < V; j++)
                 {
                                if (distance[i][k] + distance[k][j] < distance[i][j])
                                                distance[i][j] = distance[i][k] + distance[k][j];

                            return 0;
                 }
```

```
a){ {0,   7,  INF, 4},
    {INF, 0,   13, INF},
    {INF, INF, 0,   12},
    {INF, INF, INF, 0}};
```

```
b){ {0,   7,  20, 24},
    {INF, 0,   13, 25},
    {INF, INF, 0,   12},
    {INF, INF, INF, 0}};
```

```
c){0,   INF,  20, 24},
  {INF, INF,   13, 25},
    {INF, INF, 0,   12},
    {INF, INF, INF, 0}
    {INF, 0,   13, 25},
    {INF, INF, 0,   12},
    {24, INF, INF, 0}};
```

d) None of the mentioned

Answer: b

Explanation: The program computes the shortest sub distances.

10. What is the maximum number of edges present in a simple directed graph with 7 vertices if there exists no cycles in the graph?

a) 21 b) 7 c) 6 d) 49

Answer: c

Explanation: If the no cycles exists then the difference between the number of vertices and edges is 1.

11. How many of the following statements are correct?

i) All cyclic graphs are complete graphs. ii) All complete graphs are cyclic graphs.

iii) All paths are bipartite. iv) All cyclic graphs are bipartite.

v) There are cyclic graphs which are complete.

a) 1 b) 2 c) 3 d) 4

Answer: b

Explanation: Statements iii) and v) are correct.

12. All paths and cyclic graphs are bipartite graphs.

a) True b) False

Answer: b

Explanation: Only paths and even cycles are bipartite graphs.

13. What is the number of vertices of degree 2 in a path graph having n Vertices here n>2.

a) n-2 b) n c) 2 d) 0

Answer: a

Explanation: Only the first and the last vertex would have degree 1, others would be of degree 2.

14. All trees with n vertices consists of n-1 edges.

a) True b) False

Answer: a

Explanation: A trees is acyclic in nature.

15. Which of the following graphs are isomorphic to each other?

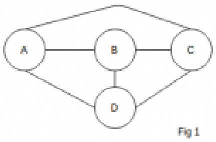


Fig 1

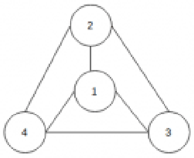


Fig 2

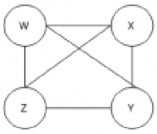


Fig 3

a) fig 1 and fig 2

b) fig 2 and fig 3

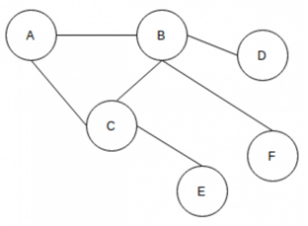
c) fig 1 and fig 3

d) fig 1, fig 2 and fig 3

Answer: d

Explanation: All three graphs are Complete graphs with 4 vertices.

16. In the given graph which edge should be removed to make it a Bipartite Graph?



a) A-C
b) B-E
c) C-D
d) D-E

Answer: a
Explanation: The resultant graph would be a Bipartite Graph having {A,C,E} and {D, B} as its subgroups.
17. What would the time complexity to check if an undirected graph with V vertices and E edges is Bipartite or not given its adjacency matrix?
a) O(E*E) b) O(V*V) c) O(E) d) O(V)
Answer: b
Explanation: A graph can be checked for being Bipartite by seeing if it is 2-colorable or not, which can be obtained with the help of BFS.
18. A graph is a set of points, called?
A. Nodes B. Edge C. fields D. lines
Answer: A
Explanation: A graph is a set of points, called nodes or vertices, which are interconnected by a set of lines called edges
19. Graph consists of a?
A. non-empty set of vertices B. empty set of vertices
C. Both A and B D. None of the above
ans : A
Explanation: Graph consists of a non-empty set of vertices or nodes V and a set of edges E.
20. Number of edges incident with the vertex V is called?
A. Degree of a Graph B. Handshaking Lemma
C. Degree of a Vertex D. None of the above
ans : C
Explanation: Degree of a Vertex − The degree of a vertex V of a graph G (denoted by deg (V)) is the number of edges incident with the vertex V.

SET 4

1> Let GG be the simple graph with 20 vertices and 100 edges. The size of the minimum vertex cover of GG is 8. Then, the size of the maximum independent set of GG is:
A.12 B.8 C. <8 D.>12
2> How many perfect matching are there in a complete graph of 66 vertices?

A. 1515
B. 2424
C. 3030
D. 60

3> Consider an undirected random graph of eight vertices. The probability that there is an edge between a pair of vertices is 1/2. What is the expected number of unordered cycles of length three?
A.1/8 B.1 **C.7** D.8
4> Which of the following statements is/are TRUE for undirected graphs?

P: no of odd degree vertices is even

Q: Sum of degrees of all vertices

A.P Only B, Q Only **C. Both P & Q** C. None

5> The line graph L(G) of a simple graph G is defined as follows: · There is exactly one vertex v(e) in L(G) for each edge e in G. · For any two edges e and e' in G, L(G) has an edge between v(e) and v(e'), if and only if e and e'are incident with the same vertex in G. Which of the following statements is/are TRUE?

(P) The line graph of a cycle is a cycle.
(Q) The line graph of a clique is a clique.
(R) The line graph of a planar graph is planar.
(S) The line graph of a tree is a tree.

a. P Only B. P and R Only C. R Only D. P,Q and S Only

6> Let G be a simple undirected planar graph on 10 vertices with 15 edges. If G is a connected graph, then the number of bounded faces in any embedding of G on the plane is equal to

A. 3 B.4 C.5 **D.6**

7. Let G be a complete undirected graph on 6 vertices. If vertices of G are labeled, then the number of distinct cycles of length 4 in G is equal to

A.15 B.30 **C. 45** D. 360

8. Let G = (V,E) be a graph. Define ξ(G) = Σd id x d, where id is the number of vertices of degree d in G. If S and T are two different trees with ξ(S) = ξ(T),then

A. |S| = 2|T| B. |S| = |T|-1 **C. |S| = |T|** D. |S| = |T|+1

9. The degree sequence of a simple graph is the sequence of the degrees of the nodes in the graph in decreasing order. Which of the following sequences can not be the degree sequence of any graph?

(I) 7, 6, 5, 4, 4, 3, 2, 1
(II) 6, 6, 6, 6, 3, 3, 2, 2
(III) 7, 6, 6, 4, 4, 3, 2, 2
(IV) 8, 7, 7, 6, 4, 2, 1, 1

A.I and II B.III and IV C. IV only **D. II and IV**

10. What is the chromatic number of an n-vertex simple connected graph which does not contain any odd length cycle? Assume n >= 2.

A. 2 B. 3 C. n-1 D. n

11. Which one of the following is TRUE for any simple connected undirected graph with more than 2 vertices?

a. No two vertices have the same degree. **B. At least two vertices have the same degree**
c. At least three vertices have the same degree D. all vertices have the same degree

12. Which of the following statements is true for every planar graph on n vertices?

A.The graph is connected
B.The graph is Eulerian
C.The graph has a vertex-cover of size at most 3n/4
D. The graph has an independent set of size at least n/3

13. G is a graph on n vertices and 2n - 2 edges. The edges of G can be partitioned into two edge-disjoint spanning trees. Which of the following is NOT true for G?

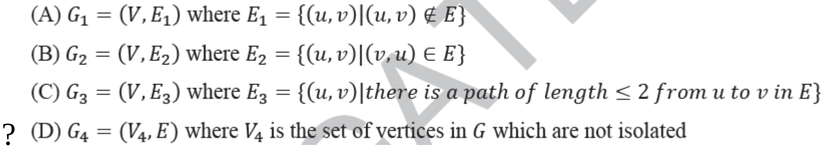
A. For every subset of k vertices, the induced subgraph has at most 2k-2 edges
B. The minimum cut in G has at least two edges
C. There are two edge-disjoint paths between every pair to vertices
D. There are two vertex-disjoint paths between every pair of vertices

14. Let G be the non-planar graph with the minimum possible number of edges. Then G has

A. 9 edges and 5 vertices **B. 9 edges and 6 vertices**
C. 10 edges and 5 vertices D. 10 edges and 6 vertices

15. Which of the following graphs has an Eulerian circuit?
A. Any k-regular graph where kis an even number. B. A complete graph on 90 vertices
C. The complement of a cycle on 25 vertices D. None of the above

16. Let G=(V,E) be a directed graph where V is the set of vertices and E the set of edges. Then which one of the following graphs has the same strongly connected components as G



(A) $G_1 = (V, E_1)$ where $E_1 = \{(u,v)|(u,v) \notin E\}$

(B) $G_2 = (V, E_2)$ where $E_2 = \{(u,v)|(v,u) \in E\}$

(C) $G_3 = (V, E_3)$ where $E_3 = \{(u,v)|there\ is\ a\ path\ of\ length \leq 2\ from\ u\ to\ v\ in\ E\}$

? (D) $G_4 = (V_4, E)$ where V_4 is the set of vertices in G which are not isolated

A.A **B.B** C.C D.D

17. Consider an undirected graph G where self-loops are not allowed. The vertex set of G is {(i, j): 1 <= i <= 12, 1 <= j <= 12}. There is an edge between (a, b) and (c, d) if |a − c| <= 1 and |b − d| <= 1. The number of edges in this graph is __________.
A.500 B.502 **C.506** D.510

18. An ordered n-tuple (d1, d2, … , dn) with d1 >= d2 >= ⋯ >= dn is called graphic if there exists a simple undirected graph with n vertices having degrees d1, d2, … , dn respectively. Which of the following 6-tuples is NOT graphic?
A. (1, 1, 1, 1, 1, 1) B. (2, 2, 2, 2, 2, 2)
C. (3, 3, 3, 1, 0, 0) D. (3, 2, 1, 1, 1, 0)

19. The maximum number of edges in a bipartite graph on 12 vertices is __________________________.
A.36 B.12 C.48 D.24

20. A cycle on n vertices is isomorphic to its complement. The value of n is _____.
A.2 B.4 C.6 **D.5**

SET 5

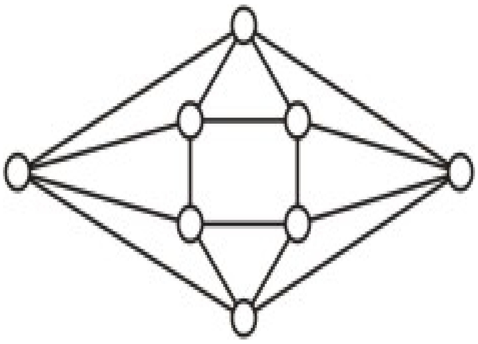
1. If G is a forest with n vertices and k connected components, how many edges does G have?
A. floor(n/k) B. ceil(n/k) **C.n-k** d.n-k-1

2. The 2^n vertices of a graph G corresponds to all subsets of a set of size n, for n >= 6 . Two vertices of G are adjacent if and only if the corresponding sets intersect in exactly two elements. The number of vertices of degree zero in G is:
a.1 b.n **c. n+1** d. 2n

3. The 2^n vertices of a graph G corresponds to all subsets of a set of size n, for n >= 6. Two vertices of G are adjacent if and only if the corresponding sets intersect in exactly two elements. The number of connected components in G is:
a.n **b.n+2** c. 2n/2 d. 2n / n

4.Let s and t be two vertices in a undirected graph G + (V, E) having distinct positive edge weights. Let [X, Y] be a partition of V such that s ∈ X and t ∈ Y. Consider the edge e having the minimum weight amongst all those edges that have one vertex in X and one vertex in Y. Let the weight of an edge e denote the congestion on that edge. The congestion on a path is defined to be the maximum of the congestions on the edges of the path. We wish to find the path from s to t having minimum congestion. Which one of the following paths is always such a path of minimum congestion?
A. a path from s to t in the minimum weighted spanning tree
b. a weighted shortest path from s to t
c.a n Euler walk from s to t
d. a Hamiltonian path from s to t

5. The minimum number of colours required to colour the following graph, such that no two adjacent vertices are assigned the same colour, is



a.2 b.3 **c.4** d.5

6. Let G be an arbitrary graph with n nodes and k components. If a vertex is removed from G, the number of components in the resultant graph must necessarily lie between

a. k and n b. k-1 and k+1 **c. k-1 and n-1** d. k+1 and n-k

7. How many perfect matchings are there in a complete graph of 6 vertices ?

a.15 b.24 c.30 d.60

8. A graph G = (V, E) satisfies |E| ≤ 3 |V| - 6. The min-degree of G is defined as $\min_{v \in V} \{\text{degree}\ (v)\}$. Therefore, min-degree of G cannot be

a.3 b.4 c.5 **d.6**

9. The minimum number of colours required to colour the vertices of a cycle with η nodes in such a way that no two adjacent nodes have the same colour is

a.2 b.3 c.4 d**. n - 2⌊n/2⌋ + 2**

10. Maximum number of edges in a n - node undirected graph without self loops is

a. n2 **b. n(n - 1)/2** c.n-1 d. (n + 1) (n)/2

11. Let G be a connected planar graph with 10 vertices. If the number of edges on each face is three, then the number of edges in G is _______________.

a.24 b.20 c.32 d.64

12. A graph is self-complementary if it is isomorphic to its complement. For all self-complementary graphs on n vertices, n is

a. A multiple of 4 b.even c.odd **d. Congruent to 0 mod 4, or 1 mod 4**

13. In a connected graph, a bridge is an edge whose removal disconnects a graph. Which one of the following statements is True?

a. A tree has no bridge

b. A tree has no bridge

c. Every edge of a clique with size ≥ 3 is a bridge (A clique is any complete subgraph of a graph)

d. A graph with bridges cannot have a cycle

14. What is the number of vertices in an undirected connected graph with 27 edges, 6 vertices of degree 2, 3 vertices of degree *4* and remaining of degree 3?

a.10 b.11 c.18 **d.19**

15. If all the edge weights of an undirected graph are positive, then any subset of edges that connects all the vertices and has minimum total weight is a

a. Hamiltonian cycle b.grid c.hypercube **d.tree**

16. Consider a weighted undirected graph with positive edge weights and let uv be an edge in the graph. It is known that the shortest path from the source vertex s to u has weight 53 and the shortest path from s to v has weight 65. Which one of the following statements is always true?

A. weight (u, v) < 12 b. weight (u, v) ≤ 12

c. weight (u, v) > 12 **d. weight (u, v) ≥ 12**

17. G is a simple undirected graph. Some vertices of G are of odd degree. Add a node v to G and make it adjacent to each odd degree vertex of G. The resultant graph is sure to be

a.regular b.complete c.Hamiltonian **d.euler**

18. Consider the following statements

(I) Let T be a binary search tree with 4 height. The minimum and maximum possible nodes of T are 5 and 15 respectively.

(II) In a binary tree, the number of internal nodes of degree 2 is 6, and the number of internal nodes of degree 1 is 8. The number of leaf nodes in the binary tree is 15.

Which of the following statement(s) is/are correct?

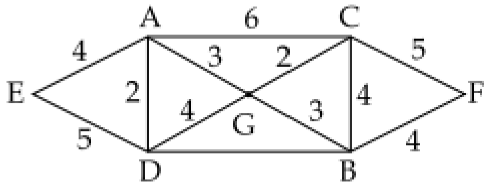
a. only I b. only II c. both I and II **d. neither I nor II**

19. Let G = (V, E) be any connected undirected edge-weighted graph. The weights of the edges in E are positive. Consider the following statements:

1. The path between a pair of vertices in a minimum spanning tree of an undirected graph is necessarily the shortest (minimum weight) path.
2. Minimum Spanning Tree of G is always unique and shortest path between a pair of vertices may not be unique.

a. only 1 b.only 2 c. both 1 and 2 **d. neither 1 nor 2**

20. Consider the graph given below : Use Kruskal's algorithm to find a minimal spanning tree for the graph. The List of the edges of the tree in the order in which they are choosen is? (1) AD, AE, AG, GC, GB, BF (2) GC, GB, BF, GA, AD, AE (3) GC, AD, GB, GA, BF, AE (4) AD, AG, GC, AE, GB, BF



A.1 b.1,2 c.1,2,3 **d.1,2,3,4**

CHAPTER -7 SEARCHING

Introduction to Searching in Data Structure

Searching in data structure refers to the process of finding location LOC of an element in a list. This algorithm can be executed on both internal as well as external data structures. The efficiency of searching an element increases the efficiency of any algorithm.

Searching Techniques in Data Structure

1. Sequential Search

This is the traditional technique for searching an element in a collection of elements. In this type of search, all the elements of the list are traversed one by one to find if the element is present in the list or not. One example of such an algorithm is a linear search. This is a straightforward and basic algorithm. Suppose ARR is an array of n elements, and we need to find location LOC of element ITEM in ARR. For this, LOC is assigned to -1, which indicates that ITEM is not present in ARR. While comparing ITEM with data at each ARR location, and once ITEM == ARR[N], LOC is updated with location N+1. Hence we found the ITEM in ARR.

Algorithm:

LSEARCH(ARR, N, ITEM, LOC) Here ARR Is the array of N number of elements, ITEM

holds the value we need to search in the array and algorithm returns LOC, the location where

ITEM is present in the ARR. Initially, we have to set LOC = -1.

```
1. Set LOC = -1,i=1
2. Repeat while DATA[i] != ITEM:
    i=i+1
3. If i=N+1 ,then Set LOC =0
   Else LOC = N+1
4. Exit.
```

Complexity of algorithm

Space complexity

As linear search algorithm does not use any extra space, thus its space complexity = O(n) for an array of n number of elements.

Time Complexity

- **Worst-case complexity:** O(n) – This case occurs when the search element is not present in the array.
- **Best case complexity:** O(1) – This case occurs when the first element is the element to be searched.
- **Average complexity:** O(n) – This means when an element is present somewhere in the middle of the array.

2. Binary Search

This is a technique to search an element in the list using the divide and conquer technique. This type of technique is used in the case of sorted lists. Instead of searching an element one by one in the list, it directly goes to the middle element of the list, divides the array into 2 parts, and decides element lies in which sub-array the element exists.

Suppose ARR is an array with sorted n number of elements present in increasing order. With every step of this algorithm, the searching is confined within BEG and END, which are the beginning and ending index of sub-arrays. The index MID defines the middle index of the array where,

```
MID = INT(beg + end )/2
```

It needs to be checked if ITEM < ARR[N} where ITEM is the element that we need to search in ARR.

- If ITEM = ARR[MID] then LOC = MID and exit .
- If ITEM < ARR[MID} then ITEM can appear in the left sub-array, then BEG will be the same and END = MID -1 and repeat.
- If ITEM > ARR[MID] then ITEM can appear in the right subarray then BEG = MID+1 and END will be the same and repeat.

After this MID is again calculated for respective sub-arrays, if we didn’t find the ITEM, the algorithm returns -1 otherwise LOC = MID.

Algorithm:

BSEARCH(ARR, LB, UB, ITEM, LOC) Here, ARR is a sorted list of elements, with LB and UB are lower and upper bounds for the array. ITEM needs to be searched in the array and algorithm returns location LOC, index at which ITEM is present else return -1.

```
1. Set BEG = LB, END = UB and MID = INT([BEG+END]/2)
2. Repeat step 3 and 4 while BEG <= END and ARR[MID] != ITEM
3. IF ITEM< ARR[MID] then:
    Set END = MID-1
   Else:
    Set BEG = MID+1
4. Set MID = INT(BEG+END)/2
5. IF ARR[MID] = ITEM then:
    Set LOC = MID
   Else:
```

Set LOC = NULL

6. Exit.

The complexity of Binary Search

Here are the complexities of the binary search given below.

- **Worst Case:** O(nlogn)
- **Best Case:** O(1)
- **Average Case:** O(nlogn)

Jump Search

- Like Binary Search, Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.
  For example, suppose we have an array arr[] of size n and block (to be jumped) size m. Then we search at the indexes arr[0], arr[m], arr[2m]…..arr[km] and so on. Once we find the interval (arr[km] < x < arr[(k+1)m]), we perform a linear search operation from the index km to find the element x.
  Let's consider the following array: (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610). Length of the array is 16. Jump search will find the value of 55 with the following steps assuming that the block size to be jumped is 4.
  STEP 1: Jump from index 0 to index 4;
  STEP 2: Jump from index 4 to index 8;
  STEP 3: Jump from index 8 to index 12;
  STEP 4: Since the element at index 12 is greater than 55 we will jump back a step to come to index 8.
  STEP 5: Perform linear search from index 8 to get the element 55.

  Important points:
- Works only sorted arrays.
- The optimal size of a block to be jumped is (√ n). This makes the time complexity of Jump Search O(√ n).
- The time complexity of Jump Search is between Linear Search ((O(n)) and Binary Search (O (Log n)).
- Binary Search is better than Jump Search, but Jump search has an advantage that we traverse back only once (Binary Search may require up to O(Log n) jumps, consider a situation where the element to be searched is the smallest element or smaller than the smallest). So in a system where binary search is costly, we use Jump Search.

Interpolation Search

Given a sorted array of n uniformly distributed values arr[], write a function to search for a particular element x in the array.
The Interpolation Search is an improvement over Binary Search for instances, where the values in a sorted array are uniformly distributed. Binary Search always goes to the middle element to check. On the other hand, interpolation search may go to different locations

according to the value of the key being searched. For example, if the value of the key is closer to the last element, interpolation search is likely to start search toward the end side.

To find the position to be searched, it uses following formula.

```
// The idea of formula is to return higher value of pos
// when element to be searched is closer to arr[hi]. And
// smaller value when closer to arr[lo]
pos = lo + [ (x-arr[lo])*(hi-lo) / (arr[hi]-arr[Lo]) ]

arr[] ==> Array where elements need to be searched
x     ==> Element to be searched
lo    ==> Starting index in arr[]
hi    ==> Ending index in arr[]
```

Algorithm

Rest of the Interpolation algorithm is the same except the above partition logic.
Step1: In a loop, calculate the value of "pos" using the probe position formula.
Step2: If it is a match, return the index of the item, and exit.
Step3: If the item is less than arr[pos], calculate the probe position of the left sub-array. Otherwise calculate the same in the right sub-array.
Step4: Repeat until a match is found or the sub-array reduces to zero.

Runtime complexity of interpolation search algorithm is **O(log (log n))** as compared to **O(log n)** of BST in favorable situations.

Indexed Sequential Search

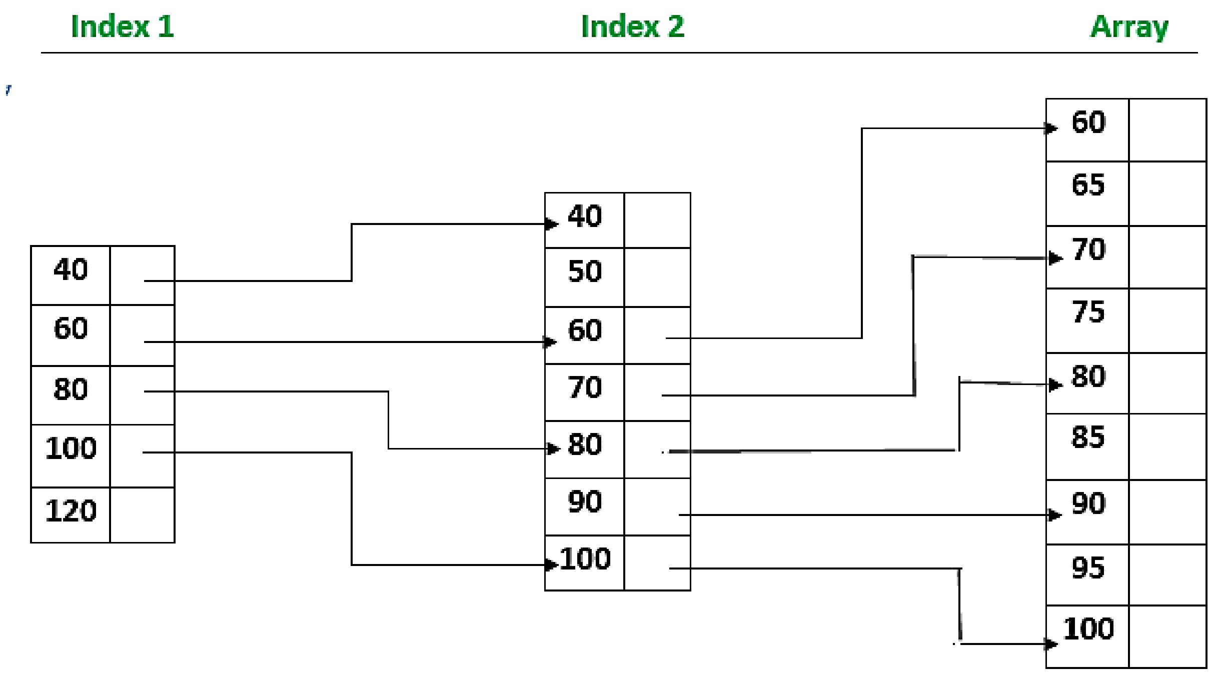
In this searching method, first of all, an index file is created, that contains some specific group or division of required record when the index is obtained, then the partial indexing takes less time cause it is located in a specified group.
Note: When the user makes a request for specific records it will find that index group first where that specific record is recorded.

Characteristics of Indexed Sequential Search:

- In Indexed Sequential Search a sorted index is set aside in addition to the array.
- Each element in the index points to a block of elements in the array or another expanded index.
- The index is searched 1st then the array and guides the search in the array.
- The advantage of the indexed sequential method is that items in the table can be examined sequentially if all records in the file have to be accessed, yet the search time for particular items is reduced. A sequential search is performed on the smaller index rather than on the large table. Once the index position is found, the search is made on the record table itself.

Note: Indexed Sequential Search actually does the indexing multiple time, like creating the index of an index.

Explanation by diagram "Indexed Sequential Search":



TOPIC-SEARCHING

ESSENTIAL QUIZ

1 .Finding the location of a given element from the collection of item is called as

A. Searching
B. Sorting
C. Mining
D. Discovering

OUTPUT: A

2. Search in which record is checked and read for desired items in file linearly is classified as

A. Combinational search
B. Quadratic search
C. Linear research
D. Linear search

OUTPUT: D

3. Which is the following searching algorithms works on the divide and conquer?

A. Binary search
B. Linear search
C. Sequential Search
D. All of the above

OUTPUT: A

4. What is the total number of the comparison done in the binary search?

A. n
B. n^2
C. n+1
D. log(N+1)

OUTPUT: D

5. In the following type of searching key-comparisons are needed

A. Linear search
B. Non-linear search
C. Address calculation search
D. A and B

Answer: D

6. In the following type of searching key-comparisons are not needed

A. Linear search
B. Non-linear search
C. Address calculation search
D. A and B

Answer: C

7. Binary search is ____ type of search.

A. Linear search
B. Non linear search
C. Address calculation search
D. A and B

Answer: A

8. Splay tree search is ___ type of search

A. Linear search
B. Non linear search
C. Address calculation search
D. None

Answer: B

9. B-tree search is a ___ type of search

A. Just tree search
B. multi-way tree search
C. graph search
D. None

Answer: B

10. DFS is a ____ type of search

A. Just tree search
B. multi-way tree search
C. graph search
D. None

Answer: C

11. BFS is a ____ type of search

A. Just tree search
B. multi-way tree search
C. graph search
D. None

12. Searching techniques are classified in to__ types

A. 2
B. 3

C. 4
D. none

Answer: A

13. The element that is going to be searched in a list is called ___

A. Key
B. item
C. table
D. file

Answer: A

14. If a key is found in a list that is called ___ type of search

A. Unsuccessful
B. successful
C. partial success
D. partial unsuccessful

Answer: B

15.The following type of search is easy to implement.

A. Linear search
B. Non linear search
C. Interpolation
D. none

Answer : A

16.In linear search with array, how many comparisons are needed in best case?

A. 0
B. 1
C. n
D. n/2

Answer: B

17. In linear search with array, how many comparisons are needed in average case?

A. 0
B. 1
C. n
D. n+1/2

Answer: D

18. In linear search with array, how many comparisons are needed in worst case?

A. 0
B. 1
C. n
D. n/2

Answer: C

19. In ___type of search the list is divided in to two parts.

A. Linear search
B. Binary search
C. random search
D. None

Answer: B

20. Asymptotic complexity of linear search with array in average case is

A. O(1)
B. O(n)
C. O(n/2)
D. logn

Answer: B

TOPIC-SEARCHING
ESSENTIAL QUIZ

1. Asymptotic complexity of linear search with array in worst case is

A. O(1)
B. O(n)
C. O(n/2
D. logn

Answer: B

2. Binary search algorithm cannot be applied to__

A. Sorted Linked list
B. sorted binary trees
C. sorted linear array
D. pointer array

Answer: D

3.Which of the following is not a limitation of binary search algorithm?

A. must use a sorted array.
B. requirement of sorted array is expensive when a lot of insertion and deletions are needed.
C. there must be a mechanism to access middle element directly.
D. binary search algorithm is not efficient when the data elements more than 1500.

Answer: D

4. The complexity of Binary search algorithm is

A. O (n)
B. O (log n)
C. O (n2)
D. O (logn2)

Answer: B

5.The time factor when determining the efficiency of algorithm is measured by__.

A. Counting the number of key operations
B. Counting the microseconds
C. Counting the number of statements
D. Counting the kilobytes of algorithm

Answer: A

6.Binary search can be applied on the sorted __________.

A. array or list.
B. Arguments
C. Queues
D. pointers.

Answer: A

7. Binary search is useful when there are large number of_______in array.

A. Arguments
B. values
C. Elements
D. all the above

Answer: C

8 .In binary search, we compare the value with the elements in the __________ position of the array.

A. right
B. Left
C. Random
D. middle

Answer: D

9. Which of the following is not the required condition for binary search algorithm?

A. A.The list must be sorted
B. There should be the direct access to the middle element in any sub list
C. There must be mechanism to delete and/or insert elements in list.
D. Number values should only be present

Answer: C

10. Finding the location of the element with a given value is ___.

A. Traversal
B. Search
C. Sort
D. None of above

Answer: B

11. In linear search time complexity in best case for a successful search is___.

A. 1
B. n
C. n+1/2
D. n -1

Answer: A

12. In linear search time complexity in average case for a successful search is___.

A. 1
B. n
C. n+1/2
D. n -1

Answer: C

13. In linear search time complexity in average case for a Unsuccessful search is__.

A. 1
B. n
C. n+1/2
D. n -1

Answer: B

14. In linear search with list time complexity in average case for a Unsuccessful search is

A. A.1
B. n
C. n+1/2
D. n -1

Answer: C

15. In linear search with list time complexity in worst case for a Unsuccessful search is

A. 1
B. n
C. n+1/2
D. n -1

Answer: B

16. In Binary search time complexity in best case for a successful search is __

A. 1
B. Log2N
C. log n+1
D. N log N

Answer: A

17. In Binary search time complexity in worst case for a successful search is __

A. 1
B. Log2N
C. log n+1
D. N log N

Answer: B

18. In Binary search time complexity in average case for a successful search is __

A. 1
B. Log2N
C. log n+1
D. N log N

Answer: B

19. In Binary search time complexity in average case for a Unsuccessful search is __

A. 1
B. Log2N
C. log n+1
D. N log N

Answer: B

20. In Binary search time complexity in a best case for a Unsuccessful search is __

A. 1
B. Log2N
C. log n+1
D. N log N

Answer: B

TOPIC-SEARCHING
ESSENTIAL QUIZ

1.In Binary search time complexity in worst case for a Unsuccessful search is __

A. 1
B. Log2N
C. log n+1
D. N log N

Answer: B

2. The number of comparisons for the Binary Tree Search in best case is __

A. 1
B. n+1
C. n-1
D. n+1/2

Answer: A

3.The number of comparisons for the Binary Tree Search in average case is __

A. 1
B. n+1
C. n
D. n+1/2

Answer: C

4.The number of comparisons for the Binary Tree Search in worst case is __

A. 1
B. n+1
C. n
D. n+1/2

Answer: D

5.The worst case occur in linear search algorithm when item is __

A. Somewhere in the middle of the array
B. Not in the array at all.
C. The last element in the array
D. The last element in the array or is not there at all.

Answer: D

6. What are the applications of binary search?

A. To find the lower/upper bound in an ordered sequence
B. Union of intervals
C. Debugging
D. All of the mentioned

ANSWER: D

7. How can Jump Search be improved?

A. Start searching from the end
B. Begin from the kth item, where k is the step size
C. Cannot be improved
D. Step size should be other than sqrt(n)

ANSWER: B

8. What is the advantage of recursive approach than an iterative approach?

a) Consumes less memory

b) Less code and easy to implement
c) Consumes more memory
d) More code has to be written
Answer: b
Explanation: A recursive approach is easier to understand and contains fewer lines of code.
9. What is the length of the step in jump search?
a) n
b) n/2
c) sqrt(n)
Answer: c
Explanation: If the step size is 1, it becomes a linear search, if it is n, we reach the end of the list in just on step, if it is n/2, it becomes similar to binary search, therefore the most efficient step size is found to be sqrt(n).
10. Interpolation search performs better than binary search when?
a) array has uniformly distributed values but is not sorted
b) array is sorted and has uniform distribution of values
c) array is sorted but the values are not uniformly distributed
d) array is not sorted
Answer: b
Explanation: Interpolation search is an improvement over a binary search for the case when array is sorted and has uniformly distributed values. Binary search performs better when the values are not distributed uniformly.
11. What is the time complexity of interpolation search when the input array has uniformly distributed values and is sorted?
a) O(n)
b) O(log log n)
c) O(n log n)
d) O(log n)
Answer: b
Explanation: Interpolation search goes to different positions in the array depending on the value being searched. It is an improvement over binary search and has a time complexity of O(log log n).
12. Which of the following searching algorithm is fastest when the input array is sorted and has uniformly distributed values?
a) jump search
b) exponential search
c) binary search
d) interpolation search
Answer: d
Explanation: Interpolation search has a time complexity of O(log log n) when the array is sorted and has uniformly distributed values. It has the least time complexity out of the given options for such a case.
13. Which of the following searching algorithm is fastest when the input array is sorted but has non uniformly distributed values?

a) jump search
b) linear search
c) binary search
d) interpolation search
Answer: c
Explanation: Interpolation search has a time complexity of O(n) when the array does not have uniformly distributed values. So in such a case binary search has the least time complexity out of the given options.
14. Which of the following searching algorithm is fastest when the input array is not sorted but has uniformly distributed values?
a) jump search
b) linear search
c) binary search
d) interpolation search
Answer: b
Explanation: Out of the given options linear search is the only searching algorithm which can be applied to arrays which are not sorted. It has a time complexity of O(n) in the worst case.
15. The searching technique that takes O (1) time to find a data is

A. Hashing
B. Linear search
C. Binary search
D. Tree search
Answer: Hashing

16. A characteristic of the data that binary search uses but the linear search ignores is the___________.

A. Order of the elements of the list.
B. Length of the list.
C. Maximum value in list.
D. Type of elements of the list.

Answer: (a).Order of the elements of the list.
17. In binary search, we compare the value with the elements in the __________ position of the array.

A. right
B. Left
C. Random
D. Middle
Answer: D

18. Which of the following algorithm does not divide the list −

a. linear search
b. binary search
c. merge sort
d. quick sort
Answer: A

19. The worst case complexity of binary search matches with −

a. interpolation search

b. linear search
c. merge sort
d. none of the above

Answer: B

20. State True or False.

i) Binary search is used for searching in a sorted array.

ii) The time complexity of binary search is O(logn).
a. True, False
b. False, True
c. False, False
d. True, True

Answer: D

ADVANCE LEVEL QUIZ
TOPIC: SEARCHING

1. What is the output of following program?

```
#include <stdio.h>
 void print(int n, int j)
{
   if (j >= n)
      return;
   if (n-j > 0 && n-j >= j)
       printf("%d %dn", j, n-j);
   print(n, j+1);
}
 int main()
{
    int n = 8;
    print(n, 1);
}
```

```
A)   1 7                               B)   17
     26                                     26
     35                                     35
     44                                     44
     44
C)    17                                        D)  12
     26                                             34
     35                                   56
                                            78
```

OUTPUT: B
Explanation: For a given number n, the program prints all distinct pairs of positive integers with sum equal to n.

2. Which of the following is correct recurrence for worst case of Binary Search?

A) T(n) = 2T(n/2) + O(1) and T(1) = T(0) = O(1)
B) T(n) = T(n-1) + O(1) and T(1) = T(0) = O(1)
C) T(n) = T(n/2) + O(1) and T(1) = T(0) = O(1)
D)T(n) = T(n-2) + O(1) and T(1) = T(0) = O(1)
OUTPUT: C

3. Given a sorted array of integers, what can be the minimum worst case time complexity to find ceiling of a number x in given array? Ceiling of an element x is the smallest element present in array which is greater than or equal to x. Ceiling is not present if x is greater than the maximum element present in array. For example, if the given array is {12, 67, 90, 100, 300, 399} and x = 95, then output should be 100.
A) O(LogLogn) B) O(n)

C) O(Logn) D) O(Logn * Logn)
OUTPUT: C

4. Consider the C function given below. Assume that the array listA contains n (> 0) elements, sorted in ascending order.

```
int ProcessArray(int *listA, int x, int n)
{
    int i, j, k;
    i = 0;
    j = n-1;
    do
    {
        k = (i+j)/2;
        if (x <= listA[k])
            j = k-1;
        if (listA[k] <= x)
            i = k+1;
    }
    while (i <= j);
    if (listA[k] == x)
        return(k);
    else
        return -1;
}
```

Which one of the following statements about the function ProcessArray is CORRECT?
A) It will run into an infinite loop when x is not in listA.
B) It is an implementation of binary search.
C) It will always find the maximum element in listA.
D) It will return −1 even when x is present in listA.

OUTPUT: B
5. Consider a sorted array of n numbers. What would be the time complexity of the best known algorithm to find a pair 'a' and 'b' such that |a-b| = k , k being a positive integer.
A) O(n)
B) O(n log n)
C) O(n ^ 2)
D) O(log n)

OUTPUT: A
Explanation: Just maintain two pointers at the start and accordingly increment one of them depending upon whether difference is less than or greater than k. Just a single pass is required so the answer is O(n).

6. The average number of key comparisons done in a successful sequential search in a list of length it is
A) log n
B) (n-1)/2
C) n/2
D) (n+1)/2
OUTPUT: D
Explanation: If element is at 1 position then it requires 1 comparison. If element is at 2 position then it requires 2 comparison. If element is at 3 position then it requires 3 comparison. Similarly, If element is at n position then it requires n comparison.
Total comparison
= n(n+1)/2
For average comparison
= (n(n+1)/2) / n
= (n+1)/2
7. The recurrence relation that arises in relation with the complexity of binary search is:
A) T(n) = 2T(n/ 2) + k , where k is constant
B) T(n) = T(n / 2) + k , where k is constant
C) T(n) = T(n / 2) + log n
D) T(n) = T(n / 2) + n
OUTPUT: D
8. Suppose there are 11 items in sorted order in an array. How many searches are required on the average, if binary search is employed and all searches are successful in finding the item?
A) 3.00
B)3.46
C) 2.81
D)3.33
OUTPUT: A
9. The average case occurs in the Linear Search Algorithm when:
A) The item to be searched is in some where middle of the Array
B) The item to be searched is not in the array
C) The item to be searched is in the last of the array
D) The item to be searched is either in the last or not in the array
OUTPUT: A
10. Suppose that we have numbers between 1 and 1000 in a binary search tree and we want to search for the number 365. Which of the following sequences could not be the sequence of nodes examined?
A) 4, 254, 403, 400, 332, 346, 399, 365
B) 926, 222, 913, 246, 900, 260, 364, 365
C) 927, 204,913, 242, 914, 247, 365
D) 4, 401, 389, 221, 268, 384, 383, 280, 365
OUTPUT: C
11. Consider a sorted array of n numbers and a number x. What would be the time complexity of the best known algorithm to find a triplet with sum equal to x. For

example, arr[] = {1, 5, 10, 15, 20, 30}, x = 40. Then there is a triplet {5, 15, 20} with sum 40.

A)O(n)
B) O(n^2)
C) O(n Log n)
D) O(n^3)
 OUTPUT: B

12.The worst case occurred in the linear search algorithm when

A) The element in the middle of an array
B) Item present in the last
C) Item present in the starting
D) Item has maximum value
OUTPUT : B

13. What is the average case occur in the linear search when

A) The element in the middle of an array
B) Item present in the last
C) Item present in the starting
D) Item has maximum value
OUTPUT: A

14.Choose the code snippet which uses recursion for linear search.

a)

```
public void linSearch(int[] arr, int first, int last, int key)
{
      if(first == last)
    {
            System.out.print("-1");
      }
      else
    {
            if(arr[first] == key)
        {
                  System.out.print(first);
            }
            else
        {
                  linSearch(arr, first+1, last, key);
            }
      }
}
```

b)

```
   public void linSearch(int[] arr, int first, int last, int key)
   {
            if(first == last)
        {
                  System.out.print("-1");
            }
            else
        {
                  if(arr[first] == key)
            {
                        System.out.print(first);
                  }
                  else
            {
                        linSearch(arr, first+1, last-1, key);
                  }
            }
   }
```

c)

```
public void linSearch(int[] arr, int first, int last, int key)
{
      if(first == last)
    {
            System.out.print("-1");
      }
      else
    {
            if(arr[first] == key)
        {
                  System.out.print(last);
            }
            else
        {
                  linSearch(arr, first+1, last, key);
            }
      }
}
```

d)

```
public void linSearch(int[] arr, int first, int last, int key)
{
      if(first == last)
    {
            System.out.print("-1");
      }
      else
    {
            if(arr[first] == key)
        {
                  System.out.print(first);
            }
            else
        {
                  linSearch(arr, first+1, last+1, key);
            }
      }
}
```

Answer: A

Explanation: Every time check the key with the array value at first index, if it is not equal then call the function again with an incremented first index.

15. The array is as follows: 1,2,3,6,8,10. At what time the element 6 is found? (By using linear search(recursive) algorithm)

a) 4th call

b) 3rd call

c) 6th call

d) 5th call

Answer: A

Explanation: Provided that the search starts from the first element, the function calls itself till the element is found. In this case, the element is found in 4th call.

16. What is the recurrence relation for the linear search recursive algorithm?

a) T(n-2)+c

b) 2T(n-1)+c

c) T(n-1)+c

d) T(n+1)+c

Answer: C

Explanation: After each call in the recursive algorithm, the size of n is reduced by 1. Therefore the optimal solution is T(n-1)+c.

17. Which of the following code snippet performs linear search recursively?

a)

```
for(i=0;i<n;i++)
   {
              if(a[i]==key)
              printf("element found");
   }
```

b)

```
LinearSearch(int[] a, n,key)
   {
              if(n<1)
              return False
              if(a[n]==key)
              return True
              else
              LinearSearch(a,n-1,key)
   }
```

c)

```
LinearSearch(int[] a, n,key)
   {
              if(n<1)
              return True
              if(a[n]==key)
              return False
              else
              LinearSearch(a,n-1,key)
   }
```

d)

```
LinearSearch(int[] a, n,key)
```

```
        {
                if(n<1)
                return False
                if(a[n]==key)
                return True
                else
                LinearSearch(a,n+1,key)
        }
```

Answer: B
Explanation: Compare n with first element in arr[]. If element is found at first position, return it. Else recur for remaining array and n.
18. The array is as follows: 1,2,3,6,8,10. Given that the number 17 is to be searched. At which call it tells that there's no such element? (By using linear search(recursive) algorithm)
a) 7th call
b) 9th call
c) 17th call
d) The function calls itself infinite number of times
Answer: A
Explanation: The function calls itself till the element is found. But at the 7th call it terminates as goes outside the array.
19. Given an input arr = {2,5,7,99,899}; key = 899; What is the level of recursion?
a)5
b)2
c)3
d)4

20. Given an array arr = {45,77,89,90,94,99,100} and key = 99; what are the mid values(corresponding array elements) in the first and second levels of recursion?
a)90and99
b)90and94
c)89and99
d) 89 and 94

21. Which algorithmic technique does Fibonacci search use?
a) Brute force
b) Divide and Conquer
c) Greedy Technique
d) Backtracking
Answer: b
Explanation: With every iteration, we divide the given array into two sub arrays(not necessarily equal).

22. Choose the recursive formula for the Fibonacci series.(n>=1)
a) F(n) = F(n+1) + F(n+2)
b) F(n) = F(n) + F(n+1)
c) F(n) = F(n-1) + F(n-2)
d) F(n) = F(n-1) – F(n-2)
Answer: c

23.Suppose we want to arrange the n numbers stored in any array such that all negative values occur before all positive ones. Minimum number of exchanges required in the worst case is

a) N-1
b) N
c) N+1
d) None of these

Explanation:

Worst case happens when all the positive numbers occur before the negative numbers.
In this case, take a positive number from the left side and negative number from the right side and do exchange operation. Then after n/2 exchanges operation, you will reach middle of the array and all the negative value will be present before positive value.so in worst case n/2 exchanges required.

24. A and B are two sorted lists of integers such that A[n]<B[1].An element is to be searched in both the lists. Maximum no of comparisons using binary search is

A) O(n)
B) O($\log_2$ 2n)
C) O($\log_2$ 2n+1)
D) O(log n)
Answer: B

25. At most, how many comparisons are required to search a sorted vector of 1023 elements using the binary search algorithm?

a) 10
b) 15
c) 20
d) 30
Answer: A

26.Suppose DATA array contains 1000000 elements. Using the binary search algorithm, one requires only about n comparisons to find the location of an item in the DATA array, then n is

a) 7
b) 20
c) 38
d) 45
Answer: B

28.Which of the following statements is used in the binary search algorithm to halve the array?

a) middle Sub = middle Sub/2;
b) middle Sub = (stop Sub - start Sub)/2;
c) middle Sub = start Sub + stop Sub/2;
d) middle Sub = (start Sub + stop Sub)/2;
Answer: D

CHAPTER-8
SORTING

INTRODUCTION TO SORTING

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

- **Telephone Directory** − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.
- **Dictionary** − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

There are two different categories in sorting:

- **Internal sorting**: If the input data is such that it can be adjusted in the main memory at once, it is called internal sorting.
- **External sorting**: If the input data is such that it cannot be adjusted in the memory entirely at once, it needs to be stored in a hard disk, floppy disk, or any other storage device. This is called external sorting.

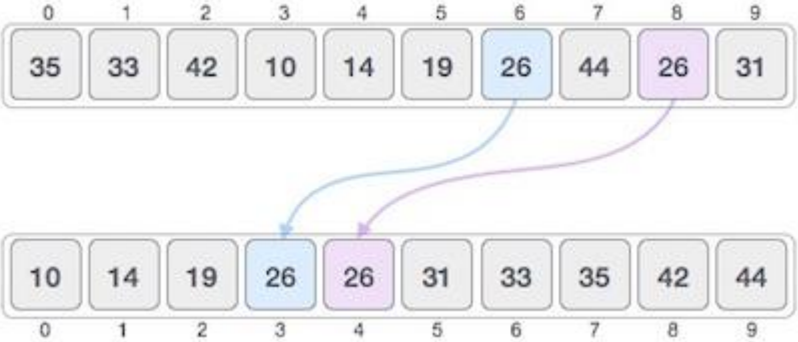
In-place Sorting and Not-in-place Sorting

Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called **in-place sorting**. Bubble sort is an example of in-place sorting.

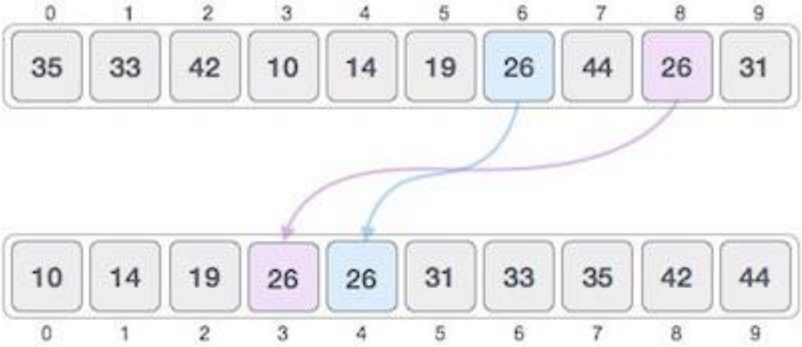
However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called **not-in-place sorting**. Merge-sort is an example of not-in-place sorting.

Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Adaptive and Non-Adaptive Sorting Algorithm

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

Counting Sort

It is a sorting technique based on the keys i.e. objects are collected according to keys which are small integers. Counting sort calculates the number of occurrence of objects and stores its key values. New array is formed by adding previous key elements and assigning to objects.

Time Complexity: O(n+k) is worst case where n is the number of element and k is the range of input.

Space Complexity: O(k) k is the range of input.

Complexity	Best Case	Average Case	Worst Case
Time Complexity	Ω(n+k)	θ(n+k)	O(n+k)
Space Complexity			O(k)

Limitation of Counting Sort

- It is effective when range is not greater than number of object.
- It is not comparison based complexity.
- It is also used as sub-algorithm for different algorithm.
- It uses partial hashing technique to count the occurrence.
- It is also used for negative inputs.

Algorithm

- STEP 1 START
- STEP 2 Store the input array
- STEP 3 Count the key values by number of occurrence of object

- STEP 4 Update the array by adding previous key elements and assigning to objects
- STEP 5 Sort by replacing the object into new array and key= key-1
- STEP 6 STOP

Comparison based sorting –

In comparison based sorting, elements of an array are compared with each other to find the sorted array.

- Bubble sort and Insertion sort –

- Selection sort
- Merge sort
- Heap sort
- Quick sort

Non-comparison based sorting –

In non-comparison based sorting, elements of array are not compared with each other to find the sorted array.

- Radix sort
- Count Sort
- Bucket Sort

Bubble Sort

Bubble Sort is a comparison based sorting algorithm. In this algorithm adjacent elements are compared and swapped to make the correct sequence. This algorithm is simpler than other algorithms, but it has some drawbacks also. This algorithm is not suitable for a large number of data set. It takes much time to solve the sorting tasks.

The complexity of the Bubble Sort Technique

- **Time Complexity:** O(n) for best case, O(n^2) for average and worst case
- **Space Complexity:** O(1)

Input and Output

```
Input:
A list of unsorted data: 56 98 78 12 30 51
Output:
Array after Sorting: 12 30 51 56 78 98
```

Algorithm

```
bubbleSort( array, size)
```

Input − An array of data, and the total number in the array

Output − The sorted Array

```
Begin
   for i := 0 to size-1 do
      flag := 0;
      for j:= 0 to size –i – 1 do
         if array[j] > array[j+1] then
            swap array[j] with array[j+1]
            flag := 1
      done

      if flag ≠ 1 then
         break the loop.
```

```
  done
End
```

Insertion Sort

This sorting technique is similar with the card sorting technique, in other words, we sort cards using insertion sort mechanism. For this technique, we pick up one element from the data set and shift the data elements to make a place to insert back the picked up an element into the data set.

The complexity of the Insertion Sort Technique

- Time Complexity: O(n) for best case, O(n^2) for average and worst case
- Space Complexity: O(1)

Input and Output

```
Input:
The unsorted list: 9 45 23 71 80 55
Output:
Array before Sorting: 9 45 23 71 80 55
Array after Sorting: 9 23 45 55 71 80
```

Algorithm

```
insertionSort(array, size)
```

Input − An array of data, and the total number in the array

Output &− The sorted Array

```
Begin
  for i := 1 to size-1 do
    key := array[i]
    j := i
    while j > 0 AND array[j-1] > key do
      array[j] := array[j-1];
      j := j – 1
    done
    array[j] := key
  done
End
```

Selection Sort

In the selection sort technique, the list is divided into two parts. In one part all elements are sorted and in another part the items are unsorted. At first, we take the maximum or minimum data from the array. After getting the data (say minimum) we place it at the beginning of the list by replacing the data of first place with the minimum data. After performing the array is getting smaller. Thus this sorting technique is done.

The complexity of Selection Sort Technique

- Time Complexity: O(n^2)
- Space Complexity: O(1)

Input and Output

```
Input:
The unsorted list: 5 9 7 23 78 20
Output:
Array before Sorting: 5 9 7 23 78 20
Array after Sorting: 5 7 9 20 23 78
```

Algorithm

```
selectionSort(array, size)
```

Input − An array of data, and the total number in the array
Output − The sorted Array

```
Begin
  for i := 0 to size-2 do //find minimum from ith location to size
    iMin := i;
    for j:= i+1 to size – 1 do
      if array[j] < array[iMin] then
        iMin := j
    done
    swap array[i] with array[iMin].
  done
End
```

<u>Merge Sort</u>

The merge sort technique is based on divide and conquers technique. We divide the whole dataset into smaller parts and merge them into a larger piece in sorted order. It is also very effective for worst cases because this algorithm has lower time complexity for the worst case also.

The complexity of Merge Sort Technique

- **Time Complexity:** O(n log n) for all cases
- **Space Complexity:** O(n)

Input and Output

```
Input:
The unsorted list: 14 20 78 98 20 45
Output:
Array before Sorting: 14 20 78 98 20 45
Array after Sorting: 14 20 20 45 78 98
```

Algorithm

merge(array, left, middle, right)

Input − The data set array, left, middle and right index
Output − The merged list

```
Begin
  nLeft := m - left+1
  nRight := right – m
  define arrays leftArr and rightArr of size nLeft and nRight respectively

  for i := 0 to nLeft do
    leftArr[i] := array[left +1]
  done

  for j := 0 to nRight do
    rightArr[j] := array[middle + j +1]
  done
  i := 0, j := 0, k := left
  while i < nLeft AND j < nRight do
    if leftArr[i] <= rightArr[j] then
      array[k] = leftArr[i]
      i := i+1
    else
      array[k] = rightArr[j]
      j := j+1
    k := k+1
  done

  while i < nLeft do
    array[k] := leftArr[i]
    i := i+1
    k := k+1
  done

  while j < nRight do
    array[k] := rightArr[j]
    j := j+1
    k := k+1
  done
End
```

mergeSort(array, left, right)
Input − An array of data, and lower and upper bound of the array
Output − The sorted Array

```
Begin
  if lower < right then
    mid := left + (right - left) /2
    mergeSort(array, left, mid)
    mergeSort (array, mid+1, right)
    merge(array, left, mid, right)
End
```

Quick Sort

The quicksort technique is done by separating the list into two parts. Initially, a pivot element is chosen by partitioning algorithm. The left part of the pivot holds the smaller values than the pivot, and right part holds the larger value. After partitioning, each separate lists are partitioned using the same procedure.

The complexity of Quicksort Technique

- Time Complexity: O(n log n) for best case and average case, O(n^2) for the worst case.
- Space Complexity: O(log n)

Input and Output

```
Input:
The unsorted list: 90 45 22 11 22 50
Output:
Array before Sorting: 90 45 22 11 22 50
Array after Sorting: 11 22 22 45 50 90
```

Algorithm
partition(array, lower, upper)
Input: The data set array, lower boundary and upper boundary
Output: Pivot in the correct position

```
Begin
   pivot := array[lower]
   start := lower and end := upper
   while start < end do
      while array[start] <= pivot AND start < end do
         start := start +1
      done

      while array[end] > pivot do
         end := end – 1
      done
      if start < end then
         swap array[start] with array[end]
   done

   array[lower] := array[end]
   array[end] := pivot
   return end
End
```

quickSort(array, left, right
Input − An array of data, and lower and upper bound of the array
Output − The sorted Array

```
Begin
   if lower < right then
      q = partition(arraym left, right)
      quickSort(array, left, q-1)
      quickSort(array, q+1, right)
End
```

<u>Radix Sort</u>

Radix sort is a non-comparative sorting algorithm. This sorting algorithm works on the integer keys by grouping digits which share the same position and value. The radix is the base of a number system. As we know that in the decimal system the radix or base is 10. So for sorting some decimal numbers, we need 10 positional boxes to store numbers.

The complexity of Radix Sort Technique

- Time Complexity: O(nk)
- Space Complexity: O(n+k)

Input and Output

```
Input:
The unsorted list: 802 630 20 745 52 300 612 932 78 187
Output:
Data before Sorting: 802 630 20 745 52 300 612 932 78 187
Data after Sorting: 20 52 78 187 300 612 630 745 802 932
```

Algorithm

```
radixSort(array, size, maxDigit)
```

Input − An array of data, and the total number in the array, digit count of max number
Output − Sorted array.

```
Begin
   define 10 lists as pocket
   for i := 0 to max -1 do
      m = 10^i+1
      p := 10^i
      for j := 0 to n-1 do
         temp := array[j] mod m
         index := temp / p
         pocket[index].append(array[j])
      done

      count := 0
      for j := 0 to radix do
         while pocket[j] is not empty
            array[count] := get first node of pocket[j] and delete it
            count := count +1
      done
   done
End
```

<u>Shell Sort</u>

The shell sorting technique is based on the insertion sort. In the insertion sort sometimes we need to shift large block to insert an item in the correct location. Using shell sort, we can avoid a large number of shifting. The sorting is done with a specific interval. After each pass, the interval is reduced to make the smaller interval.

The complexity of the Shell Sort Technique

- Time Complexity: O(n log n) for best case, and for other cases, it depends on the gap sequence.
- Space Complexity: O(1)

Input and Output

```
Input:
The unsorted list: 23 56 97 21 35 689 854 12 47 66
Output:
Array before Sorting: 23 56 97 21 35 689 854 12 47 66
Array after Sorting: 12 21 23 35 47 56 66 97 689 854
```

Algorithm

```
shellSort(array, size)
```

Input − An array of data, and the total number in the array
Output − The sorted Array

```
Begin
   for gap := size / 2, when gap > 0 and gap is updated with gap / 2 do
```

```
    for j:= gap to size– 1 do
      for k := j-gap to 0, decrease by gap value do
        if array[k+gap] >= array[k]
          break
        else
          swap array[k + gap] with array[k]
      done
    done
  done
End
```

Heap Sort

Heap sort is performed on the heap data structure. We know that heap is a complete binary tree. Heap tree can be of two types. Min-heap or max heap. For min heap the root element is minimum and for max heap the root is maximum. After forming a heap, we can delete an element from the root and send the last element to the root. After these swapping procedure, we need to re-heap the whole array. By deleting elements from root we can sort the whole array.

The complexity of Heap Sort Technique

- **Time Complexity:** O(n log n)
- **Space Complexity:** O(1)

Input and Output

```
Input:
A list of unsorted data: 30 8 99 11 24 39
Output:
Array before Sorting: 30 8 99 11 24 39
Array after Sorting: 8 11 24 30 39 99
```

Algorithm

heapify(array, size)

Input − An array of data, and the total number in the array

Output − The max heap using an array element

```
Begin
  for i := 1 to size do
    node := i
    par := floor (node / 2)
    while par >= 1 do
      if array[par] < array[node] then
        swap array[par] with array[node]
      node := par
      par := floor (node / 2)
    done
  done
End
```

heapSort(array, size)

Input: An array of data, and the total number in the array

Output −nbsp;sorted array

```
Begin
```

```
  for i := n to 1 decrease by 1 do
    heapify(array, i)
    swap array[1] with array[i]
  done
End
```

Time and Space Complexity Comparison Table:

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	**$\Omega(N)$**	**$\Theta(N^2)$**	**$O(N^2)$**	**$O(1)$**
Selection Sort	**$\Omega(N^2)$**	**$\Theta(N^2)$**	**$O(N^2)$**	**$O(1)$**
Insertion Sort	**$\Omega(N)$**	**$\Theta(N^2)$**	**$O(N^2)$**	**$O(1)$**
Merge Sort	**$\Omega(N \log N)$**	**$\Theta(N \log N)$**	**$O(N \log N)$**	**$O(N)$**
Heap Sort	**$\Omega(N \log N)$**	**$\Theta(N \log N)$**	**$O(N \log N)$**	**$O(1)$**
Quick Sort	**$\Omega(N \log N)$**	**$\Theta(N \log N)$**	**$O(N^2)$**	**$O(N \log N)$**
Radix Sort	**$\Omega(N k)$**	**$\Theta(N k)$**	**$O(N k)$**	**$O(N + k)$**
Count Sort	**$\Omega(N + k)$**	**$\Theta(N + k)$**	**$O(N + k)$**	**$O(k)$**
Bucket Sort	**$\Omega(N + k)$**	**$\Theta(N + k)$**	**$O(N^2)$**	**$O(N)$**

TOPIC-SORTING
ESSENTIAL QUIZ

1. A stable sorting algorithm –
    a) does not crash.
    b) does not run out of memory.
    c) does not change the sequence of appearance of elements.
    d) does not exists.

Answer : C

Explanation: A stable sorting algorithm like bubble sort, does not change the sequence of appearance of similar element in the sorted list.

2. Which of the following is example of in-place algorithm?
    a) Bubble Sort
    b) Merge Sort
    c) Insertion Sort
    d) All of the above

Answer : B

Explanation

Only Merge sort requires extra space.

3. Which of the following is not a stable sorting algorithm?

    a) Insertion sort

b) Selection sort
c) Bubble sort
d) Merge sort

ANSWER: B

4. Which of the following is a stable sorting algorithm?

a) Merge sort
b) Typical in-place quick sort
c) Heap sort
d) Selection sort

ANSWER: A

5. Which of the following is not an in-place sorting algorithm?

a) Selection sort
b) Heap sort
c) Quick sort
d) Merge sort

ANSWER: D

6. Running merge sort on an array of size n which is already sorted is

a) O(n)
b) O(nlogn)
c) O(n2)
d) None

ANSWER: B

7. The time complexity of a quick sort algorithm which makes use of median, found by an O(n) algorithm, as pivot element is

a) O(n2)
b) O(nlogn)
c) O(nloglogn)
d) O(n)

ANSWER: B

8. Which of the following is not a non-comparison sort?

a) Counting sort
b) Bucket sort
c) Radix sort
d) Shell sort

ANSWER: D

9. The time complexity of heap sort in worst case is

    a) O(logn)
    b) O(n)
    c) O(nlogn)
    d) O(n2)

ANSWER: C

10. If the given input array is sorted or nearly sorted, which of the following algorithm gives the best performance?

    a) Insertion sort
    b) Selection sort
    c) Quick sort
    d) Merge sort

ANSWER: A

11. Which of the following algorithm pays the least attention to the ordering of the elements in the input list?

    a) Insertion sort
    b) Selection sort
    c) Quick sort
    d) None

ANSWER: B

12. Consider the situation in which assignment operation is very costly. Which of the following sorting algorithm should be performed so that the number of assignment operations is minimized in general?

    a) Insertion sort
    b) Selection sort
    c) Heap sort
    d) None

ANSWER: B

13. Time complexity of bubble sort in best case is

    a) θ (n)
    b) θ (nlogn)
    c) θ (n2)
    d) θ (n(logn) 2)

ANSWER: A

14. Given a number of elements in the range [0….n3]. which of the following sorting algorithms can sort them in O(n) time?

a) Counting sort
b) Bucket sort
c) Radix sort
d) Quick sort

ANSWER: C

15. Which of the following algorithms has lowest worst case time complexity?

a) Insertion sort
b) Selection sort
c) Quick sort
d) Heap sort

ANSWER: D

16. Which of the following sorting algorithms is/are stable

a) Counting sort
b) Bucket sort
c) Radix sort
d) All of the above

ANSWER: D

17. Counting sort performs …………. Numbers of comparisons between input elements.

a) 0
b) n
c) nlogn
d) n2

ANSWER: A

18. The running time of radix sort on an array of n integers in the range [0……..n5 -1] when using base 10 representation is

a) θ (n)
b) θ (nlogn)
c) θ (n2)
d) none

ANSWER: B

19. The running time of radix sort on an array of n integers in the range [0……..n5 -1] when using base n representation is

a) θ (n)
b) θ (nlogn)
c) θ (n2)
d) None

ANSWER: A

20. Which of the following sorting algorithm is in-place

    a) Counting sort
    b) Radix sort
    c) Bucket sort
    d) None

ANSWER: B

TOPIC-SORTING
ESSENTIAL QUIZ

1. The radix sort does not work correctly if each individual digit is sorted using

    a) Insertion sort
    b) Counting sort
    c) Selection sort
    d) Bubble sort

ANSWER: C

2. Which of the following sorting algorithm has the running time that is least dependent on the initial ordering of the input?

    a) Insertion sort
    b) Quick sort
    c) Merge sort
    d) Selection sort

ANSWER: D

3. Time complexity to sort elements of binary search tree is

    a) O(n)
    b) O(nlogn)
    c) O(n2)
    d) O(n2logn)

ANSWER: A

4. The lower bound on the number of comparisons performed by comparison-based sorting algorithm is

    a) Ω (1)
    b) Ω (n)
    c) Ω (nlogn)
    d) Ω (n2)

ANSWER: C

5. Which of the following algorithm(s) can be used to sort n integers in range [1…..n3] in O(n) time?

a) Heap sort
b) Quick sort
c) Merge sort
d) Radix sort

ANSWER: D

6. Which of the following algorithm design technique is used in the quick sort algorithm?

a) Dynamic programming
b) Backtracking
c) Divide-and-conquer
d) Greedy method

ANSWER: C

7. Merge sort uses

a) Divide-and-conquer
b) Backtracking
c) Heuristic approach
d) Greedy approach

ANSWER: A

8. For merging two sorted lists of size m and n into sorted list of size m+n, we require comparisons of

a) O(m)
b) O(n)
c) O(m+n)
d) O(logm + logn)

ANSWER: C

9. A sorting technique is called stable if it

a) Takes O(nlogn) times
b) Maintains the relative order of occurrence of non-distinct elements
c) Uses divide-and-conquer paradigm
d) Takes O(n) space

ANSWER: B

10. In a heap with n elements with the smallest element at the root, the seventh smallest element can be found in time

a) θ (nlogn)
b) θ (n)
c) θ (logn)
d) θ (1)

ANSWER: A

11. What would be the worst case time complexity of the insertion sort algorithm, if the inputs are restricted to permutation of 1…..n with at most n inversion?

    a) θ (n2)
    b) θ (nlogn)
    c) θ (n1.5)
    d) θ (n)

ANSWER: D

12. In a binary max heap containing n numbers, the smallest element can be found in time

    a) θ (n)
    b) θ (logn)
    c) θ (loglogn)
    d) θ (1)

ANSWER: A

13. What is the advantage of bubble sort over other sorting techniques?
    a) It is faster
    b) Consumes less memory
    c) Detects whether the input is already sorted
    d) All of the mentioned

Answer: Option C

Explanation:

Bubble sort is one of the simplest sorting techniques and perhaps the only advantage it has over other techniques is that it can detect whether the input is already sorted.

14. The given array is arr = {1,2,4,3}. Bubble sort is used to sort the array elements. How many iterations will be done to sort the array?
    a) 4
    b) 2
    c) 1
    d) 0

Answer: Option A

Explanation:

Even though the first two elements are already sorted, bubble sort needs 4 iterations to sort the given array.

15. What is the best case complexity of QuickSort?
    a) O(nlogn)
    b) O(logn)
    c) O(n)
    d) O(n2)

Answer: Option A

Explanation:

The array is partitioned into equal halves, using the Divide and Conquer master theorem, the complexity is found to be O(nlogn).

16. The given array is arr = {2,3,4,1,6}. What are the pivots that are returned as a result of subsequent partitioning?
    a) 1 and 3
    b) 3 and 1

c) 2 and 6
d) 6 and 2

Answer: Option A
Explanation:
The call to partition returns 1 and 3 as the pivot elements.

17. In addition to the pancake sorting problem, there is the case of the burnt pancake problem in which we are dealing with pancakes (discs) that are burnt on one side only. In this case it is taken that the burnt side must always end up _______
    a) Faced down
    b) Faced up
    c) It doesn’t matter
    d) Both sides are burnt

Answer: Option A
Explanation:
A varation of this pancake is with burnt pancakes. Here each pancake has a burnt side and all pancakes must, in addition, end up with the burnt side on bottom. It is a more difficult version of the regular pancake problem.

18. What is the disadvantage of selection sort?
    a) It requires auxiliary memory
    b) It is not scalable
    c) It can be used for small keys
    d) None of the mentioned

Answer: Option B
Explanation:
As the input size increases, the performance of selection sort decreases.

19. The time complexity of heap sort is ....

    a) O(n)

    b) O(logn)

    c) O(n2)

    d) O(n logn)

ANSWER:D

20. If the number of record to be sorted large and the key is long, then ...... sorting can be efficient.

    a) Merge

    b) Heap

    c) Quick

    d) Bubble

Answer: Option C

TOPIC-SORTING
ESSENTIAL QUIZ

1. If the number of recoSrd to be sorted large and the key is short, then ...... sorting can be efficient.

    a) Merge

    b) Heap

    c) Radix

    d) Bubble

Answer: Option C

2. The total number of comparisons in a bubble sort is ....

    a) O(n logn)

    b) O(2n)

    c) O(n2)

    d) O(n)

Answer: Option A

3. Selection sort first finds the .... element in the list and put it in the first position.

    a) Middle element

    b) Largest element

    c) Last element

    d) Smallest element

Answer: Option D

4. Quick sort is also known as ....

    a) merge sort

    b) tree sort

    c) shell sort

    d) partition and exchange sort

Answer: Option D

5. The operation that combines the element is of A and B in a single sorted list C with n=r+s element is called ....

    a) Inserting

    b) Mixing

    c) Merging

    d) Sharing

Answer: Option C

6. A tree sort is also known as .... sort.

    a) quick

    b) shell

    c) heap

    d) selection

Answer: Option C

7. .... sorting is good to use when alphabetizing large list of names.

    a) Merge

    b) Heap

    c) Radix

    d) Bubble

Answer: Option C

8. The easiest sorting is ....

    a) quick sort

    b) shell sort

    c) heap sort

    d) selection sort

Answer: Option D

9. Which of the following sorting algorithm is of divide and conquer type?

a) Bubble sort

b) Insertion sort

c) Quick sort

d) Merge sort

Answer: Option C

10. Merging k sorted tables into a single sorted table is called ......

a) k way merging

b) k th merge

c) k+1 merge

d) k-1 merge

Answer: Option A

11. The function used to modify the way of sorting the keys of records is called ....

a) Indexing function

b) Hash function

c) Addressing function

d) All of the above

Answer: Option B

12. Which of the following is an external sorting?

a) Insertion Sort
b) Bubble Sort
c) Merge Sort
d) Tree Sort

Answer: Option C

13. Very slow way of sorting is ....

a) Insertion sort
b) Heap sort
c) Bubble sort
d) Quick sort

Answer: Option A

14. Which of the following is an internal sorting?

a) Tape Sort
b) 2-way Merge Sort
c) Merge Sort

d) Tree Sort

Answer: Option D

15. Sorting a file F usually refers to sorting F with respect to a particular key called .....

a) Basic key

b) Primary key

c) Starting key

d) Index key

Answer: Option B

16. What is not true about insertion sort?

a) Exhibits the worst case performance when the initial array is sorted in reverse order.

b) Worst case and average case performance is O(n2)

c) Can be compared to the way a card player arranges his card from a card deck.

d) None of the above.

Answer: (d).None of the above.

17. A pivot element to partition unsorted list is used in

a) Merge Sort

b) Quick Sort

c) Insertion Sort

d) Selection Sort

Answer: (b).Quick Sort

18. Time required to merge two sorted lists of size m and n, is

a) O(m | n)

b) O(m + n)

c) O(m log n)

d) O(n log m)

Answer: (b)

19. Quick sort running time depends on the selection of

a) size of array

b) pivot element

c) sequence of values

d) none of the above

Answer: (b).pivot element

20. Which one of the below is not divide and conquer approach?

a) Insertion Sort

b) Merge Sort

c) Shell Sort

d) Heap Sort

Answer: (b).Merge Sort

21. Which of the below given sorting techniques has highest best-case runtime complexity

a) quick sort

b) selection sort

c) insertion sort

d) bubble sort

Answer: (b).selection sort

22. How many swaps are required to sort the given array using bubble sort - { 2, 5, 1, 3, 4} ?
    a) 4
    b) 5
    c) 6
    d) 7

Answer: (a).4

23. An adaptive sorting algorithm −
    a) adapts to new computers.
    b) takes advantage of already sorted elements.
    c) takes input which is already sorted.
    d) none of the above.

Answer: (b).takes advantage of already sorted elements.

24. The following sorting algorithms maintain two sub-lists, one sorted and one to be sorted −
    a) Selection Sort
    b) Insertion Sort
    c) Merge Sort
    d) both A & B

Answer: (d).

25. You have to sort a list L consisting of a sorted list followed by a few “random” elements. Which of the following sorting methods would be especially suitable for such a task?
    a) Bubble sort
    b) Selection sort
    c) Quick sort
    d) Insertion sort

Answer: (d).

26. A sort which relatively passes through a list to exchange the first element with any element less than it and then repeats with a new first element is called
    a) insertion sort
    b) selection sort
    c) heap sort
    d) quick sort

Answer: (d)

27. Which of the following sorting algorithms does not have a worst case running time of O (n^2) ?
    a) Insertion sort
    b) Merge sort
    c) Quick sort
    d) Bubble sort

Answer: (b).

28. The total number of companions required to merge 4 sorted files containing 15, 3, 9 and 8 records into a single sorted file is
    a) 66
    b) 39
    c) 33
    d) 15

Answer: (c).

29. Which of the following sorting methods would be most suitable for sorting a list which is almost sorted
    a) Bubble Sort
    b) Insertion Sort
    c) Selection Sort
    d) Quick Sort

Answer: (a).

30. 2. Consider that n elements are to be sorted. What is the time complexity of Bubble sort?
    a) O(1)
    b) O(logn)
    c) O(n)
    d) O(n^2)

Answer: (d)

31. Which array is the fastest on an average, but sometimes unbalanced partitions can lead to very slow sorting?
    a) Insertion sort
    b) Selection sort
    c) Quick sort
    d) None of the above

Answer: (c)

32. _________is efficient for data sets which are already substantially sorted. The time complexity is O(n + d), where d is the number of inversions.
    a) Insertion sort
    b) Sorted array
    c) Unsorted array
    d) None of the above

Answer: (a)

33. A desirable choice for the partitioning element in quick sort is
    a) First element of the list
    b) Last element of the list
    c) Randomly chosen element of the list
    d) Median of the list

Answer: (a).

34. A sorting algorithm is stable if
    a) its time complexity is constant irrespective of the nature of input

b) preserves the original order of records with equal keys
c) its space complexity is constant irrespective of the nature of input
d) it sorts any volume of data in a constant time

Answer: (b)

35. Match the following:

List - I

a) Bubble Sort
b) Shell Sort
c) Selection Sort

List - II

d) O(n)
e) O(n2)
f) O(n log n)

a. i → a, ii → b, iii → c
b. i → b, ii → c, iii → a
c. i → a, ii → c, iii → b
d. i → b, ii → a, iii → c

Answer: (b).

36. The largest and the second largest number from a set of n distinct numbers can be found in

a) O(n)
b) O(2n)
c) O(n^2)
d) O(log n) Answer: (a).

Answer: (a).

37. Which of the following algorithms exhibits the unnatural behavior that, minimum number of comparisons are needed if the list to be sorted is in the reverse sorted order and maximum number of comparisons are needed if they are already in sorted order?

a) heap sort
b) Radix sort
c) Binary insertion sort
d) There can't be any such sorting method

Answer: (c).

38. Which of the following sorting algorithm has the worst time complexity of n log(n)?

a) Heap sort
b) Quick sort
c) Selection sort
d) Insertion sort

Answer: (a).

39. Which of the following sorting methods sorts a given set of items that is already in sorted order or in reverse sorted order with equal speed?

a) Heap sort
b) Quick sort

c) Selection sort
d) Insertion sort

Answer: (b)

40. A sort which compares adjacent elements in a list and switches where necessary is
    a) heap sort
    b) quick sort
    c) bubble sort
    d) insertion sort

Answer: (c)

41. A sort which iteratively passes through a list to exchange the first element with any element less than it and then repeats with a new first element is called
    a) heap sort
    b) quick sort
    c) selection sort
    d) insertion sort

Answer: (c)

42. What is recurrence for worst case of QuickSort and what is the time complexity in Worst case?
    a) Recurrence is $T(n) = T(n-2) + O(n)$ and time complexity is $O(n^2)$
    b) Recurrence is $T(n) = T(n-1) + O(n)$ and time complexity is $O(n^2)$
    c) Recurrence is $T(n) = 2T(n/2) + O(n)$ and time complexity is O(nLogn)
    d) Recurrence is $T(n) = T(n/10) + T(9n/10) + O(n)$ and time complexity is O(nLogn)

Answer: (b)

43. Suppose we have a O(n) time algorithm that finds median of an unsorted array. Now consider a QuickSort implementation where we first find median using the above algorithm, then use median as pivot. What will be the worst case time complexity of this modified QuickSort.
    a) $O(n^2$ Logn)
    b) $O(n^2)$
    c) O(n Logn Logn)
    d) O(nLogn)

Answer: (d)

44. What is the advantage of bubble sort over other sorting techniques?
    a) It is faster
    b) Consumes less memory
    c) Detects whether the input is already sorted
    d) All of the mentioned

Answer: (c)

45. The given array is arr = {3,4,5,2,1}. The number of iterations in bubble sort and selection sort respectively are,
    a) 5 and 4
    b) 4 and 5
    c) 2 and 4

d) 2 and 5

Answer: (a)

ADVANCE LEVEL QUIZ
TOPIC: SORTING

Q1. Suppose we are sorting an array of eight integers using quicksort, and we have just finished the first partitioning with the array looking like this:

2 5 1 7 9 12 11 10

Which statement is correct?

a. The pivot could be either the 7 or the 9.

b. The pivot could be the 7, but it is not the 9

c. The pivot is not the 7, but it could be the 9

d. Neither the 7 nor the 9 is the pivot.

Answer: (a).

Q2. Consider the situation in which assignment operation is very costly. Which of the following sorting algorithm should be performed so that the number of assignment operations is minimized in general?

a. Insertion sort

b. Selection sort

c. Heap sort

d. None

Answer: (b)

Q3. The complexity of sorting algorithm measures the as a function of the number n of items to be sorter.

a. average time

b. running time

c. average-case complexity

d. case-complexity

Answer: (b).

Q4. The given array is arr = {2,3,4,1,6}. What are the pivots that are returned as a result of subsequent partitioning?

a. 1 and 3

b. 3 and 1

c. 2 and 6

d. 6 and 2

Answer: (a).

Q5. Which of the following code performs the partition operation in QuickSort?

```
a) private static int partition(int[] arr, int low, int high)
{
int left, right, pivot_item = arr[low];
 left = low;
 right = high;
 while(left > right)
 {
  while(arr[left] <= pivot_item)
  {
   left++;
  }
  while(arr[right] > pivot_item)
  {
   right--;
  }
  if(left < right)
  {
   swap(arr, left, right);
  }
 }
 arr[low] = arr[right];
 arr[right] = pivot_item;
 return right;
}
```

```
b) private static int partition(int[] arr, int low, int high)
{
 int left, right, pivot_item = arr[low];
 left = low;
 right = high;
 while(left <= right)
 {
  while(arr[left] <= pivot_item)
  {
   left++;
  }
  while(arr[right] > pivot_item)
  {
   right--;
  }
  if(left < right)
  {
   swap(arr, left, right);
  }
 }
 arr[low] = arr[right];
 arr[right] = pivot_item;
 return right;
}
```

```
c) private static int partition(int[] arr, int low, int high)
{
 int left, right, pivot_item = arr[low];
 left = low;
 right = high;
 while(left <= right)
 {
  while(arr[left] > pivot_item)
  {
   left++;
  }
  while(arr[right] <= pivot_item)
  {
   right--;
  }
  if(left < right)
  {
   swap(arr, left, right);
  }
 }
 arr[low] = arr[right];
 arr[right] = pivot_item;
 return right;
}
```

```
d) private static int partition(int[] arr, int low, int high)
{
 int left, right, pivot_item = arr[low];
 left = low;
 right = high;
 while(left > right)
 {
  while(arr[left] > pivot_item)
  {
   left++;
  }
  while(arr[right] <= pivot_item)
  {
   right--;
  }
  if(left < right)
  {
   swap(arr, left, right);
  }
 }
 arr[low] = arr[right];
 arr[right] = pivot_item;
 return right;
}
```

a. A
b. B
c. C
d. D
Answer: (b).

Q6. What is a randomized QuickSort?

a. The leftmost element is chosen as the pivot
b. The rightmost element is chosen as the pivot
c. Any element in the array is chosen as the pivot
d. A random number is generated which is used as the pivot
Answer: (c).

Q7. Select the appropriate recursive call for QuickSort.(arr is the array, low is the starting index and high is the ending index of the array, partition returns the pivot element)

a) public static void quickSort(int[] arr, int low, int high)

```
{
 int pivot;
 if(high>low)
 {
  pivot = partition(arr, low, high);
  quickSort(arr, low, pivot-1);
  quickSort(arr, pivot+1, high);
 }
}
```

b) public static void quickSort(int[] arr, int low, int high)

```
{
 int pivot;
 if(high<low)
 {
  pivot = partition(arr, low, high);
  quickSort(arr, low, pivot-1);
  quickSort(arr, pivot+1, high);
 }
}
```

c) public static void quickSort(int[] arr, int low, int high)

```
{
 int pivot;
 if(high>low)
 {
  pivot = partition(arr, low, high);
  quickSort(arr, low, pivot);
  quickSort(arr, pivot, high);
 }
}
```

d) None of the mentioned
a. A
b. B
c. C

d. D

Answer: (a).

Q8. The given array is arr = {1,2,4,3}. Bubble sort is used to sort the array elements. How many iterations will be done to sort the array with improvised version?

a. 4

b. 2

c. 1

d. 0

Answer: (b).

Q9. What is the best case efficiency of bubble sort in the improvised version?

a. O(nlogn)

b. O(logn)

c. O(n)

d. O(n^2)

Answer: (c).

Q10. How can you improve the best case efficiency in bubble sort? (The input is already sorted)

```
a) boolean swapped = false;
 for(int j=arr.length-1; j>=0 && swapped; j--)
 {
  swapped = true;
  for(int k=0; k<j; k++)
  {
   if(arr[k] > arr[k+1])
   {
    int temp = arr[k];
    arr[k] = arr[k+1];
    arr[k+1] = temp;
    swapped = false;
   }
  }
 }
```

```
b) boolean swapped = true;
 for(int j=arr.length-1; j>=0 && swapped; j--)
 {
  swapped = false;
  for(int k=0; k<j; k++)
  {
   if(arr[k] > arr[k+1])
   {
    int temp = arr[k];
    arr[k] = arr[k+1];
    arr[k+1] = temp;
   }
  }

}
```

```
c) boolean swapped = true;
```

```
 for(int j=arr.length-1; j>=0 && swapped; j--)
 {
  swapped = false;
  for(int k=0; k<j; k++)
  {
   if(arr[k] > arr[k+1])
   {
    int temp = arr[k];
    arr[k] = arr[k+1];
    arr[k+1] = temp;
    swapped = true;
   }
  }
 }
```

```
d) boolean swapped = true;
 for(int j=arr.length-1; j>=0 && swapped; j--)
 {
  for(int k=0; k<j; k++)
  {
   if(arr[k] > arr[k+1])
   {
    int temp = arr[k];
    arr[k] = arr[k+1];
    arr[k+1] = temp;
    swapped = true;
   }
  }
 }
```

a. A
b. B
c. C
d. D
Answer: (c).

Q11. The given array is arr = {1,2,3,4,5}. (bubble sort is implemented with a flag variable)The number of iterations in selection sort and bubble sort respectively are,

a. 5 and 4
b. 1 and 4
c. 0 and 4
d. 4 and 1
Answer: (b)

Q12. You have to sort a list L, consisting of a sorted list followed by a few 'random' elements. Which of the following sorting method would be most suitable for such a task?

a. Bubble sort
b. Selection sort
c. Quick sort
d. Insertion sort
Answer: D

Q13. The average case and worst case complexities for Merge sort algorithm are

a. O (n2), O (n2)

b. O (n2), O (n log2n)
c. O (n log2n), O (n2)
d. O (n log2n), O (n log2n)
Answer: D

Q14. Selection sort algorithm design technique is an example of
a. Greedy method
b. Divide-and-conquer
c. Dynamic Programming
d. Backtracking
Answer: A

Q15. Which of the following algorithm design technique is used in merge sort?
a. Greedy method
b. Backtracking
c. Dynamic programming
d. Divide and Conquer
Answer: D

Q16. Which one of the following in place sorting algorithms needs the minimum number of swaps?
a. Quick sort
b. Insertion sort
c. Selection sort
d. Heap sort
Answer: C

Q17. Which of the following algorithms sort n integers, having the range 0 to (n2 - 1), in ascending order in O(n) time ?
a. Selection sort
b. Bubble sort
c. Radix sort
d. Insertion sort
Answer: C

Q18. Consider the following sorting algorithms. I. Quicksort II. Heapsort III. Mergesort which of them perform in least time in the worst case?
a. I and II only
b. II and III only
c. III only
d. I, II and III
Answer: B

Q19. Which of the following is true for computation time in insertion, deletion and finding maximum and minimum element in a sorted array?
a. Insertion – 0(1), Deletion – 0(1), Maximum – 0(1), Minimum – 0(l)
b. Insertion – 0(1), Deletion – 0(1), Maximum – 0(n), Minimum – 0(n)
c. Insertion – 0(n), Deletion – 0(n), Maximum – 0(1), Minimum – 0(1)
d. Insertion – 0(n), Deletion – 0(n), Maximum – 0(n), Minimum – 0(n)
Answer: C

Q20. The auxiliary space of insertion sort is O(1), what does O(1) mean ?
a. The memory (space) required to process the data is not constant.
b. It means the amount of extra memory Insertion Sort consumes doesn't depend on the input. The algorithm should use the same amount of memory for all inputs.
c. It takes only 1 kb of memory.

d. It is the speed at which the elements are traversed.
Answer: B

Q1. What is the best sorting algorithm to use for the elements in array are more than 1 million in general?
a. Merge sort.
b. Bubble sort.
c. Quick sort.
d. Insertion sort.
Answer: C

Q2. Which of the below given sorting techniques has highest best-case runtime complexity.
a. Quick sort
b. Selection sort
c. Insertion sort
d. Bubble sort
Answer: B

Q3. A sorting technique is called stable if:
a. It takes O(nlog n)time
b. It maintains the relative order of occurrence of non-distinct elements
c. It uses divide and conquer paradigm
d. It takes O(n) space
Answer: B

Q4. If one uses straight two-way merge sort algorithm to sort the following elements in ascending order 20, 47, 15, 8, 9, 4, 40, 30, 12, 17 then the order of these elements after the second pass of the algorithm is:
a. 8, 9, 15, 20, 47, 4, 12, 17, 30, 40
b. 8, 15, 20, 47, 4, 9, 30, 40, 12, 17
c. 15, 20, 47, 4, 8, 9, 12, 30, 40, 17
d. 4, 8, 9, 15, 20, 47, 12, 17, 30, 40
Answer: B

Q5. Quicksort is run on two inputs shown below to sort in ascending order taking first element as pivot,
(i) 1, 2, 3,......., n (ii) n, n-1, n-2,......, 2, 1
Let C1 and C2 be the number of comparisons made for the inputs (i) and (ii) respectively. Then,
a. C1 < C2
b. C1 > C2
c. C1 = C2
d. We cannot say anything for arbitrary n
Answer: C

Q6. You are given a sequence of n elements to sort. The input sequence consists of n/k subsequences, each containing k elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length n is to sort the k elements in each of the n/k subsequences. The lower bound on the number of comparisons needed to solve this variant of the sorting problem is
a. Ω (n)
b. Ω (n/k)
c. Ω (nlogk)
d. Ω (n/klogn/k)
Answer: C

Q7. Which one of the following is the recurrence equation for the worst case time complexity of the Quicksort algorithm for sorting n(≥ 2) numbers? In the recurrence equations given in the options below, c is a constant.

a. T(n) = 2T (n/2) + cn
b. T(n) = T(n – 1) + T(0) + cn
c. T(n) = 2T (n – 2) + cn
d. T(n) = T(n/2) + cn
Answer: B

Q8. Assume that a mergesort algorithm in the worst case takes 30 seconds for an input of size 64. Which of the following most closely approximates the maximum input size of a problem that can be solved in 6 minutes?

a. 256
b. 512
c. 1024
d. 2048
Answer: B

Q9. The worst case running times of Insertion sort, Merge sort and Quick sort, respectively, are:

a. Θ(n log n), Θ(n log n) and Θ(n2)
b. Θ(n2), Θ(n2) and Θ(n Log n)
c. Θ(n2), Θ(n log n) and Θ(n log n)
d. Θ(n2), Θ(n log n) and Θ(n2)
Answer: D

Q10. Assume that the algorithms considered here sort the input sequences in ascending order. If the input is already in ascending order, which of the following are TRUE ?

I. Quicksort runs in Θ(n2) time
II. Bubblesort runs in Θ(n2) time
III. Mergesort runs in Θ(n) time
IV. Insertion sort runs in Θ(n) time

a. I and II only
b. I and III only
c. II and IV only
d. I and IV only
Answer: D

Q11. Assume that we use Bubble Sort to sort n distinct elements in ascending order. When does the best case of Bubble Sort occur?

a. When elements are sorted in ascending order
b. When elements are sorted in descending order
c. When elements are not sorted by any order
d. There is no best case for Bubble Sort. It always takes O(n*n) time
Answer: A

Q12. If we use Radix Sort to sort n integers in the range (nk/2,nk], for some k>0 which is independent of n, the time taken would be?

a. Θ(n)
b. Θ(kn)
c. Θ(nlogn)
d. Θ(n2)
Answer: C

Q13. Consider an array of elements arr[5]= {5,4,3,2,1} , what are the steps of insertions done while doing insertion sort in the array.

a. 4 5 3 2 1 3 4 5 2 1 2 3 4 5 1 1 2 3 4 5
b. 5 4 3 1 2 5 4 1 2 3 5 1 2 3 4 1 2 3 4 5
c. 4 3 2 1 5 3 2 1 5 4 2 1 5 4 3 1 5 4 3 2
d. 4 5 3 2 1 2 3 4 5 1 3 4 5 2 1 1 2 3 4 5
Answer: A

Q14. Which is the correct order of the following algorithms with respect to their time Complexity in the best case ?

a. Merge sort > Quick sort >Insertion sort > selection sort
b. insertion sort < Quick sort < Merge sort < selection sort
c. Merge sort > selection sort > quick sort > insertion sort
d. Merge sort > Quick sort > selection sort > insertion sort
Answer: B

Q15. Which of the following statements is correct with respect to insertion sort ?
***Online - can sort a list at runtime**
***Stable - doesn't change the relative order of elements with equal keys.**

a. Insertion sort is stable, online but not suited well for large number of elements.
b. Insertion sort is unstable and online
c. Insertion sort is online and can be applied to more than 100 elements
d. Insertion sort is stable & online and can be applied to more than 100 elements
Answer: A

Q16. Consider the array A[]= {6,4,8,1,3} apply the insertion sort to sort the array . Consider the cost associated with each sort is 25 rupees, what is the total cost of the insertion sort when element 1 reaches the first position of the array ?

a. 50
b. 25
c. 75
d. 100
Answer: A

Q17. The number of elements that can be sorted in time using heap sort is

a. $\Theta(1)$ b. $\Theta(\sqrt{logn})$

c. $\Theta(Logn/(LogLogn))$ d. $\Theta(Logn)$

Answer: C

Q18. Which of the following is true about merge sort?

a. Merge Sort works better than quick sort if data is accessed from slow sequential memory.
b. Merge Sort is stable sort by nature
c. Merge sort outperforms heap sort in most of the practical situations.
d. All of the above.
Answer: D

Q19. Given an array where numbers are in range from 1 to n6, which sorting algorithm can be used to sort these number in linear time?

a. Not possible to sort in linear time
b. Radix Sort
c. Counting Sort
d. Quick Sort
Answer: B

Q20. In quick sort, for sorting n elements, the (n/4)th smallest element is selected as pivot using an O(n) time algorithm. What is the worst case time complexity of the quick sort?

a. (n)
b. (nLogn)
c. (n^2)
d. (n^2 log n)
Answer: B

ADVANCE LEVEL QUIZ
TOPIC: SEARCHING

Q21. Consider the Quicksort algorithm. Suppose there is a procedure for finding a pivot element which splits the list into two sub-lists each of which contains at least one-fifth of the elements. Let T(n) be the number of comparisons required to sort n elements. Then
a. T(n) <= 2T(n/5) + n
b. T(n) <= T(n/5) + T(4n/5) + n
c. T(n) <= 2T(4n/5) + n
d. T(n) <= 2T(n/2) + n
Answer: B
Q22. Let P be a QuickSort Program to sort numbers in ascending order using the first element as pivot. Let t1 and t2 be the number of comparisons made by P for the inputs {1, 2, 3, 4, 5} and {4, 1, 5, 3, 2} respectively. Which one of the following holds?
a. t1 = 5
b. t1 < t2
c. t1 > t2
d. t1 = t2
Answer: C
Q23. You have an array of n elements. Suppose you implement quicksort by always choosing the central element of the array as the pivot. Then the tightest upper bound for the worst case performance is
a. O(n2)
b. O(nLogn)
c. Theta(nLogn)
d. O(n3)
Answer: A
Q24. In a permutation a1.....an of n distinct integers, an inversion is a pair (ai, aj) such that i < j and ai > aj. What would be the worst case time complexity of the Insertion Sort algorithm, if the inputs are restricted to permutations of 1.....n with at most n inversions?
a. Θ (n2)
b. Θ (n log n)
c. Θ (n1.5)
d. Θ (n)
Answer: D
Q25. Randomized quicksort is an extension of quicksort where the pivot is chosen randomly. What is the worst case complexity of sorting n numbers using randomized quicksort?
a. O(n)
b. O(n Log n)
c. O(n2)
d. O(n!)
Answer: C
Q26. Which of the following changes to typical QuickSort improves its performance on average and are generally done in practice.

1) Randomly picking up to make worst case less likely to occur.
2) Calling insertion sort for small sized arrays to reduce recursive calls.
3) QuickSort is tail recursive, so tail call optimizations can be done.
4) A linear time median searching algorithm is used to pick the median, so that the worst case time reduces to O(nLogn)
a. 1 and 2
b. 2, 3, and 4
c. 1, 2 and 3
d. 2, 3 and 4
Answer: C

Q27. You have to sort 1 GB of data with only 100 MB of available main memory. Which sorting technique will be most appropriate?
a. Heap sort
b. Merge sort
c. Quick sort
d. Insertion sort
Answer: B

Q28. What is the worst case time complexity of insertion sort where position of the data to be inserted is calculated using binary search?
a. N
b. NlogN
c. N^2
d. N(logN)^2
Answer: C

Q29. The tightest lower bound on the number of comparisons, in the worst case, for comparison-based sorting is of the order of
a. N
b. N^2
c. NlogN
d. N(logN)^2
Answer: C

Q30. In a modified merge sort, the input array is splitted at a position one-third of the length(N) of the array. Which of the following is the tightest upper bound on time complexity of this modified Merge Sort.
a. N(logN base 3)
b. N(logN base 2/3)
c. N(logN base 1/3)
d. N(logN base 3/2)
Answer: D

Q31. Which sorting algorithm will take least time when all elements of input array are identical? Consider typical implementations of sorting algorithms.
a. Insertion Sort
b. Heap Sort
c. Merge Sort
d. Selection Sort
Answer: A

Q32. list of n string, each of length n, is sorted into lexicographic order using the merge-sort algorithm. The worst case running time of this computation is
a. O (n log n)
b. O (n2 log n)

c. O (n2 + log n)
d. O (n2)
Answer: B

Q33. In quick sort, for sorting n elements, the (n/4)th smallest element is selected as pivot using an O(n) time algorithm. What is the worst case time complexity of the quick sort?
a. (n)
b. (nLogn)
c. (n^2)
d. (n^2 log n)
Answer: B

Q34. Consider the Quicksort algorithm. Suppose there is a procedure for finding a pivot element which splits the list into two sub-lists each of which contains at least one-fifth of the elements. Let T(n) be the number of comparisons required to sort n elements. Then
a. T(n) <= 2T(n/5) + n
b. T(n) <= T(n/5) + T(4n/5) + n
c. T(n) <= 2T(4n/5) + n
d. T(n) <= 2T(n/2) + n
Answer: B

Q35. Which of the following sorting algorithms has the lowest worst-case complexity?
a. Merge Sort
b. Bubble Sort
c. Quick Sort
d. Selection Sort
Answer: A

Q36. Which sorting algorithms is most efficient to sort string consisting of ASCII characters?
a. Quick sort
b. Heap sort
c. Merge sort
d. Counting sort
Answer: D

Q37. Suppose we are sorting an array of eight integers using heapsort, and we have just finished some heapify (either maxheapify or minheapify) operations. The array now looks like this: 16 14 15 10 12 27 28 How many heapify operations have been performed on root of heap?
a. 1
b. 2
c. 3 or 4
d. 5 or 6
Answer: B

Q38. Suppose we are sorting an array of eight integers using quicksort, and we have just finished the first partitioning with the array looking like this:
2 5 1 7 9 12 11 10
Which statement is correct?
a. The pivot could be either the 7 or the 9.
b. The pivot could be the 7, but it is not the 9
c. The pivot is not the 7, but it could be the 9
d. Neither the 7 nor the 9 is the pivot.
ANSWER:A

Q39. Which of the following is not true about comparison based sorting algorithms?
a. The minimum possible time complexity of a comparison based sorting algorithm is O(nLogn) for a random input array
b. Any comparison based sorting algorithm can be made stable by using position as a criteria when two elements are compared
c. Counting Sort is not a comparison based sorting algortihm
d. Heap Sort is not a comparison based sorting algorithm.
ANSWER:D
Q40. Consider a situation where swap operation is very costly. Which of the following sorting algorithms should be preferred so that the number of swap operations are minimized in general?
a. Heap Sort
b. Selection Sort
c. Insertion Sort
d. Merge Sort
ANSWER:B
Q41. Given an unsorted array. The array has this property that every element in array is at most k distance from its position in sorted array where k is a positive integer smaller than size of array. Which sorting algorithm can be easily modified for sorting this array and what is the obtainable time complexity?
a. Insertion Sort with time complexity O(kn)
b. Heap Sort with time complexity O(nLogk)
c. Quick Sort with time complexity O(kLogk)
d. Merge Sort with time complexity O(kLogk)
ANSWER:B
Q42. Which of the following sorting algorithms in its typical implementation gives best performance when applied on an array which is sorted or almost sorted (maximum 1 or two elements are misplaced).
a. Quick Sort
b. Heap Sort
c. Merge Sort
d. Insertion Sort
ANSWER:D

SEARCHING

1. What do you understand by a binary search? What is the best scenario of using it?
Answer: A binary search is an algorithm that starts with searching in the middle element. If the middle element is not the target element then it further checks whether to continue searching the lower half of the higher half. The process continues until the target element is found. The binary search works best when applied to a list with sorted or ordered elements.

2. What are the advantages of Binary search over linear search?

There are relatively less number of comparisons in binary search than that in linear search. In average case, linear search takes O(n) time to search a list of n elements while Binary search takes O(log n) time to search a list of n elements.

3. Can Binary Search be used for linked lists?
Since random access is not allowed in linked list, we cannot reach the middle element in O(1) time. Therefore Binary Search is not possible for linked lists. There are other ways though, refer Skip List for example.

4. What are the types of searching used in Data Structures?

The two primary methods of searching are linear search and binary search.

Linear search involves iterating over a data unit in order to perform the required operation.

Binary search is more efficient in a way that it has the ability to split the data unit into chunks and then perform a search operation.

5. Can you explain the interpolation search technique?

The interpolation search technique is an enhanced variant of binary search. It works on the probing position of the required value.

SORTING

1. Could you explain how does the selection sort work on an array?

The selection sort begins with finding the smallest element. It is switched with the element present at subscript 0. Next, the smallest element in the remaining subarray is located and switched with the element residing in the subscript 1.

The aforementioned process is repeated until the biggest element is placed at the subscript n-1, where n represents the size of the given array.

2. How does insertion sort differ from selection sort?

Both insertion and selection approaches maintain two sub-lists, sorted and unsorted. Each takes one element from the unsorted sub-list and place it into the sorted sub-list. The distinction between the two sorting processes lies in the treatment of the current element. Insertion sort takes the current element and places it in the sorted sublist at the appropriate location. Selection sort, on the other hand, searches for the minimum value in the unsorted sub-list and replaces the same with the present element.

3. What do you understand by shell sort?

The shell sort can be understood as a variant of the insertion sort. The approach divides the entire list into smaller sub-lists based on some gap variable. Each sub-list is then sorted using insertion sort.

4. How does bubble sort work?

Bubble sort is one of the most used sorting techniques out there. It is applied to arrays where elements adjacent to each other are compared and values are exchanged based on the order of arrangement. It’s called bubble sort because of the concept of exchanging elements like a bubble floating to the top of the water and larger entities sinking down to the bottom end.

5. **What is merge sort?**

Merge sort is a method of sorting, which is based on the divide and conquer technique. Here, data entities adjacent to each other are first merged and sorted in every iteration to create sorted lists. These smaller sorted lists are combined at the end to form the completely sorted list.

6. What are the advantages of Selection Sort?

It is simple and easy to implement.

It can be used for small data sets.

It is 60 per cent more efficient than bubble sort.

7. Which sorting algorithm is considered the fastest? Why?

A single sorting algorithm can’t be considered best, as each algorithm is designed for a particular data structure and data set. However, the QuickSort algorithm is generally considered the fastest because it has the best performance for most inputs.Its advantages over other sorting algorithms include the following:

- Cache-efficient: It linearly scans and linearly partitions the input. This means we can make the most of every cache load.
- Can skip some swaps: As QuickSort is slightly sensitive to input that is in the right order, it can skip some swaps.
- Efficient even in worst-case input sets, as the order is generally random.
- Easy adaption to already- or mostly-sorted inputs.
- When speed takes priority over stability.

CHAPTER-9

Hashing

- Hashing is the process of mapping large amount of data item to smaller table with the help of hashing function.
- Hashing is also known as **Hashing Algorithm** or **Message Digest Function**.
- It is a technique to convert a range of key values into a range of indexes of an array.
- It is used to facilitate the next level searching method when compared with the linear or binary search.
- Hashing allows to update and retrieve any data entry in a constant time O(1).
- Constant time O(1) means the operation does not depend on the size of the data.
- Hashing is used with a database to enable items to be retrieved more quickly.
- It is used in the encryption and decryption of digital signatures.

What is Hash Function?

- A fixed process converts a key to a hash key is known as a **Hash Function.**
- This function takes a key and maps it to a value of a certain length which is called a **Hash value** or **Hash.**
- Hash value represents the original string of characters, but it is normally smaller than the original.

- It transfers the digital signature and then both hash value and signature are sent to the receiver. Receiver uses the same hash function to generate the hash value and then compares it to that received with the message.
- If the hash values are same, the message is transmitted without errors.

What is Hash Table?

- Hash table or hash map is a data structure used to store key-value pairs.
- It is a collection of items stored to make it easy to find them later.
- It uses a hash function to compute an index into an array of buckets or slots from which the desired value can be found.
- It is an array of list where each list is known as bucket.
- It contains value based on the key.
- Hash table is used to implement the map interface and extends Dictionary class.
- Hash table is synchronized and contains only unique elements.

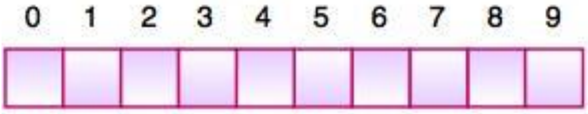


Fig. Hash Table

- The above figure shows the hash table with the size of n = 10. Each position of the hash table is called as **Slot**. In the above hash table, there are n slots in the table, names = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}. Slot 0, slot 1, slot 2 and so on. Hash table contains no items, so every slot is empty.
- As we know the mapping between an item and the slot where item belongs in the hash table is called the hash function. The hash function takes any item in the collection and returns an integer in the range of slot names between 0 to n-1.
- Suppose we have integer items {26, 70, 18, 31, 54, 93}. One common method of determining a hash key is the division method of hashing and the formula is :

Hash Key = Key Value % Number of Slots in the Table

- Division method or reminder method takes an item and divides it by the table size and returns the remainder as its hash value.

Data Item	Value % No. of Slots	Hash Value
26	26 % 10 = 6	6
70	70 % 10 = 0	0
18	18 % 10 = 8	8
31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3

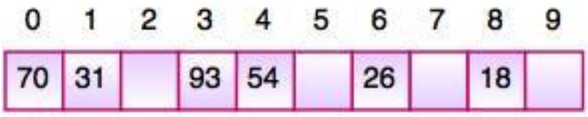


Fig. Hash Table

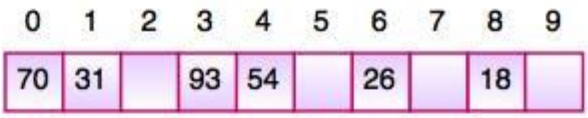
- After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure. In the hash table, 6 of the 10 slots are occupied, it is referred to as the load factor and denoted by, λ = No. of items / table size. For example , λ = 6/10.
- It is easy to search for an item using hash function where it computes the slot name for the item and then checks the hash table to see if it is present.
- Constant amount of time O(1) is required to compute the hash value and index of the hash table at that location.

Linear Probing

- Take the above example, if we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem. This problem is called as **Collision** or **Clash**. Collision creates a problem for hashing technique.
- **Linear probing is used for resolving the collisions in hash table**, data structures for maintaining a collection of key-value pairs.
- Linear probing was invented by Gene Amdahl, Elaine M. McGraw and Arthur Samuel in 1954 and analyzed by Donald Knuth in 1963.
- It is a component of open addressing scheme for using a hash table to solve the dictionary problem.
- The simplest method is called Linear Probing. Formula to compute linear probing is:

P = (1 + P) % (MOD) Table_size

For example,



0	1	2	3	4	5	6	7	8	9
70	31		93	54		26		18	

Fig. Hash Table

If we insert next item 40 in our collection, it would have a hash value of 0 (40 % 10 = 0). But 70 also had a hash value of 0, it becomes a problem.

Linear probing solves this problem:

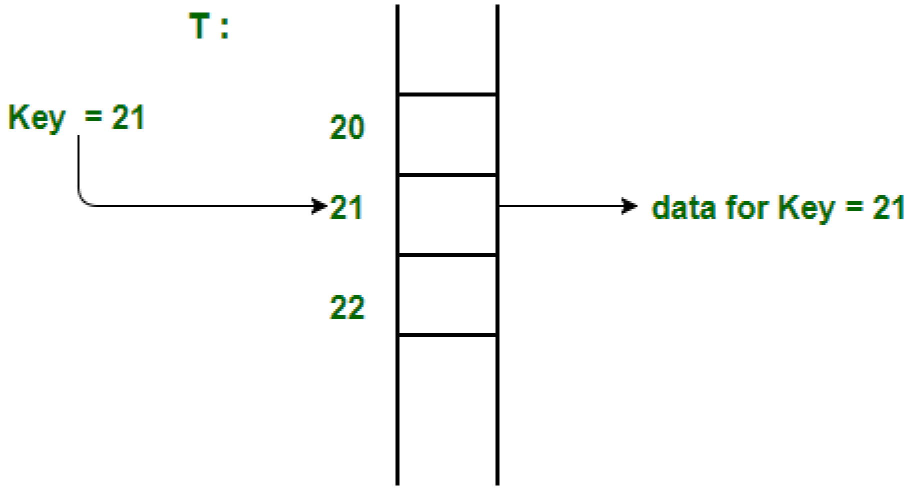
P = H(40)
44 % 10 = **0**
Position 0 is occupied by 70. so we look elsewhere for a position to store 40.

Using Linear Probing:
P= (P + 1) % table-size
0 + 1 % 10 = **1**
But, position 1 is occupied by 31, so we look elsewhere for a position to store 40.

Using linear probing, we try next position : 1 + 1 % 10 = 2
Position 2 is empty, so 40 is inserted there.

Direct Address Table

Direct Address Table is a data structure that has the capability of mapping records to their corresponding keys using arrays. In direct address tables, records are placed using their key values directly as indexes. They facilitate fast searching, insertion and deletion operations.
We can understand the concept using the following example. We create an array of size equal to maximum value plus one (assuming 0 based index) and then use values as indexes. For example, in the following diagram key 21 is used directly as index.



Advantages:

- **Searching in O(1) Time**: Direct address tables use arrays which are random access data structure, so, the key values (which are also the index of the array) can be easily used to search the records in O(1) time.
- **Insertion in O(1) Time**: We can easily insert an element in an array in O(1) time. The same thing follows in a direct address table also.
- **Deletion in O(1) Time:** Deletion of an element takes O(1) time in an array. Similarly, to delete an element in a direct address table we need O(1) time.

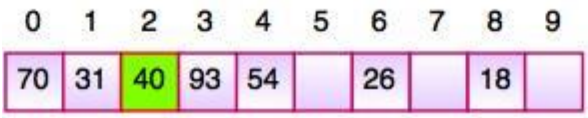
Limitations:

- Prior knowledge of maximum key value
- Practically useful only if the maximum value is very less.
- It causes wastage of memory space if there is a significant difference between total records and maximum value.

Hashing can overcome these limitations of direct address tables.

How to handle collisions?

Collisions can be handled like Hashing. We can either use Chaining or open addressing to handle collisions. The only difference from hashing here is, we do not use a hash function to find the index. We rather directly use values as indexes.



0	1	2	3	4	5	6	7	8	9
70	31	40	93	54		26		18	

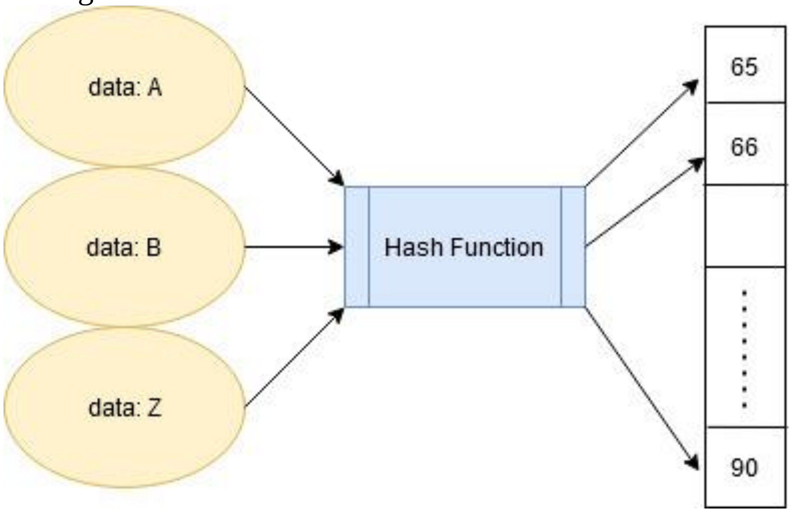
Fig. Hash Table

collision resolution techniques

Hashing is the process of transforming data and mapping it to a range of values which can be efficiently looked up.
different collision resolution techniques such as:

- Open Hashing (Separate chaining)
- Closed Hashing (Open Addressing)

- Liner Probing
- Quadratic probing
- Double hashing



- ***Hash table*****:** a data structure where the data is stored based upon its hashed key which is obtained using a hashing function.
- ***Hash function*****:** a function which for a given data, outputs a value mapped to a fixed range. A hash table leverages the hash function to efficiently map data such that it can be retrieved and updated quickly. Simply put, assume S = {s1, s2, s3, ..., sn} to be a set of objects that we wish to store into a map of size N, so we use a hash function H, such that for all s belonging to S; H(s) -> x, where x is guaranteed to lie in the range [1,N]
- ***Perfect Hash function***: a hash function that maps each item into a unique slot (no collisions).

Hash Collisions: As per the Pigeonhole principle if the set of objects we intend to store within our hash table is larger than the size of our hash table we are bound to have two or more different objects having the same hash value; a hash collision. Even if the size of the hash table is large enough to accommodate all the objects finding a hash function which generates a unique hash for each object in the hash table is a difficult task. Collisions are bound to occur (unless we find a perfect hash function, which in most of the cases is hard to find) but can be significantly reduced with the help of various collision resolution techniques.

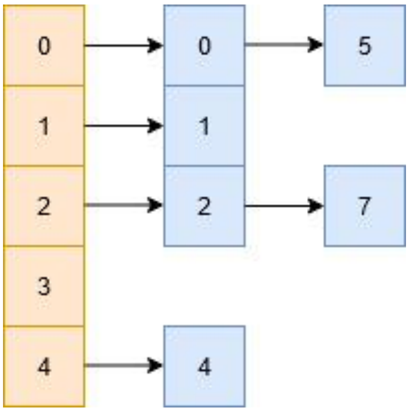
1. Open Hashing (Separate chaining)

Collisions are resolved using a list of elements to store objects with the same key together.

- Suppose you wish to store a set of numbers = {0,1,2,4,5,7} into a hash table of size 5.
- Now, assume that we have a hash function H, such that H(x) = x%5
- So, if we were to map the given data with the given hash function we'll get the corresponding values

  H(0)-> 0%5 = 0
  H(1)-> 1%5 = 1
  H(2)-> 2%5 = 2
  H(4)-> 4%5 = 4
  H(5)-> 5%5 = 0
  H(7)-> 7%5 = 2

- Clearly 0 and 5, as well as 2 and 7 will have the same hash value, and in this case we'll simply append the colliding values to a list being pointed by their hash keys.



- Obviously in practice the table size can be significantly large and the hash function can be even more complex, also the data being hashed would be more complex and non-primitive, but the idea remains the same.
  This is an easy way to implement hashing but it has its own demerits.
- The lookups/inserts/updates can become linear [O(N)] instead of constant time [O(1)] if the hash function has too many collisions.
- It doesn't account for any empty slots which can be leveraged for more efficient storage and lookups.
- Ideally we require a good hash function to guarantee even distribution of the values.
- Say, for a **load factor**
  λ=number of objects stored in table/size of the table (can be >1)
  a good hash function would guarantee that the maximum length of list associated with each key is close to the load factor.

Note that the order in which the data is stored in the lists (or any other data structures) is based upon the implementation requirements. Some general ways include insertion order, frequency of access etc.

2. Closed Hashing (Open Addressing)

This collision resolution technique requires a hash table with fixed and known size. During insertion, if a collision is encountered, alternative cells are tried until an empty bucket is found. These techniques require the size of the hash table to be supposedly larger than the number of objects to be stored (something with a load factor < 1 is ideal). There are various methods to find these empty buckets:

a. Liner Probing
b. Quadratic probing
c. Double hashing

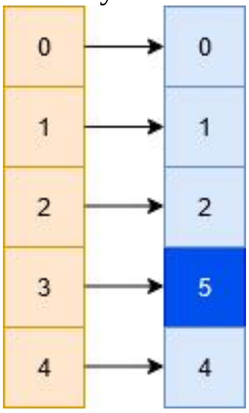
a. Linear Probing

The idea of linear probing is simple, we take a fixed sized hash table and every time we face a hash collision we linearly traverse the table in a cyclic manner to find the next empty slot.

- Assume a scenario where we intend to store the following set of numbers = {0,1,2,4,5,7} into a hash table of size 5 with the help of the following hash function H, such that H(x) = x%5.
- So, if we were to map the given data with the given hash function we'll get the corresponding values

H(0)-> 0%5 = 0
H(1)-> 1%5 = 1
H(2)-> 2%5 = 2
H(4)-> 4%5 = 4
H(5)-> 5%5 = 0

- in this case we see a collision of two terms (0 & 5). In this situation we move linearly down the table to find the first empty slot. Note that this linear traversal is cyclic in nature, i.e. in the event we exhaust the last element during the search we start again from the beginning until the initial key is reached.



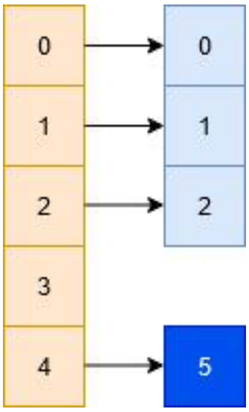
In this case our hash function can be considered as this:
H(x, i) = (H(x) + i)%N
where N is the size of the table and i represents the linearly increasing variable which starts from 1 (until empty bucket is found).

Despite being easy to compute, implement and deliver best cache performance, this suffers from the problem of clustering (many consecutive elements get grouped together, which eventually reduces the efficiency of finding elements or empty buckets).

b. Quadratic Probing

This method lies in the middle of great cache performance and the problem of clustering. The general idea remains the same, the only difference is that we look at the Q(i) increment at each iteration when looking for an empty bucket, where Q(i) is some quadratic expression of i. A simple expression of Q would be Q(i) = i^2, in which case the hash function looks something like this:
H(x, i) = (H(x) + i^2)%N

- In general, H(x, i) = (H(x) + ((c1*i^2 + c2*i + c3)))%N, for some choice of constants c1, c2, and c3
- Despite resolving the problem of clustering significantly it may be the case that in some situations this technique does not find any available bucket, unlike linear probing which always finds an empty bucket.
- Luckily, we can get good results from quadratic probing with the right combination of probing function and hash table size which will guarantee that we will visit as many slots in the table as possible. In particular, if the hash table's size is a prime number and the probing function is H(x, i) = i^2, then at least 50% of the slots in the table will be visited. Thus, if the table is less than half full, we can be certain that a free slot will eventually be found.
- Alternatively, if the hash table size is a power of two and the probing function is H(x, i) = (i^2 + i)/2, then every slot in the table will be visited by the probing function.
- Assume a scenario where we intend to store the following set of numbers = {0,1,2,5} into a hash table of size 5 with the help of the following hash function H, such that H(x, i) = (x%5 + i^2)%5.

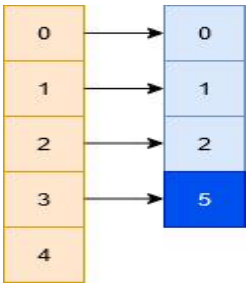


Clearly 5 and 0 will face a collision, in which case we'll do the following:
- we look at 5%5 = 0 (collision)
- we look at (5%5 + 1^2)%5 = 1 (collision)
- we look at (5%5 + 2^2)%5 = 4 (empty -> place element here)

c. Double Hashing

This method is based upon the idea that in the event of a collision we use an another hashing function with the key value as an input to find where in the open addressing scheme the data should actually be placed at.

- In this case we use two hashing functions, such that the final hashing function looks like:
  H(x, i) = (H1(x) + i*H2(x))%N
- Typically for H1(x) = x%N a good H2 is H2(x) = P - (x%P), where P is a prime number smaller than N.
- A good H2 is a function which never evaluates to zero and ensures that all the cells of a table are effectively traversed.
- Assume a scenario where we intend to store the following set of numbers = {0,1,2,5} into a hash table of size 5 with the help of the following hash function H, such that H(x, i) = (H1(x) + i*H2(x))%5
  H1(x) = x%5 and H2(x) = P - (x%P), where P = 3
  (3 is a prime smaller than 5)



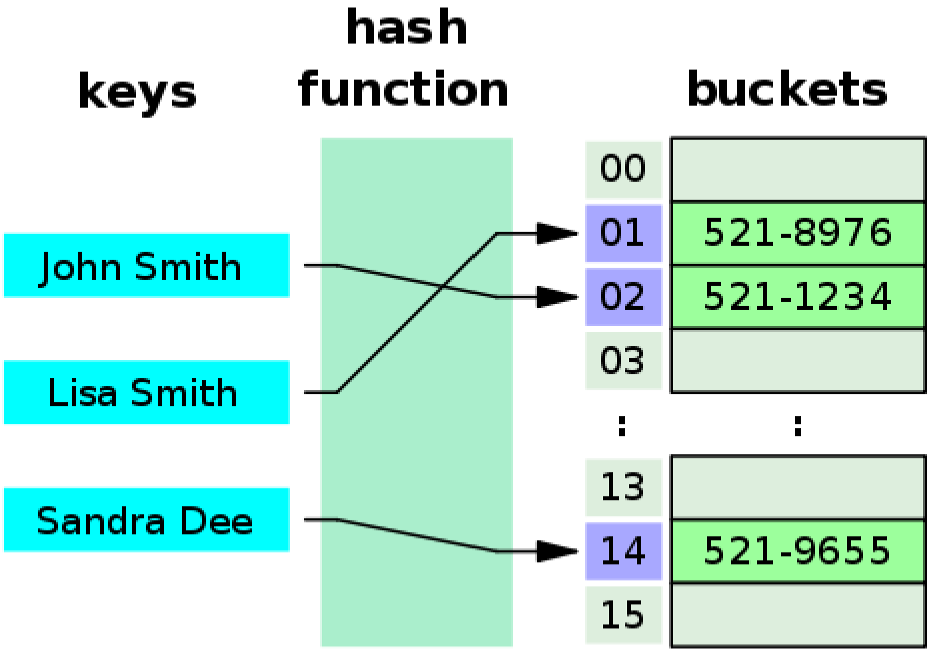
Clearly 5 and 0 will face a collision, in which case we'll do the following:
- we look at 5%5 = 0 (collision)
- we look at (5%5 + 1*(3 - (5%3)))%5 = 1 (collision)
- we look at (5%5 + 2*(3 - (5%3)))%5 = 2 (collision)
- we look at (5%5 + 3*(3 - (5%3)))%5 = 3 (empty -> place element here)

Question and Answer

1.What is Hash Table?

A hash table (hash map) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. Hash tables implement an associative array, which is indexed by arbitrary objects (keys). A hash table uses a hash function to

compute an index, also called a hash value, into an array of buckets or slots, from which the desired value can be found.



2.What is Hashing?

Hashing is the practice of using **an algorithm (or hash function)** to map data of *any* size to a *fixed* length. This is called a hash value (or sometimes hash code or hash sums or even a hash digest if you're feeling fancy). In hashing, keys are converted into *hash values* or *indexes* by using **hash functions**. Hashing is a one-way function.

3.Explain what is Hash Value?

A Hash Value (also called as Hashes or Checksum) is a string value (of specific length), which is the result of calculation of a Hashing Algorithm. Hash Values have different uses:

Indexing for Hash Tables

Determine the Integrity of any Data (which can be a file, folder, email, attachments, downloads etc).

4. What is the space complexity of a Hash Table?

The space complexity of a datastructure indicates how much space it occupies in relation to the amount of elements it holds. For example a space complexity of O(1) would mean that the datastructure alway consumes constant space no matter how many elements you put in there. O(n) would mean that the space consumption grows linearly with the amount of elements in it.

A hashtable typically has a space complexity of O(n).

5.Define what is a Hash Function?

A hash function is any function that can be used to map data of arbitrary size to fixed-size values. The values returned by a hash function are called *hash values*, hash codes, digests, or simply hashes. The values are used to index a fixed-size table called a hash table. Use of a hash function to index a hash table is called *hashing* or *scatter storage addressing*.

Mathematically speaking, a hash function is usually defined as a mapping from the universe of elements you want to store in the hash table to the range {0, 1, 2, .., numBuckets - 1}.

Some properties of Hash Functions are:

Very fast to compute (nearly constant)

One way; can not be reversed

Output does not reveal information on input

Hard to find collisions (different data with same hash)

Implementation is based on parity-preserving bit operations (XOR and ADD), multiply, or divide.

6: What is the difference between Hashing and Hash Tables?

Hashing is simply the act of turning a data chunk of arbitrary length into a fixed-width value (hereinafter called a hash value) that can be used to represent that chunk in situations where dealing with the original data chunk would be inconvenient.

A hash table is one of those situations; it simply stores references to such data chunks in a table indexed by each chunk's hash value. This way, instead of potentially comparing your desired key chunk against a huge number of chunks, you simply compute the hash value of the key chunk and do a much faster lookup with that hash value.

7: Provide a simple example of Hash Function

A one-way function is not just a hash function - a function that loses information - but a function f for which, given an image y, it is difficult to find a pre-image x such that f(x)=y.

A very simple example of a hash function that does not use any advanced math, is this simple parity function:

```
def hash(n: Nat)
  if n.even?
    0
  else
    1
  end
end
```

As you can see, it maps a large input space (the natural numbers) into a small output space (the set {0, 1}). And it is one-way: if I tell you that the result is 1, you can't tell me what the input was.

This is why they are called one-way: you can compute an image but you can't find a pre-image for a given image.

8. Detect if a List is Cyclic using Hash Table

To detect if a list is cyclic, we can check whether a node had been visited before. A natural way is to use a hash table.

Algorithm

We go through each node one by one and record each node's reference (or memory address) in a hash table. If the current node is null, we have reached the end of the list and it must not be cyclic. If current node’s reference is in the hash table, then return true.

9.Why is a hash table O 1?

Generally **O**(**1**)

would mean that our **hash table** has a complexity of **O**(n) . This is why, in general, computational complexity has 3 measures: best, average and worst-case scenarios. **Hash tables** have a **O**(**1**) complexity in best and average case scenarios, but fall to **O**(n) in their worst-case scenario.

10.What is the time complexity of HashMap?

HashMap has **complexity** of O(1) for insertion and lookup. **HashMap** allows one null key and multiple null values. **HashMap** does not maintain any order.

11.What is time complexity for hashing search technique?

Hashing is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With **hashing** we get O(1) **search time** on average (under reasonable assumptions) and O(n) in worst case

12.Why hash table is fast?

They're **faster** when searching for an element because of the way element lookups work in each. In an array, if it's unsorted you'll have to do a linear search, or if it is sorted, the best you can do is a binary search

13.How do I choose a hash table size?

But a good general "rule of thumb" is:

The **hash table** should be an array with **length** about 1.3 times the maximum number of keys that will actually be in the **table**, and.

Size of **hash table** array should be a prime number.

14.How do you rehash a hash table?

For **Rehash**, make a new array of double the previous size and make it the new bucketarray. Then traverse to each element in the old bucketArray and call the insert() for each so as to insert it into the new larger bucket array.

15.What are the different types of hashing techniques?

There are mainly two types of SQL hashing methods:

Static **Hashing**.

Dynamic **Hashing**.

16.What is the purpose of hash table?

In computing, a **hash table** (**hash** map) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A **hash table uses** a **hash function** to compute an index, also called a **hash** code, into an array of buckets or slots, from which the desired value can be found.

17.What is clustering in a hash table?

Primary **clustering** is the tendency for a collision resolution scheme such as linear probing to create long runs of filled slots near the **hash** position of keys. If the primary **hash** index is x , subsequent probes go to x+1 , x+2 , x+3 and so on, this results in Primary **Clustering**.

18.How hash table is used in database?

Hash Table is a data structure which stores data in an associative manner. In a **hash table**, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

19.What is hash table with example?

A **hash table** is a special collection that is used to store key-value items. So instead of storing just one value like the stack, array list and queue, the **hash table** stores 2 values. These 2 values form an element of the **hash table**. Below are some **example** of how values of a **hash table** might look like.

20.What is an ordered hash table?

Ordered hash tables are new in PowerShell 3.0 and great for creating new objects. Unlike regular **hash tables**, **ordered hash tables** keep the **order** in which you add keys, so you can control in which **order** these keys turn into object properties.

21.What is a collision in a hash function?

Definition: A **collision** occurs when more than one value to be hashed by a particular **hash function hash** to the same slot in the table or data structure (**hash** table) being generated by the **hash function**.

22.How does hash collision occur?

A **collision occurs** when a **hash** function returns the same bucket location for two different keys. A **collision** will **occur** when two different keys have the same hashCode, which can happen because two unequal objects in Java can have the same hashCode.

23.What is a hash collision attack?

In cryptography, a **collision attack** on a cryptographic **hash** tries to find two inputs producing the same **hash** value, i.e. a **hash collision**. This is in contrast to a preimage **attack** where a specific target **hash** value is specified. ... **Collision attack**. Find two different messages m1 and m2 such that **hash**(m1) = **hash**(m2).

24.How can hash collision be prevented?

A means for avoiding **hash collisions** by means of message pre-processing function to increase randomness and reduce redundancy of an input message whereby **hash collisions** are avoided when it is applied before **hashing** any message and the message pre-processing function comprises 4 steps like shuffling of bits,

25.What are the methods to resolve collision?

Collision resolution strategies we will look at are:

Linear probing.
Double hashing.
Random hashing.
Separate chaining

26.**What is linear probing give example?**

Linear probing is a scheme in computer programming for resolving collisions in hash tables, data structures for maintaining a collection of key–value pairs and looking up the value associated with a given key. ... In these schemes, each cell of a hash table stores a single key–value pair.

27.What is the main disadvantage of linear probing?

Disadvantage - It has secondary clustering. Two keys have the same probe sequence when they hash to the same location. It is a popular collision-resolution technique used in open-addressed hash tables

28.What is the difference between linear probing and quadratic probing?

Linear Probing has the best cache performance but suffers from clustering. Quadratic probing lies between the two in terms of cache performance and clustering. Double caching has poor cache performance but no clustering

29.How do you do quadratic probing?

Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.

30.Which is better Linear probing or chaining?

I'm surprised that you saw chained hashing to be faster than linear probing - in practice, linear probing is typically significantly faster than chaining. ... Theoretically, linear probing only gives expected O(1) lookups if the hash functions are 5-independent or if there's sufficient entropy in the keys.

31.What are the advantages and disadvantages of separate chaining and linear probing?

	Advantages	Disadvantages
Open Addressing	Memory Efficient – stores elements in empty array spaces	Creates Clusters with **Linear** and Quadratic **Probing**

Separate Chaining	Very Easy to implement	Memory Inefficient – requires a secondary data structure to store collisions Long Chains will produce **Linear** search times

32.What does separate chaining mean?
(data structure) **Definition**: A scheme in which each position in the hash table has a list to handle collisions. Each position may be just a link to the list (direct **chaining**) or may be an item and a link, essentially, the head of a list

33.What is double hashing example?
Double hashing is a collision resolving technique in Open Addressed **Hash** tables. **Double hashing** uses the idea of applying a second **hash** function to key when a collision occurs. **Double hashing** can be done using : (hash1(key) + i * hash2(key)) % TABLE_SIZE. Here hash1() and hash2() are **hash** functions and TABLE_SIZE.

34.What is re hashing & double hashing?
Double Hashing or **rehashing**: **Hash** the key a second time, using a different **hash** function, and use the result as the step size. For a given key the step size remains constant throughout a probe, but it is different for different keys. ... **Double hashing** requires that the size of the **hash** table is a prime number.

35.Is double hashing more secure?
No, multiple **hashes** are not less **secure**; they are an essential part of **secure** password use. Iterating the **hash** increases the time it takes for an attacker to try each password in their list of candidates. You can easily increase the time it takes to attack a password from hours to years

36.Why is random hashing important?
In random hashing, every time the software is initialized, a new hash function is picked, at random. This does not make attacks impossible, but it makes them much more difficult.

37.What is uniform and random hash function?
One property of ideal random hash functions that seems intuitively useful is uniformity. A. family H of hash functions is uniform if choosing a hash function uniformly at random from H. makes every hash value equally likely for every item in the universe

38.What is the key idea behind separate chaining?
The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

MCQ SET 1

1. What is a hash table?
a) A structure that maps values to keys
b) A structure that maps keys to values
c) A structure used for storage
d) A structure used to implement stack and queue
Answer: b
Explanation: A hash table is used to implement associative arrays which has a key-value pair, so the has table maps keys to values.

2. If several elements are competing for the same bucket in the hash table, what is it called?
a) Diffusion b) Replication c) Collision d) Duplication
Answer: c
Explanation: In a hash table, if sevaral elements are computing for the same bucket then there will be a clash among elements. This condition is called Collision. The Collision is reduced by adding elements to a linked list and head address of linked list is placed in hash table.

3. What is direct addressing?
a) Distinct array position for every possible key b) Fewer array positions than keys
c) Fewer keys than array positions d) Same array position for all keys
Answer: a
Explanation: Direct addressing is possible only when we can afford to allocate an array that has one position for every possible key.
4. What is the search complexity in direct addressing?
a) O(n) b) O(logn) c) O(nlogn) d) O(1)
Answer: d
Explanation: Since every key has a unique array position, searching takes a constant time.
5. What is a hash function?
a) A function has allocated memory to keys
b) A function that computes the location of the key in the array
c) A function that creates an array
d) A function that computes the location of the values in the array
Answer: b
Explanation: In a hash table, there are fewer array positions than the keys, so the position of the key in the array has to be computed, this is done using the hash function.
6. Which of the following is not a technique to avoid a collision?
a) Make the hash function appear random b) Use the chaining method
c) Use uniform hashing d) Increasing hash table size
Answer: d
Explanation: On increasing hash table size, space complexity will increase as we need to reallocate the memory size of hash table for every collision. It is not the best technique to avoid a collision. We can avoid collision by making hash function random, chaining method and uniform hashing.
7. What is the load factor?
a) Average array size b) Average key size
c) Average chain lengt d) Average hash table length
Answer: c
Explanation: In simple chaining, load factor is the average number of elements stored in a chain, and is given by the ratio of number of elements stored to the number of slots in the array.
8. What is simple uniform hashing?
a) Every element has equal probability of hashing into any of the slots
b) A weighted probabilistic method is used to hash elements into the slots
c) Elements has Random probability of hashing into array slots
d) Elements are hashed based on priority
Answer: a
Explanation: In simple uniform hashing, any given element is equally likely to hash into any of the slots available in the array.
9. In simple uniform hashing, what is the search complexity?
a) O(n) b) O(logn) c) O(nlogn) d) O(1)
Answer: d
Explanation: There are two cases, once when the search is successful and when it is unsuccessful, but in both the cases, the complexity is O(1+alpha) where 1 is to compute the hash function and alpha is the load factor.
10. In simple chaining, what data structure is appropriate?
a) Singly linked list b) Doubly linked list
c) Circular linked list d) Binary trees

Answer: b
Explanation: Deletion becomes easier with doubly linked list, hence it is appropriate.
11. The case in which a key other than the desired one is kept at the identified location is called?
a) Hashing b) Collision c) Chaining d) Open addressing
Answer: b
Explanation: When some other value is placed at a specified location other than the desired key, it is said to be a collision.
12. What data organization method is used in hash tables?
a) Stack b) Array c) Linked list d) Queue
Answer: c
Explanation: The data structure used to organize data for hash tables is linked list. It contains a data field and a pointer field.
13. The task of generating alternative indices for a node is called?
a) Collision handling b) Collision detection
c) Collision recovery d) Closed hashing
Answer: a
Explanation: Collision handling involves the process of formulating alternative indices for a key.
14. Which of the following is not a collision resolution technique?
a) Separate chaining b) Linear probing
c) Quadratic probing d) Hashing
Answer: d
Explanation: Hashing is a technique of placing data items in specific locations. Collision may occur in hashing but hashing is not a collision resolution technique.
15. Hashing is the problem of finding an appropriate mapping of keys into addresses.
a) True b) False
Answer: a
Explanation: Hashing is a data structure which is used to locate data in a table based on a key value.
16. In a hash table of size 10, where is element 7 placed?
a) 6 b) 7 c) 17 d) 16
Answer: b
Explanation: The hash location is defined as hash(f)= key mod table_size.
7 mod 10 gives 7. It is placed in 7th position.
17. What should be the load factor for separate chaining hashing?
a) 0.5 b) 1 c) 1.5 d) 2
Answer: b
Explanation: For hashing using separate chaining method, the load factor should be maintained as 1. For open addressing method, it should not exceed 0.5.
18. Which of the following operations are done in a hash table?
a) Insert only b) Search only
c) Insert and search d) Replace
Answer: c
Explanation: Hash tables are used to implement Insert and Find operations in constant average time. This is the prime purpose of hashing.
19. Which of the following is identical to that of a separate chaining hash node?
a) Linked list b) Array c) Stack d) Queue

Answer: a

Explanation: The hash node in separate chaining is similar to that of a linked list. The separate chaining hash table is an array of linked lists.

20. Which of the following is the hashing function for separate chaining?

a) H(x)=(hash(x)+f(i)) mod table size b) H(x)=hash(x)+i^{2} mod table size

c) H(x)=x mod table size d) H(x)=x mod (table size * 2)

Answer: c

Explanation: The hashing function for separate chaining is given by H(x)= key mod table size. H(x)=hash(x)+i2 mod table size defines quadratic probing.

SET 2

1. What is the correct notation for a load factor?

a) Ω b) ∞ c) ∑ d) ⅄

Answer: d

Explanation: In general, load factor is denoted as ⅄. In separate chaining method, load factor is maintained as 1.0.

2. In hash tables, how many traversal of links does a successful search require?

a) 1+⅄ b) 1+⅄^{2} c) 1+ (⅄/2) d) ⅄^{3}

Answer: c

Explanation: A successful search requires about 1+ (⅄/2) links to be traversed. There is a guarantee that at least one link must be traversed.

3. Which of the following is a disadvantage of using separate chaining using linked lists?

a) It requires many pointers b) It requires linked lists

c) It uses array d) It does not resolve collision

Answer: a

Explanation: One of the major disadvantages of using separate chaining is the requirement of pointers. If the number of elements are more, it requires more pointers.

4. What is the worst case search time of a hashing using separate chaining algorithm?

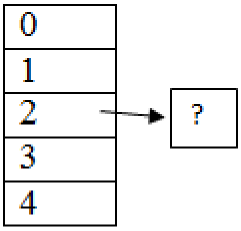
a) O(N log N) b) O(N) c) O(N^{2}) d) O(N^{3})

Answer: b

Explanation: The worst case search time of separate chaining algorithm using linked lists is mathematically found to be O(N).

5. From the given table, find '?'.

Given: hash(x)= x mod 10



a) 13 b) 16 c) 12 d) 14

Answer: c

Explanation: From the given options, 12 mod 10 hashes to 2 and hence '?' = 12.

6. Which of the following is used in hash tables to determine the index of any input record?

a) hash function b) hash linked list c) hash tree d) hash chaining

Answer: a

Explanation: Hash table is an example of a data structure that is built for fast access of elements. Hash functions are used to determine the index of any input record in a hash table.

7. What is the advantage of a hash table as a data structure?
a) faster access of data b) easy to implement
c) very efficient for less number of entries d) exhibit good locality of reference
Answer: a
Explanation: Hash table is a data structure that has an advantage that it allows fast access of elements. Hash functions are used to determine the index of any input record in a hash table.

8. What is the use of a hash function?
a) to calculate and return the index of corresponding data b) to store data
c) to erase data d) to change data
Answer: a
Explanation: Hash function calculates and returns the index for corresponding data. This data is then mapped in a hash table.

9. What is the time complexity of insert function in a hash table using a doubly linked list?
a) O(1) b) O(n) c) O(log n) d) O(n log n)
Answer: a
Explanation: Time complexity of insert function in a hash table is O(1). Condition is that the number of collisions should be low.

10. What is the time complexity of search function in a hash table using a doubly linked list?
a) O(1) b) O(n) c) O(log n) d) O(n log n)
Answer: a
Explanation: Time complexity of search function in a hash table is O(1). Condition is that the number of collisions should be low.

11. What is the time complexity of delete function in the hash table using a doubly linked list?
a) O(1) b) O(n) c) O(log n) d) O(n log n)
Answer: a
Explanation: Time complexity of delete function in a hash table is O(1). Condition is that the hash function should be such that the number of collisions should be low.

12. Hashing can be used to encrypt and decrypt digital signatures.
a) true b) false
Answer: a
Explanation: Hashing is also used in encryption algorithms. It is used for encryption and decryption of digital signatures.

13. What is the advantage of using a doubly linked list for chaining over singly linked list?
a) it takes less memory b) it is easy to implement
c) it makes the process of insertion and deletion fasterd) it causes less collisions
Answer: c
Explanation: Using a doubly linked list reduces time complexity significantly. Though it uses more memory to store the extra pointer.

14. Which of the following technique stores data in the hash table itself in case of a collision?
a) Open addressing b) Chaining using linked list
c) Chaining using doubly linked list d) Chaining using binary tree
Answer: a
Explanation: Open addressing is used to store data in the table itself in case of a collision. Whereas chaining stores data in a separate entity.

15. Which of the following technique stores data in a separate entity in case of a collision?
a) Open addressing b) Chaining using doubly linked list

c) Linear probing d) Double hashing
Answer: b
Explanation: Chaining using doubly linked list is used to store data in a separate entity (doubly linked list in this case) in case of a collision. Whereas open addressing stores it in the table itself.
16. Collision is caused due to the presence of two keys having the same value.
a) True b) False
Answer: a
Explanation: A collision is caused due to the presence of two keys having the same value. It is handled by using any one of the two methods namely:- Chaining and Open addressing.
17. Which of the following variant of a hash table has the best cache performance?
a) hash table using a linked list for separate chaining
b) hash table using binary search tree for separate chaining
c) hash table using open addressing
d) hash table using a doubly linked list for separate chaining
View Answer
Answer: c
Explanation: Implementation of the hash table using open addressing has a better cache performance as compared to separate chaining. It is because open addressing stores data in the same table without using any extra space.
18. What is the advantage of hashing with chaining?
a) cache performance is good b) uses less space
c) less sensitive to hash function d) has a time complexity of O(n) in the worst case
Answer: c
Explanation: Hashing with separate chaining has an advantage that it is less sensitive to a hash function. It is also easy to implement.
19. What is the disadvantage of hashing with chaining?
a) not easy to implement b) takes more space
c) quite sensitive to hash function d) table gets filled up easily
Answer: b
Explanation: Hashing with separate chaining has a disadvantage that it takes more space. This space is used for storing elements in case of a collision.
20. What is the time complexity of insert function in a hash table using a binary tree?
a) O(1) b) O(n) c) O(log n) d) O(n log n)
Answer: a
Explanation: Time complexity of insert function in a hash table is O(1) on an average. Condition is that the number of collisions should be low.

SET-3

1. Which of the following helps keys to be mapped into addresses?
a) hash function b) separate chaining
c) open addressing d) chaining using a linked list
Answer: a
Explanation: Hash table is an example of a data structure that is built for fast access of elements. Hash functions are used to determine the index of any input record in a hash table.
2. What is the advantage of the hash table over a linked list?
a) faster access of data b) easy to implement
c) very efficient for less number of entries d) exhibit good locality of reference
Answer: a

Explanation: Hash table is a data structure that has an advantage that it allows fast access of elements. But linked list is easier to implement as compared to the hash table.

3. Which of the following trait of a hash function is most desirable?
a) it should cause less collisions b) it should cause more collisions
c) it should occupy less space d) it should be easy to implement
Answer: a
Explanation: Hash function calculates and returns the index for corresponding data. So the most important trait of a hash function is that it should cause a minimum number of collisions.

4. What is the time complexity of insert function in a hash table using list head?
a) O(1) b) O(n) c) O(log n) d) O(n log n)
Answer: a
Explanation: Time complexity of insert function in a hash table is O(1). Condition is that the number of collisions should be low.

5. What is the time complexity of the search function in a hash table using a binary tree?
a) O(1) b) O(n) c) O(log n) d) O(n log n)
Answer: a
Explanation: Time complexity of search function in a hash table is O(1). Condition is that the number of collisions should be low.

6. What is the time complexity of the delete function in the hash table using a binary tree?
a) O(1) b) O(n) c) O(log n) d) O(n log n)
Answer: a
Explanation: Time complexity of delete function in a hash table is O(1). Condition is that the hash function should be such that the number of collisions should be low.

7. What is the advantage of a hash table over BST?
a) hash table has a better average time complexity for performing insert, delete and search operations
b) hash table requires less space
c) range query is easy with hash table d) easier to implement
Answer: a
Explanation: Hash table and BST both are examples of data structures. Hash table has an advantage that it has a better time complexity for performing insert, delete and search operations.

8. What is the disadvantage of BST over the hash table?
a) BST is easier to implement
b) BST can get the keys sorted by just performing inorder traversal
c) BST can perform range query easily
d) Time complexity of hash table in inserting, searching and deleting is less than that of BST
Answer: d
Explanation: BST has an advantage that it is easier to implement, can get the keys sorted by just performing in-order traversal and can perform range query easily. Hash table has lesser time complexity for performing insert, delete and search operations.

9. Which of the following technique stores data separately in case of a collision?
a) Open addressing b) Double hashing
c) Quadratic probing d) Chaining using a binary tree
Answer: d
Explanation: Open addressing is used to store data in the table itself in case of a collision. Whereas chaining stores data in a separate entity.

10. Separate chaining is easier to implement as compared to open addressing.
a) true b) false

Answer: a

Explanation: There are two methods of handling collisions in a hash table:- open addressing and separate chaining. Open addressing requires more computation as compared to separate chaining.

11. In open addressing the hash table can never become full.

a) True b) False

Answer: b

Explanation: There are two methods of handling collisions in a hash table:- open addressing and separate chaining. In open addressing the hash table can become full.

12. What is the time complexity of delete function in the hash table using list head?

a) O(1) b) O(n) c) O(log n) d) O(n log n)

Answer: a

Explanation: Time complexity of delete function in a hash table is O(1). Condition is that the hash function should be such that the number of collisions should be low.

13. A hash table may become full in the case when we use open addressing.

a) true b) false

Answer: a

Explanation: A hash table may become full in the case when we use open addressing. But when we use separate chaining it does not happen.

14. What is the advantage of using linked list over the doubly linked list for chaining?

a) it takes less memory b) it causes more collisions

c) it makes the process of insertion and deletion faster

d) it causes less collisions

Answer: a

Explanation: Singly linked list takes lesser space as compared to doubly linked list. But the time complexity of the singly linked list is more than a doubly linked list.

15. What is the worst case time complexity of insert function in the hash table when the list head is used for chaining?

a) O(1) b) O(n log n) c) O(log n) d) O(n)

Answer: d

Explanation: Worst case time complexity of insert function in the hash table when the list head is used for chaining is O(n). It is caused when a number of collisions are very high.

16. Which of the following technique is used for handling collisions in a hash table?

a) Open addressing b) Hashing c) Searching d) Hash function

Answer: a

Explanation: Open addressing is the technique which is used for handling collisions in a hash table. Separate chaining is another technique which is used for the same purpose.

17. By implementing separate chaining using list head we can reduce the number of collisions drastically.

a) True b) False

Answer: b

Explanation: Collision is caused when a hash function returns repeated values. So collisions can be reduced by developing a better hash function. Whereas separate chaining using list head is a collision handling technique so it has no relation with a number of collisions taking place.

18. Which of the following is an advantage of open addressing over separate chaining?

a) it is simpler to implement b) table never gets full

c) it is less sensitive to hash function d) it has better cache performance

Answer: a

Explanation: Open addressing is the technique which is used for handling collisions in a hash table. It has a better cache performance as everything is stored in the same table.

19. What is the time complexity of search function in a hash table using list head?
a) O(1) b) O(n) c) O(log n) d) O(n log n)
Answer: a
Explanation: Time complexity of search function in a hash table is O(1). Condition is that the number of collisions should be low.
20. What is the value of m' if the value of m is 19?
a) 11 b) 18 c) 17 d) 15
Answer: c
Explanation: The value of m' is chosen as a prime number slightly less than the value of m. Here the value of m is 19, hence the value of m' can be chosen as 17.

SET 4

1. Which of the following problems occur due to linear probing?
a) Primary collision b) Secondary collision
c) Separate chaining d) Extendible hashing
Answer: a
Explanation: Primary collision occurs due to linear probing technique. It is overcome using a quadratic probing technique.
2. How many probes are required on average for insertion and successful search?
a) 4 and 10 b) 2 and 6 c) 2.5 and 1.5 d) 3.5 and 1.5
Answer: c
Explanation: Using formula, the average number of probes required for insertion is 2.5 and for a successful search, it is 1.5.
3. What is the load factor for an open addressing technique?
a) 1 b) 0.5 c) 1.5 d) 0
Answer: b
Explanation: The load factor for an open addressing technique should be 0.5. For separate chaining technique, the load factor is 1.
4. Which of the following is not a collision resolution strategy for open addressing?
a) Linear probing b) Quadratic probing
c) Double hashing d) Rehashing
Answer: d
Explanation: Linear probing, quadratic probing and double hashing are all collision resolution strategies for open addressing whereas rehashing is a different technique.
5. In linear probing, the cost of an unsuccessful search can be used to compute the average cost of a successful search.
a) True b) False
Answer: a
Explanation: Using random collision resolution algorithm, the cost of an unsuccessful search can be used to compute the average cost of a successful search.
6. Which of the following is the correct function definition for linear probing?
a) F(i)= 1 b) F(i)=I c) F(i)=i^2 d) F(i)=i+1
Answer: b
Explanation: The function used in linear probing is defined as, F(i)=I where i=0,1,2,3….,n.
7………….. is not a theoretical problem but actually occurs in real implementations of probing.
a) Hashing b) Clustering c) Rehashing d) Collision

Answer: b

Explanation: Clustering is not a theoretical problem but it occurs in implementations of hashing. Rehashing is a kind of hashing.

8. What is the hash function used in linear probing?

a) H(x)= key mod table size b) H(x)= (key+ F(i^2)) mod table size

c) H(x)= (key+ F(i)) mod table size d) H(x)= X mod 17

Answer: c

Explanation: The hash function used in linear probing is defined to be H(x)= (key+ F(i)) mod table size where i=0,1,2,3,…,n.

9. Hashing can be used in online spelling checkers.

a) True b) False

Answer: a

Explanation: If misspelling detection is important, an entire dictionary can be pre-hashed and words can be checked in constant time.

10. Which technique has the greatest number of probe sequences?

a) Linear probing b) Quadratic probing

c) Double hashing d) Closed hashing

Answer: c

Explanation: Double hashing has the greatest number of probe sequences thereby efficiently resolves hash collision problems.

11. What is the formula to find the expected number of probes for an unsuccessful search in linear probing?

a) 121+1(1−⅄) b) 121+1(1−⅄)2 c) 121+1(1+⅄)

d) 121+1(1+⅄)(1−⅄)

Answer: b

Explanation: The mathematical formula for calculating the number of probes for an unsuccessful search is 121+1(1−⅄)2. For insertion, it is 121+1(1−⅄).

12. Which of the following schemes does quadratic probing come under?

a) rehashing b) extended hashing

c) separate chaining d) open addressing

Answer: d

Explanation: Quadratic probing comes under open addressing scheme to resolve collisions in hash tables.

13. Quadratic probing overcomes primary collision.

a) True b) False

Answer: a

Explanation: Quadratic probing can overcome primary collision that occurs in linear probing but a secondary collision occurs in quadratic probing.

14. What kind of deletion is implemented by hashing using open addressing?

a) active deletion b) standard deletion c) lazy deletion d) no deletion

Answer: c

Explanation: Standard deletion cannot be performed in an open addressing hash table, because the cells might have caused collision. Hence, the hash tables implement lazy deletion.

15. In quadratic probing, if the table size is prime, a new element cannot be inserted if the table is half full.

a) True b) False

Answer: b

Explanation: In quadratic probing, if the table size is prime, we can insert a new element even though table is exactly half filled. We can’t insert element if table size is more than half filled.

16. Which of the following is the correct function definition for quadratic probing?

a) $F(i)=i^2$ b) F(i)=I c) F(i)=i+1 d) $F(i)=i^2+1$

Answer: a

Explanation: The function of quadratic probing is defined as $F(i)=i^2$. The function of linear probing is defined as F(i)=i.

17.. How many constraints are to be met to successfully implement quadratic probing?

a) 1 b) 2 c) 3 d) 4

Answer: b

Explanation: 2 requirements are to be met with respect to table size. The table size should be a prime number and the table size should not be more than half full.

18. Which among the following is the best technique to handle collision?

a) Quadratic probing b) Linear probing

c) Double hashing d) Separate chaining

Answer: a

Explanation: Quadratic probing handles primary collision occurring in the linear probing method. Although secondary collision occurs in quadratic probing, it can be removed by extra multiplications and divisions.

19. Which of the following techniques offer better cache performance?

a) Quadratic probing b) Linear probing

c) Double hashing d) Rehashing

Answer: b

Explanation: Linear probing offers better cache performance than quadratic probing and also it preserves locality of reference.

20. What is the formula used in quadratic probing?

a) Hash key = key mod table size b) Hash key=(hash(x)+F(i)) mod table size

c) Hash key=(hash(x)+$F(i^2)$) mod table size d) H(x) = x mod 17

Answer: c

Explanation: Hash key=(hash(x)+$F(i^2)$) mod table size is the formula for quadratic probing. Hash key = (hash(x)+F(i)) mod table size is the formula for linear probing.

<u>SET 5</u>

1. Which scheme uses a randomization approach?

a) hashing by division b) hashing by multiplication

c) universal hashing d) open addressing

Answer: c

Explanation: Universal hashing scheme uses a randomization approach whereas hashing by division and hashing by multiplication are heuristic in nature.

2. Which hash function satisfies the condition of simple uniform hashing?

a) h(k) = lowerbound(km) b) h(k)= upperbound(mk)

c) h(k)= lowerbound(k) d) h(k)= upperbound(k)

Answer: a

Explanation: If the keys are known to be random real numbers k independently and uniformly distributed in the range 0<=k<=1, the hash function which satisfies the condition of simple uniform hashing is h(k)= lowerbound(km).

3. A good hash approach is to derive the hash value that is expected to be dependent of any patterns that might exist in the data.

a) True b) False

Answer: b

Explanation: A hash value is expected to be unrelated or independent of any patterns in the distribution of keys.

4. Interpret the given character string as an integer expressed in suitable radix notation.Character string = pt

a) 14963 b) 14392 c) 12784 d) 14452

Answer: d

Explanation: The given character string can be interpreted as (112,116) (Ascii values) then expressed as a radix-128 integer, hence the value is 112*128 + 116 = 14452.

5. What is the hash function used in the division method?

a) h(k) = k/m b) h(k) = k mod m c) h(k) = m/k d) h(k) = m mod k

Answer: b

Explanation: In division method for creating hash functions, k keys are mapped into one of m slots by taking the reminder of k divided by m.

6. What can be the value of m in the division method?

a) Any prime number b) Any even number c) $2^p - 1$ d) 2^p

Answer: a

Explanation: A prime number not too close to an exact power of 2 is often a good choice for m since it reduces the number of collisions which are likely to occur.

7. Which scheme provides good performance?

a) open addressing b) universal hashing

c) hashing by division d) hashing by multiplication

Answer: b

Explanation: Universal hashing scheme provides better performance than other schemes because it uses a unique randomisation approach.

8. Using division method, in a given hash table of size 157, the key of value 172 be placed at position ____

a) 19 b) 72 c) 15 d) 17

Answer: c

Explanation: The key 172 can be placed at position 15 by using the formula

H(k) = k mod m

H(k) = 172 mod 157

H(k) = 15.

9. How many steps are involved in creating a hash function using a multiplication method?

a) 1 b) 4 c) 3 d) 2

Answer: d

Explanation: In multiplication method 2 steps are involved. First multiplying the key value by a constant. Then multiplying this value by m.

10. What is the hash function used in multiplication method?

a) h(k) = floor(m(kA mod 1)) b) h(k) = ceil(m(kA mod 1))

c) h(k) = floor(kA mod m) d) h(k) = ceil(kA mod m)

Answer: a

Explanation: The hash function can be computed by multiplying m with the fractional part of kA (kA mod 1) and then computing the floor value of the result.

11. What is the advantage of the multiplication method?

a) only 2 steps are involved b) using constant

c) value of m not critical d) simple multiplication

Answer: c

Explanation: The value of m can be simply in powers of 2 since we can easily implement the function in most computers. $m=2^p$ where p is an integer.

12. What is the table size when the value of p is 7 in multiplication method of creating hash functions?

a) 14 b) 128 c) 49 d) 127

Answer: b

Explanation: In multiplication method of creating hash functions the table size can be taken in integral powers of 2.

$m = 2^p$

$m= 2^7$

m = 128.

13. What is the value of h(k) for the key 123456?Given: p=14, s=2654435769, w=32

a) 123 b) 456 c) 70 d) 67

Answer: d

Explanation: $A = s/2^w$

$A = 2654435769/ 2^{32}$

$k.A = 123456 * (2654435769/ 2^{32})$

$= (76300 * 2^{32}) + 17612864$

Hence r1= 76300; r0=17612864

Since w=14 the 14 most significant bits of r0 yield the value of h(k) as 67.

14. What is the average retrieval time when n keys hash to the same slot?

a) Theta(n) b) Theta(n^2) c) Theta(nlog n) d) Big-Oh(n^2)

Answer: a

Explanation: The average retrieval time when n keys hash to the same slot is given by Theta(n) as the collision occurs in the hash table.

15. Collisions can be reduced by choosing a hash function randomly in a way that is independent of the keys that are actually to be stored.

a) True b) False

Answer: a

Explanation: Because of randomization, the algorithm can behave differently on each execution, providing good average case performance for any input.

16. Double hashing is one of the best methods available for open addressing.

a) True b) False

Answer: a

Explanation: Double hashing is one of the best methods for open addressing because the permutations produced have many characteristics of randomly chosen permutations.

17. What is the hash function used in Double Hashing?

a) (h1(k) – i*h2(k))mod m b) h1(k) + h2(k)

c) (h1(k) + i*h2(k))mod m d) (h1(k) + h2(k))mod m

Answer: c

Explanation: Double hashing uses a hash function of the form (h1(k) + i*h2(k))mod m where h1 and h2 are auxiliary hash functions and m is the size of the hash table.

18. On what value does the probe sequence depend on?

a) c1 b) k c) c2 d) m

Answer: b

Explanation: The probe sequence depends in upon the key k since the initial probe position, the offset or both may vary.

19. The value of h2(k) can be composite relatively to the hash table size m.

a) True b) False

Answer: b

Explanation: The value h2(k) must be relatively prime to the hash table size m for the entire

hash table to be searched. It can be ensured by having m in powers of 2 and designing h2 so that it produces an odd number.
20. What is the running time of double hashing?
a) Theta(m) b) Theta(m^2) c) Theta(m log k) d) Theta(m^3)
Answer: a
Explanation: The running time of double hashing is Theta(m) since each possible (h1(k), h2(k)) pair yields a distinct probe sequence. Hence the performance of double hashing is better.