Data Stucture

**What is data structure explain its types?**

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organising data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List

Primitive data structures : These are the basic data structures and are directly operated upon by the machine instructions, which is in a primitive level. They are integers, floating point numbers, characters, string constants, pointers etc

Non-primitive data structures : It is a more sophisticated data structure emphasizing on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items linear and non linear

Data structure where data elements are arranged sequentially or linearly where the elements are attached to its previous and next adjacent in what is called a **linear data structure**.

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together.

Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.

[Stack](http://www.geeksforgeeks.org/stack-data-structure/)**:** Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

[Queue](http://www.geeksforgeeks.org/queue-data-structure/)**:** Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO)

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

Tree

It is a non-linear data structure that consists of various linked nodes

Graph

A graph is a non-linear data structure that has a finite number of vertices and edges, and these edges are used to connect the vertices.

**2 Data structure operations**

**Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting..

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:**The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

**Stack definition operation and application**

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out)

Mainly the following three basic operations are performed in the stack:

Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Peek or Top: Returns top element of stack.

isEmpty: Returns true if stack is empty, e

Applications of stack:

* [Balancing of symbols](https://www.geeksforgeeks.org/check-for-balanced-parentheses-in-an-expression/)
* [Infix to Postfix](https://www.geeksforgeeks.org/stack-set-2-infix-to-postfix/) /Prefix conversion
* Redo-undo features at many places like editors, photoshop.
* Forward and backward feature in web browsers
* Used in many algorithms like [Tower of Hanoi,](https://www.geeksforgeeks.org/recursive-functions/)[tree traversals](https://www.geeksforgeeks.org/618/), [stock span problem](https://www.geeksforgeeks.org/the-stock-span-problem/), [histogram problem](https://www.geeksforgeeks.org/largest-rectangular-area-in-a-histogram-set-1/).

**Push algorithum**

begin procedure push: stack, data

if stack is full

return null

endif

top ← top + 1

stack[top] ← data

end procedure

**example**

void push(int data) {

if(!isFull()) {

top = top + 1;

stack[top] = data;

} else {

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

}

**POP aloGrithum**

begin procedure pop: stack

if stack is empty

return null

endif

data ← stack[top]

top ← top - 1

return data

end procedure

**Example**

int pop(int data) {

if(!isempty()) {

data = stack[top];

top = top - 1;

return data;

} else {

printf("Could not retrieve data, Stack is empty.\n");

}

}

**Algorithm/Pseudo code for converting infix expression into postfix expression:**

1. Initialize empty stack with symbol ‘#’.

i.e. st[0]=‘#’

2. Read the character from infix expression.

i.e. symbol=infix[ i]

3. while SP(st[top]) > IP(symbol)

postfix [ j]=pop(st[top])

4. if SP(st[top]) != IP(symbol)

push(symbol)

else

pop(st[top])

5. Repeat the steps (2) to (4) till the last character of infix expression.

6. While stack becomes empty, //only for partially parenthesized expression

postfix [ j]=pop(st[top])

**What is recursion and explain tower of Hanoi problem**

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

**#include<stdio.h>**

**int count=0;**

**void tower(int n, char s, char t, char d)**

**{**

**if(n==1)**

**{**

**printf(“Move disc 1 from %c to %c “, s, d);**

**count ++;**

**return;**

**}**

**tower(n-1, s, d, t);**

**printf(“Move disc %d from %c to %c”, n, s,d);**

**count ++;**

**tower(n-1, t, s, d);**

**}**

**void main()**

**{**

**int n;**

**printf(“Enter the number of discs”);**

**scanf(“%d”, &n);**

**tower(n, ‘A’,’B’,’C’);**

**printf(“Total number of moves =%d”, count);**

**}**

**Fibonacci number**

Finding nth Fibonacci number

#include<stdio.h>

int fibo(int n)

{

if (n==1)

return 0;

else if (n==2)

return 1;

return fibo(n-1) + fibo (n-2);

}

void main()

{

int n;

printf(“Enter value of n”);

scanf(“%d”, &n);

if(n<=0)

printf(“invalid input”);

else

printf(“%d th fibonacci number is %d “, n, fibo(n));

}

**Unit 3**

**Explain Queue and its types**

Queue is a non-primitive linear data structure, where the elements are inserted at rear end and deleted from the front end.The item inserted first will be the first to be deleted. Hence it is known as First In First Out(FIFO) data structure.

The primitive operations of a queue structure include:

* Inserting an element into queue
* Deleting element from queue
* Displaying the contents of a queue

**Circular Queue** is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle

In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

Program for Implementation of Circular Queue

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

#define MAX 3

void insert(int q[], int \*f, int \*r, int item)

{

if(\*f==(\*r+1)%MAX)

{

printf("\nQueue overflow");

return;

}

\*r=(\*r+1)%MAX;

q[\*r]=item;

if(\*f== -1)

(\*f)++;

}

void del(int q[], int \*f, int \*r)

{

if(\*f==-1 )

{

printf("\nQueue underflow");

return;

}

printf("\nDeleted element is %d", q[\*f]);

if(\*f==\*r)

\*f=\*r=-1;

else

\*f=(\*f+1)%MAX;

}

void disp(int q[], int \*f, int \*r)

{

int i;

if(\*f== -1)

{

printf("\nNo elements to display!!");

return;

}

printf("\n Contents of queue:\n");

if(\*f>\*r)

{

for(i=\*f;i<MAX;i++)

printf("%d\t",q[i]);

for(i=0;i<=\*r;i++)

printf("%d\t", q[i]);

}

else

{

for(i=\*f;i<=\*r;i++)

printf("%d\t",q[i]);

}

}

void main()

{

int q[10], f=-1, r=-1,item, opt;

for(;;)

{

printf("\n\*\*\*\*\*Circular Queue operations\*\*\*\*\*");

printf("\n1.Insert\n 2.Delete\n 3.Display \n 4.Exit");

printf("\nEnter your option: ");

scanf("%d",&opt);

switch(opt)

{

case 1: printf("\nEnter item to be inserted:");

scanf("%d",&item);

insert(q,&f,&r,item);

break;

case 2: del(q, &f, &r);

break;

case 3: disp(q, &f,&r);

break;

case 4:

default:exit(0);

}

}

}

**Priority Queue**

Priority Queue is a data structure in which the items are served (deleted) based on their priority levels. The insertion and deletion operations of priority queue are based on the priority of the elements.

**Double-Ended Queue**

A double ended queue or deque (pronounced as deck) is a set of items from which items may be deleted from either end and items may be inserted at either end. A deque can have

some sub-types:

* Input restricted deque: deletion can be made at both the ends, but insertion can be

made at only one end.

* Output restricted deque: deletion is at only one end and insertion can be made at

both the ends.

**(Dequeue program)**

**Unit 3**

**Explain diff b\w Static and dynamic**

|  |  |
| --- | --- |
| Static | dynamic |
| Used when the number of memory locations required is known in advance | Used when number of memory locations required is unknown. |
| The size of the memory to be allocatedis fixed and can not be varied during runtime. | The size of memory can be increased or decreased during run time. |
| The variables with statically allocated memory are named ones and can bemanipulated directly | The variables with dynamically allocated memory are unnamed and can be manipulated indirectly only with the help of pointers. |
| The allocation and de-allocation of  memory for variables is handled by the compiler automatically. | The memory allocation and de-  allocation must be explicitly performed by the programmer. |
| As memory for the variables is decided at compile time and data manipulation is done on these locations, execution of the program is faster. | As memory must be allocated during run time, the execution will be slower |
| Memory for global andstatic variables will be allocated in Data Segment and for local variables in Stack Segment | Memory allocation will be in Heap  Segment |

**Explain The limitations of an array**

The limitations of an array are explained below −

* An array which is formed will be homogeneous. That is, in an integer array only integer values can be stored
* **Array size is fixed:** The array is static, which means its size is always fixed. The memory which is allocated to it cannot be increased or decreased.
* **Array is Contiguous blocks of memory:** The array stores data in contiguous(one by one) memory location

**Explain memory management ?**

**There are two types of available memories- stack and heap.** Static memory allocation can only be done on stack whereas dynamic memory allocation can be done on both stack and heap

**While allocating memory on heap we need to delete the memory manually as memory is not freed(deallocated) by the compiler itself even if the scope of allocated memory finishes(as in case of stack)**

There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

malloc()

calloc()

free()

realloc()

**malloc() method**

The “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.

**Syntax:**

ptr = (cast-type\*) malloc(byte-size)

**For Example:**

***ptr = (int\*) malloc(100 \* sizeof(int));***

**calloc() method**

“calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:

It initializes each block with a default value ‘0’.

It has two parameters or arguments as compare to malloc().  
Syntax:

ptr = (cast-type\*)calloc(n, element-size);

***example: ptr = (float\*) calloc(25, sizeof(float));***

**free() method**

**“free”** method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used,

Synta**x:**free(ptr);

**Re alloc() method**

if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.  
**Syntax:**

ptr = realloc(ptr, newSize);

**What is linked List ? explain its types**

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers inked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list

Various operations that can be performed on any linked list:

* Insert at front
* Insert at rear
* Delete from front
* Delete from rear
* Display the contents
* Insert at any position
* Delete from any position
* Search for a particular item
* Delete a particular item
* Creating ordered

#include<stdio.h>

#include<alloc.h>

#include<conio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*link;

};

typedef struct node \*NODE;

NODE getnode()

{

NODE x;

x=(NODE) malloc(sizeof(struct node));

if(x==NULL)

{

printf("no memory in heap");

exit(0);

}

return x;

}

NODE insert\_front(int item, NODE start)

{

NODE temp;

temp = getnode();

temp->data=item;

temp->link=start;

return temp;

}

NODE delete\_front(NODE start)

{

NODE temp;

if(start==NULL)

{

printf("no element to delete\n");

return start;

}

temp=start;

printf(“Deleted item=%d", temp->data);

start=start->link;

free(temp);

return start;

}

NODE insert\_rear(int item, NODE start)

{

NODE temp, cur;

temp=getnode();

temp->data=item;

temp->link=NULL;

if (start==NULL)

return temp;

cur=start;

while(cur->link!=NULL)

cur=cur->link;

cur->link=temp;

return start;

}

void display(NODE start)

{

NODE temp;

if(start==NULL)

{

printf("No element to display\n");

return ;

}

printf("The contents of list:\n");

temp=start;

while(temp!=NULL)

{

printf("%d\n", temp->data);

temp=temp->link;

NODE delete\_rear(NODE start)

{

NODE prev, cur;

if(start==NULL)

{

printf("no element to delete\n");

return start;

}

if(start->link==NULL)

{

printf("\nDeleted element is%d", start->data);

free(start);

return NULL;

}

prev=NULL;

cur=start;

while(cur->link!=NULL)

{

prev=cur;

cur=cur->link;

}

printf("\nDeleted element is %d", cur->data); .

free(cur);

prev->link=NULL;

return start;

}

void main()

{

int opt, item;

NODE start=NULL;

for(;;)

{

printf("1.Insert Front\n 2.Insert Rear\n 3. Display\n”);

printf(“ 4.Delete Front\n 5.Delete Rear\n");

printf("enter your option:");

scanf("%d",&opt);

switch(opt)

{

case 1: printf("\nenter item");

scanf("%d",&item);

start=insert\_front(item,start);

break;

case 2: printf("\nenter item");

scanf("%d",&item);

start=insert\_rear(item,start);

break;

case 3: display(start);

break;

case 4: start=delete\_front(start);

break;

case 5: start=delete\_rear(start);

break;

default: exit(0);

}

}

}

**Explain circular linked list**

***Circular linked list*** is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list

**Program for Operations on Circular linked lists**

#include<stdio.h>

#include<alloc.h>

#include<conio.h>

#include<stdlib.h>

struct node

{

int data;

struct node \*link;

};

typedef struct node \*NODE;

NODE getnode()

{

NODE x;

x=(NODE) malloc(sizeof(struct node));

if(x==NULL)

{

printf("no memory in heap");

exit(0);

}

return x;

}

NODE insert\_front(int item, NODE last)

{

NODE temp;

temp = getnode();

temp->data=item;

temp->link=temp;

if(last==NULL)

return temp;

temp ->link = last->link;

last ->link=temp;

return last;

}

NODE insert\_rear(int item, NODE last)

{

NODE temp;

temp=getnode();

temp->data=item;

temp->link=temp;

if (last==NULL)

return temp;

temp->link=last->link;

last->link=temp;

return temp;

}

void display(NODE last)

{

NODE temp;

if(last==NULL)

{

printf("No element to display\n");

return ;

}

temp=last->link;

printf("The contents of list:\n");

while(temp!=last)

{

printf(“%d\n” temp->data);

temp=temp->link;

}

printf(“%d”, temp->data);

}

NODE delete\_front(NODE last)

{

NODE temp;

if(last==NULL)

{

printf("no element to delete\n");

return NULL;

}

if(last->link==last)

{

printf(“The item deleted is %d”, last->data);

free(last);

return NULL;

}

temp=last->link;

last->link=temp->link;

printf(“Item deleted is %d”, temp->data);

free(temp);

return last;

}

NODE delete\_rear(NODE last)

{

NODE prev;

if(last==NULL)

{

printf("no element to delete\n");

return NULL;

}

if(last->link==last)

{

printf(“The item deleted is %d”, last->data);

free(last);

return NULL;

}

prev=last->link;

while(prev->link!=last)

prev=prev->link;

prev->link=last->link;

printf("\nDeleted element is %d", last->data);

free(last);

return prev;

}

void main()

{

int opt, item;

NODE last=NULL;

for(;;)

{

printf("1.Insert Front \n 2.Insert Rear\n 3.Display\n”)

printf(“ 4.Delete Front \n 5.Delete Rear\n");

printf("Enter your option:");

scanf("%d",&opt);

switch(opt)

{

case 1: printf("\nenter item");

scanf("%d",&item);

last=insert\_front(item,last);

break;

case 2: printf("\nenter item");

scanf("%d",&item);

last=insert\_rear(item,last);

break;

case 3: display(last);

break;

case 4: last=delete\_front(last);

break;

case 5: last=delete\_rear(last);

break;

default: exit(0);

}

}

}

**Doubly Linked Lists** Doubly linked lists are like singly linked lists, in which for each node there are two pointers -- one to the next node, and one to the previous node. This makes life nice in many ways:

• You can traverse lists forward and backward.

• You can insert anywhere in a list easily. This includes inserting before a node, after a node, at the front of the list, and at the end of the list and

• You can delete nodes very easily

/\* Node of a doubly linked list \*/

struct Node {

    int data;

    struct Node\* next; // Pointer to next node in DLL

    struct Node\* prev; // Pointer to previous node in DLL

};

**What is header node**

A header node is a special node that is found at the beginning of the list. A list that contains this type of node, is called the header-linked list. This type of list is useful when information other than that found in each node is needed.

**Unit 5**

**Explain**

**Insertion sort**

Input an array A of *n* numbers

2. Initialize *i* = 1 and repeat through steps 4 by incrementing *i* by one.

(*a*) If (*i* < = *n* – 1)

(*b*) Swap = A [I],

(*c*) Pos = *i* – 1

3. Repeat the step 3 if (Swap < A[Pos] and (Pos >= 0))

(*a*) A [Pos+1] = A [Pos]

(*b*) Pos = Pos-1

4. A [Pos +1] = Swap

5. Exit

**Selection sort**

1. Input *n* numbers of an array A

2. Initialize *i* = 0 and repeat through step5 if (*i* < *n* – 1)

(*a*) min = *a*[*i*]

(*b*) loc = *i*

160 PRINCIPLES OF DATA STRUCTURES USING C AND C++

3. Initialize *j* = *i* + 1 and repeat through step 4 if (*j* < *n* – 1)

4. if (a[*j*] < min)

(*a*) min = a[*j*]

(*b*) loc = *j*

5. if (loc ! = i)

(*a*) swap = *a*[*i*]

(*b*) *a*[*i*] = *a*[loc]

(*c*) *a*[loc] = swap

6. display “the sorted numbers of array A”

7. Exit

**Quick sort**

1. Input *n* number of elements in an array A

2. Initialize low = 2, up = *n* , key = A[(low + up)/2]

3. Repeat through step 8 while (low < = up)

4. Repeat step 5 while(A [low] > key)

5. low = low + 1

6. Repeat step 7 while(A [up] < key)

7. up = up–1

8. If (low < = up)

(*a*) Swap = A [low]

(*b*) A [low] = A [up]

(*c*) A [up] = swap

(*d*) low=low+1

(*e*) up=up–1

9. If (1 < up) Quick sort (A, 1, up)

10. If (low < *n*) Quick sort (A, low, *n*)

11. Exit

**Explain binary Search tree**

Binary Search Tree is a binary tree, which is either empty or satisfies the following

properties :

1. Every node has a value and no two nodes have the same value (*i.e*., all the values

are unique).

2. If there exists a left child or left sub tree then its value is less than the value of the

root.

3. The value(s) in the right child or right sub tree is larger than the value of the root

node.

**Inserting node**

1. Input the DATA to be pushed and ROOT node of the tree

2. NEWNODE = Create a New Node.

3. If (ROOT == NULL)

(*a*) ROOT=NEW NODE

4. Else If (DATA < ROOT →Info)

(*a*) ROOT = ROOT →Lchild

(*b*) GoTo Step 4

5. Else If (DATA > ROOT →Info)

(*a*) ROOT = ROOT →Rchild

(*b*) GoTo Step 4

6. If (DATA < ROOT →Info)

(*a*) ROOT →LChild = NEWNODE

7. Else If (DATA > ROOT →Info)

(*a*) ROOT →RChild = NEWNODE

8. Else

(*a*) Display (“DUPLICATE NODE”)

(*b*) EXIT

9. NEW NODE →Info = DATA

10. NEW NODE →LChild = NULL

11. NEW NODE →RChild = NULL

12. EXIT

**Searching a node**

1. Input the DATA to be searched and assign the address of the root node to ROOT.

2. If (DATA == ROOT →Info)

(*a*) Display “The DATA exist in the tree”

(*b*) GoTo Step 6

3. If (ROOT == NULL)

(*a*) Display “The DATA does not exist”

(*b*) GoTo Step 6

4. If(DATA > ROOT→Info)

(*a*) ROOT = ROOT→RChild

(*b*) GoTo Step 2

5. If(DATA < ROOT→Info)

(*a*) ROOT = ROOT→Lchild

(*b*) GoTo Step 2

6. Exit

**Difference b/w bfs and dfs with algorithum**

[BFS stands for **Breadth First Search**](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/) is a vertex based technique for finding a shortest path in graph. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue.

[DFS stands for **Depth First Search**](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/) is a edge based technique. It uses the [Stack data structure](http://www.geeksforgeeks.org/stack-data-structure/), performs two stages, first visited vertices are pushed into stack and second if there is no vertices then visited vertices are popped.

|  |  |  |
| --- | --- | --- |
| 2. | BFS(Breadth First Search) uses Queue data structure for finding the shortest path. | DFS(Depth First Search) uses Stack data structure. |
| 3. | BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex. | In DFS, we might traverse through more edges to reach a destination vertex from a source. |
| 3. | BFS is more suitable for searching vertices which are closer to the given source. | DFS is more suitable when there are solutions away from source. |
| 4. | BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles. | DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop. | 5. |

**ALGORITHM (BFS)**

1. Input the vertices of the graph and its edges G = (V, E)

2. Input the source vertex and assign it to the variable S.

3. Add or push the source vertex to the queue.

4. Repeat the steps 5 and 6 until the queue is empty (*i.e.,* front > rear)

5. Pop the front element of the queue and display it as visited.

6. Push the vertices, which is neighbor to just, popped element, if it is not in the

queue and displayed (*i.e.,* not visited).

7. Exit.

**Algorithm(dfs)**

1. Input the vertices and edges of the graph G = (V, E).

2. Input the source vertex and assign it to the variable S.

3. Push the source vertex to the stack.

4. Repeat the steps 5 and 6 until the stack is empty.

5. Pop the top element of the stack and display it.

6. Push the vertices which is neighbor to just popped element, if it is not in the

queue and displayed (ie; not visited).

7. Exit.

**Prim’s Algorithum**

Suppose G = (V,E) is a graph and T is a minimum spanning tree of grph G.

1. Initialize the spanning tree T to contain a vertex *v*1.

2. Choose an edge *e* = (*v*1, *v*2) of G such that *v*2 not equal to *v*1 and *e* has smallest

weight among the edges of G incident with *v*1.

3. Select an edge *e* = (*v*2, *v*3) of G such that *v*2 is not equal to *v*3 and *e* has smallest

weight among the edge of G incident with *v*2.

4. Suppose the edge *e*1, *e*2, *e*3, ...... *ei* Then select an edge *ei* + 1 = (V*j*, V*k*) such that

(*a*) V*j* €{*v*1, *v*2, *v*3, ...... *vi*, *vi* + 1} and

(*b*) V*k* ∉€{*v*1, *v*2, *v*3, ...... *vi*, *vi* + 1} such that ei+1 has smallest weight among the

edge of G

5. Repeat the step 4 until (*n* – 1) edges have been chosen

6. Exit

**Shortest path algorithum(Dijisktra)**

1. Input the source vertices and assign it to S

(*a*) Set W(*s*) = 0 and

(*b*) Set W (*v*) = \_\_\_ for all vertices V is not equal to S

2. Set Q = V which is a set of vertices in the graph

3. Suppose *m* be a vertices in Q for which W(*m*) is minimum

4. Make the vertices *m* as visited and delete it from the set Q

5. Find the vertices I which are incident with m and member of Q (That is the

vertices which are not visited)

6. Update the weight of vertices I = {*i*1, *i*2 ...... *ik*} by

(*a*) W(*i*1) = min [W(*i*1), W(*m*) + W(*m*, *i*1)]

7. If any changes is made in W(*v*), store the vertices to corresponding vertices *i*,

using the array, for tracing the shortest path

8. Repeat the process from step 3 to 7 until the set Q is empty

9. Exit

**Unit 4**

**What is asymptotic notations? List and explain the asymptotic notation**

**) Θ Notation:** The theta notation bounds a function from above and below, so it defines exact asymptotic behavior.   
A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.   
3n3 + 6n2 + 6000 = Θ(n3)   
Dropping lower order terms is always fine because there will always be a number(n) after which Θ(n3) has higher values than Θ(n2) irrespective of the constants involved.   
For a given function g(n), we denote Θ(g(n)) is following set of functions. 

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such

that 0 <= c1\*g(n) <= f(n) <= c2\*g(n) for all n >= n0}

**2) Big O Notation:** The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.   
If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:   
1. The worst case time complexity of Insertion Sort is Θ(n^2).   
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and n0 such that 0 <= f(n) <= c\*g(n) forall n >= n0}

**3) Ω Notation:** Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.   
Ω Notation can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the [best case performance of an algorithm is generally not useful](https://www.geeksforgeeks.org/analysis-of-algorithms-set-2-asymptotic-analysis/), the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.  

Ω (g(n)) = {f(n): there exist positive constants c and n0 such that 0 <= c\*g(n) <= f(n) for all n >= n0}.

**Define algorithum.explain the steps involved in algorithum?**

**List out important problem types. Explain any two of them?**