BUBBLE SORT

- In bubble sort, consecutive items are compared and exchanged on each pass through the list.
- In bubble sort, in each pass through the data, **the smallest element is bubbled to the beginning of the unsorted segment array.**
- **Bubble sort algorithm from the low end and bubbled up i.e. bubbles to the largest element**

ALGORITHM:

```
void bubble_sort(int *list,int noofelements )
1. [Initialize]
    curdata,nextdata,temp;
 2. Repeat for curdata = 0 to  curdata<(noofelements -1)
        a. Repeat 1,2  for nextdata= 0 to
            nextdata<  (noofelements- 1)- currentdata)
                1. if (list[nextdata]>list[nextdata+1])
                    swap the values
          2. increment nextdata
    3. Call display_sorted_list(list,noofelements)
```

EFFICIENCY:

- The outer loop executes n times(from 0 to noofelements -1).
- The inner loop is dependant on the outer loop(**nextdata<noofelements-currentdata)**,we have a dependant quadratic loop.
    - The efficiency is **f(n)=(n)(n+1)/2**
    - in big-O notation the efficiency of bubble sort is **$O(n^2)$**

INSERTION SORT

- The list is divided into 2 parts:
    - Sorted and Unsorted
- In this, the first element of the unsorted sublist is transferred to the sorted sublist by inserting it at the appropriate place.
- If there is a list of n elements, the straight insertion sort will take at most n-1 passes to sort the data.
- Algorithm

```
Algorithm insertionsort(int a[], int noofelements)
1. [Declare]
   curdata, prevdata,temp
   2. For curdata=1 to to curdata <noofelements
      1.For prevdata=curdata to  prevdata>0step -1
                    1. if( x[prevdata-1]>x[prevdata])
                        a.temp=x[prevdata]
                        b. x[prevdata]=x[prevdata-1]
                        c. x[prevdata-1]=temp
   3. Display array x after sorting
```

EFFICIENCY:

- The outer loop executes n-1 times(from 1 to noofelements).
- The inner loop is dependant on the outer loop(prevdata=curdata to 1 step -1), we have a dependant quadratic loop.
    - in big-O notation the efficiency of insertion sort is **$O(n^2)$**

RADIX SORT:

- Used to sort a large list of name alphabetically.
- **Algorithm of radix sort**
- **1. Find the largest element in the list and find out the number of digits in the element. The number of digits in the largest element determine the total number of iterations.**
  2. In the first iteration, the elements are picked up and kept in various pockets by checking their unit's digit.
  3. The data is then collected from pocket 0 to pocket 9 and they are given as input to sorter.
  4. In the second iteration, the elements are picked up and kept in various pockets by checking their ten's digit.
  5. Step 3 is repeated
  6. In the third iteration, the elements are picked up and kept in various pockets by checking their hundred's digit.
  7. Step 3 is repeated once again and so on…
- **Complexity of Radix sort:**
  - **The time requirement depends on the number of digits and the number of elements in the file. The loop is traversed "m times" for each digit and the inner loop is traversed "n times" for each element in the file.**
  - **The sort is approx O(m*n) .**
  - **m approximates to $\log_2 n$ and so that O(m*n) approximates to O(n $\log_2 n$).**

QUICK SORT:

- Quick sort is an exchange sort developed by C.A.R Hoare.
- It is more efficient than the bubble sort because fewer exchanges are required to correctly position an element.
- Working
  - Each iteration selects an element known as **pivot** divides the list into 3 groups:
    - A partition of elements whose keys are **less than the pivot's key.**
    - **The pivot element** that is placed in the list
    - A partition of elements whose keys are **greater than the pivot's key.**
    - The **sorting then continues by quick sorting the left followed by quick sorting the right partition.**
- diagram
- **Hoare's original algorithm** selected the **pivot key as the first element in the list**
- **R.C Singleton** improved the sort by **selecting the pivot key as the median value of three elements.**
- Each pass in the quick sort divides the list into 3 parts:
  - A list of elements smaller than the pivot key
  - The pivot key and
  - A list of elements greater than the pivot key
- **Algorithm**

Quick_sort(a[], first ,last)

1.[Initialize]

low=first

high=last

pivot=a[(low+high)/2]

2. Repeat A,B,C

while(low<=high)

A. Repeat step A

while(a[low]<pivot)

A. low=low+1

B. Repeat step B

while(a[high]>pivot)

A. high=high-1

C. if (low<=high)

1.swap low and high values

2. low=low+1

3.high=high-1

4.if(first<high)

Quick_sort(a, first, high)

5.if(low<last)

Quick_sort(a, low,last)

6. End

EFFICIENCY:

- The first loop(step 2) in conjunction with step A and B looks at each element in the portion of the array being sorted.Therefore they loop through the list **n times.**
- **The list is divided into 2 sublists roughly of the same size using the pivot. Because the list is divided into 2 the number of loops is logarithmic.**
- **Therefore the efficiency is O(nlogn)**

MERGE SORT:

Algorithm Merge-Sort(Ar, l, r)

if l < r **then**

1.mid = (l+r)/2

2. Merge-Sort(A, left, mid)

3.Merge-Sort(A, mid+1, right)

4.Merge(A, left, mid +1, right)

HEAP SORT:

- The heap sort algorithm is an improved version of straight selection sort.
- Based on a tree structure that reflects a particular order of a corporate hierarchy.
- i.e. In the corporate management, president is at the top. When the president retires, the VP competes for the job and becomes the president and creates a vacancy.Hence the vacancy continuously appears at the top. This idea illustrates the heap sort method.
- The heap sort proceeds in two phases:
  - The entries are arranged in the heap(build_heap)
  - Remove the element from the top of the heap and promote another entry to take its place.

Heap_Sort(A,n)

1. build_heap(A)
2. Repeat a,b,c for i=n to 1 step -1

   a. swap(A[0], A[n])

   b. n=n-1

c. reheapdown(A,0,n)

SELECTION SORT:

- In this sort, select the smallest item and place it in the sorted order.
- The list at any moment is divided into 2 sublists viz sorted and unsorted which are divided by an imaginary wall.
- Easiest method of sorting.
- In this , select the beginning element and the smallest element in the list and exchange. After each selection and exchange, the wall between the sorted sublist and unsorted sublist moves one element increasing the number of sorted elements and decreasing the number of unsorted elements.
- Each time one element is moved from unsorted sublist to the sorted sublist, one pass is completed.
- For n elements, we need n-1 passes to completely rearrange the data.
- Diagram:
- Algorithm:

Algorithm selection(a[], noofelements)

Pre :list :array of integer elements

last:nth element in the list

Post:sorted list of data

1. For curdata = 0 to curdata< (noofelements-1)
    1. For nextdata = curdata +1 to nextdata<noofelements
        1. if a[curdata]>a[nextdata]
            swap the values

2.Print the sorted array

EFFICIENCY:

- The outer loop executes n-1 times(from 0 to last -1).
- The inner loop is dependant on the outer loop(subsequentdata = currentdata +1),we have a dependant quadratic loop.
    - in big-O notation the efficiency of insertion sort is **$O(n^2)$**

SHELL SORT:

- Shell sort is named after its creator, Donald Shell.
- Improved version of straight insertion sort.
- This method is also called **diminishing-increment sort**
- **Complexity of shell sort :**
- **Worst case performance :depends on gap sequence. But the known is : $O(n\log_2 n)$**
- **Best case performance$O(n)$**
- **Average case performance :depends on gap sequence**
- Algorithm

void shell_sort(int a[],int size)

1.[Declare]

temp,gap,i,j,xchng=1

2. For gap=size/2 to gap> 0 step gap/2

1.do

A.xchng=0

B. for i=0 to i<size-gap

1. if(a[i]>a[i+gap])

a. temp=a[i]
b. a[i] =a[i+gap]
c. a[i+gap]=temp
d. xchng=1

while(xchng ==1)

3. display_sorted_list(list,noofelements)

- **Advantage of Shell sort:**
  - Its only efficient for medium size lists. For bigger lists, the algorithm is not the best choice.
- **Disadvantage of Shell sort:**
  - It is a complex algorithm and its not nearly as efficient as the merge, heap, and quick sorts.
  - The shell sort is still significantly slower than the merge, heap, and quick sorts.

HASHING

HASHING LIST SEARCHES:

- Hashing or hash function is a key-to-address transformation in which the keys map to the addresses in the list.
- Hashing is a key-to-address mapping process
- Diagram

SYNONYM:

- The set of keys that hash to the same location is called a synonym.

COLLIISION:

- A Collision occurs when a hashing algorithm produces an address for a key and that address is already occupied.
- The address produced by the hashing algorithm is known as the home address.
- The memory that contains all of the home addresses is known as the prime area.

HASHING METHODS:

1.DIRECT:

- In direct hashing, the key is the address without any algorithmic manipulation.
- Therefore the data structure must contain an element for every possible key.
- Advantage
  - Applications of direct hashing are limited but can be powerful because there are no synonyms and therefore no collisions.
- Disadvantage
  - Address space is as large as the key space
- Direct hashing is an ideal method but its application is very limited It can be used only for small lists in which the keys map to a filled list

2.SUBTRACTION:

- In this method, the key is transformed to an address by subtracting a fixed number from it.
- It is simple and guarantees that there will be no collisions.
- Limitations:
- limited.It can be used only for small lists in which the keys map to a filled list

SIMILARITY BETWEEN DIRECT HASHING AND SUBTRACTION:

- The direct hashing and substraction methods both guarantee search with no collisions.They are one-to-one hashing methods.i.e. only one key hashes to each address

3.MODULO DIVISION:

- Also known as division remainder, this method divides the key by the array size.and uses the remainder for the address.

  address = key MOD listsize

  when address range from 0 to listsize-1

  Or

address = (key MOD listsize) +1

when address range from 1 to listsize

- This method works with any list size.However a list size that is a prime number produces fewer collisions than other list sizes.Therefore make the array size a prime number.
- Example

4.DIGIT EXTRACTION:

- In this method, selected digits are extracted from the key and used as the address.
- Example:
  - using six-digit employee number to hash to a three-digit address (000–999), we could select the first, third and fourth digits (from the left) and use them as the address.

5.MID SQUARE:

- In midsquare hashing , the key is squared and the address is selected from the middle of the squared numbers.
- Advantage:
  - Entire key is used to calculate the address, reducing chances of collisions
- Disadvantage:
  - The size of the key.Eg. If a key is 6 digits, the product will be 12 digits which is beyond the max integer size of many computers.
- Variation of Mid Square Method
  - Select a portion of the key such as the first 3 digits and then use the midsquare method.

6.FOLDING:

- Two folding methods are used
  - Fold shift
  - Fold boundary
- Fold Shift
  - In fold shift, the key value is divided into parts such that
    the size of the parts = size of the required address
  - The left and right parts are shifted and added with the middle part.
  - If sum>size of the address , discard the leading digits.
  - Fold Boundary:
  - The left and right numbers are folded on a fixed boundary between them and centre number.The two outside values are thus reversed.
  - Examples

7.ROTATION:

- Rotation hashing is not used by itself but is incorporated in combination with other hashing methods.
- Most useful when key are assigned serially.

- A simple hashing algorithm tends to create synonyms when hashing keys are identical except for the last character.
- Rotating the last character to the front of the key minimizes this effect.
- Modula division method do not work well with rotation method.
- Rotation is used only in combination with folding and pseudorandom hashing

8.PSEUDO GENERATION:

- In this the key is used as the seed in a pseudorandom number generator.
- The resulting random number is then scaled into the possible range using modulo-division method.
- A common random number generator is

$$y = ax + c$$

$$x = \text{key}$$

a and c = factors that should be prime numbers since prime numbers minimize collisions

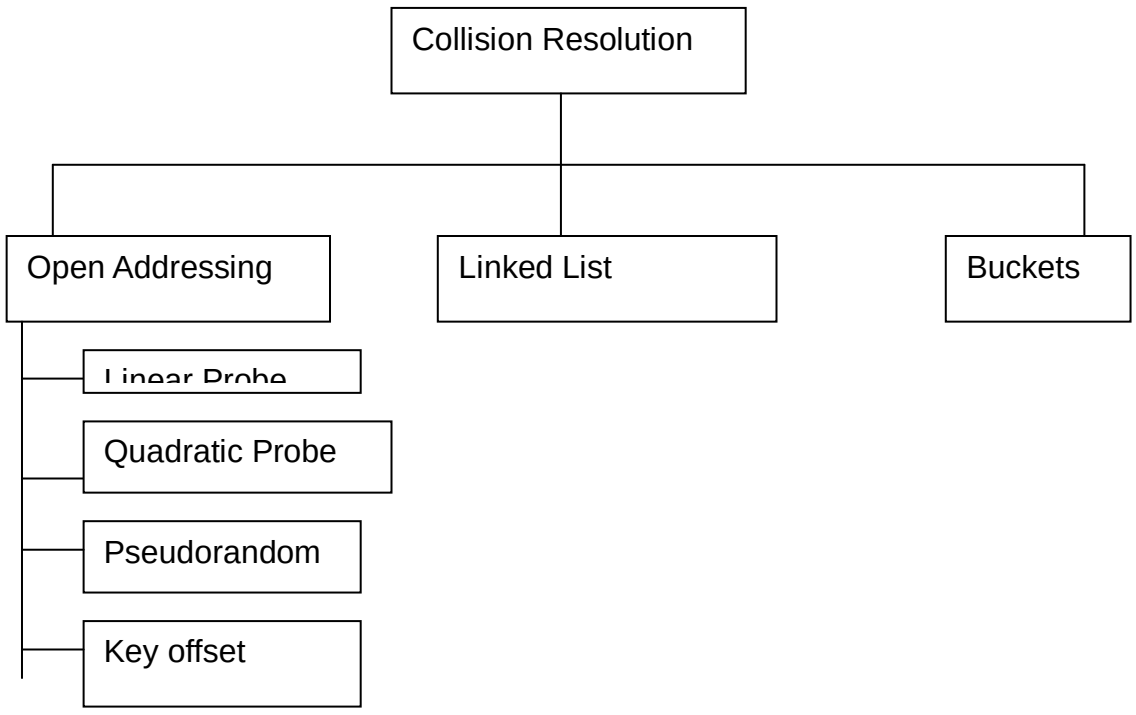
- Example

MULTILICATIVE METHOD:

- The hashing technique uses the following formula:
- h(key) =floor(m * frac(c* key))
  where floor =integer part of real number
  frac(x)= fractional part
  c= 0.618 , yields good theoritical properties
- Disadvantage:
  - Slower than modulo division method

NOTE: All hash functions except direct hashing and subtraction hashing are in such a way that “many keys hash to one address”.

COLLISION RESOLUTION:



Concepts for collision resolution methods:

- Load factor:
  - The load factor(α-alpha)of a hashed list is :
    Number of elements in the list *100
    Number of physical elements allocated for the list
  - Fullness of a file is measured by its load factor
  - When the address space of a relative file gets full, the probability of collision arises dramatically.
  - A load factor 70% or 80% gives reasonable performance.
  - Example. If a file contains n records, the address space should have room for storing 1.25n records (80%)
- Clustering:
  - As the data are added to the list ,some hashing algorithms tend to cause data to group within the list.
  - This tendency of data to build up unevenly across a hashed list is known as clustering.
  - Clustering is usually created by collisions.
  - Two types of clusters exist:
  - Primary clustering:
    - Occur when data cluster around a home address.
      - The collision resolution is based on home address
      - Easy to identify.
      - Primary cluster slows down the operations
      - Example
  - Secondary clustering:
    - The collision resolution is not based on the home address.
    - The collision resolution algorithm spreads the collisions across the entire list.
    - Not easy to identify
    - The time to locate a requested element of data becomes faster.
    - Example

Open Addressing:

- Resolves collisions in the prime area i.e. the area that contains all of the home addresses.
- When a collision occurs, the prime area addresses are searched for an unoccupied element where the data can be placed.

1.LINEAR PROBE:

- Resolves collisions in the prime area i.e. the area that contains all of the home addresses.
- When a collision occurs, the prime area addresses are searched for an unoccupied element where the data can be placed.

- We must ensure that the next collision resolution address lies within the boundaries of the list. Eg: if a key hashes to the last location in the list, adding 1 must produce the address of the first element. Similarly if a key hashes to the first location in the list, subtracting 1 must produce the address of the last element.
- Advantages:
  - Simple to implement.
  - Data tend to remain near their home address.
  - Linear probes tend to produce primary clustering.
  - Linear probes tend to make the search algorithm more complex.

2.QUADRATIC PROBE:

- In the quadratic probe,
  - the increment = the collision probe number squared
  - The new address = collision location + increment
  - Disadvantage:
    - Time required to square the probe number. Therefore instead of multiplication factor, we can use an increment factor that increases by 2 with each probe.
    - It is not possible to generate a new address for every element in the list.

DOUBLE HASHING:

1.pseudorandom collision resolution

- Uses pseudorandom number to resolve the collision.
- In this, rather than use the key as a factor in the random-number calculation, we use the collision address.
- Advantage:
  - Pseudorandom collision resolution have simple solution
  - Produces only one collision resolution path through the list.

2.key offset:

- It is a double hashing method that produces different collision paths for different keys.
- The pseudorandom number generator produces a new address as a function of the previous address, key offset produces a new address as a function of the previous address and the key.
- Example
- Therefore each key resolves its collision at a different address

NOTE:In each method, rather than use an arithmetic probe function, the address is rehashed Both methods prevent primary clustering.

LINKED LIST COLLISION RESOLUTION:

- Major disadvantage of open addressing is that each collision resolution increases the probability of future collisions. This disadvantage is eliminated by Linked list collision resolution.

- Example
- Linked list collision resolution uses separate area to store collisions and chains all synonyms together in a linked list.
- Uses two storage areas :prime area and overflow area :
    - Each element in the prime area contains an additional field- a link head pointer to a linked list of overflow data in the overflow area.
    - When collision occurs , one element is stored in the prime area and chained to its corresponding linked list in the overflow area.
    - The linked list data can be stored in any order, but a LIFO sequence is the most common as it the fastest.

BUCKET HASHING:

- The keys are hashed to buckets. The buckets are nodes that accommodate multiple data occurrences.
- Because a bucket can hold multiple data, collisions are postponed until the bucket is full.
- Problems in bucket hashing:
    - It uses more space because many of the buckets are empty or partially empty at any given time.
    - It does not completely resolve the collision problem.
- Example

APPLICATION OF HASHING:

- Hash tables are good in situations where you have enormous amounts of data from which you would like to quickly search and retrieve information.
- A few typical hash table implementations would be in the following situations:
    - For driver's license record's. With a hash table, you could quickly get information about the driver (ie. name, address, age) given the licence number.
    - For compiler symbol tables. The compiler uses a symbol table to keep track of the user-defined symbols in a C++ program. This allows the compiler to quickly look up attributes associated with symbols (for example, variable names)
    - For internet search engines.
    - For telephone book databases. You could make use of a hash table implementation to quickly look up John Smith's telephone number.
    - For electronic library catalogs. Hash Table implementations allow for a fast find among the millions of materials stored in the library.
    - For implementing passwords for systems with multiple users. Hash Tables allow for a fast retrieval of the password which corresponds to a given username.

STACKS:

- Stores a set of elements in a particular order
- Stack principle: LAST IN FIRST OUT= LIFO
  i.e the last element inserted is the first one to be removed
- Example
  - Stack of plates
  - Stack of coins

QUEUES:

- A linear list of elements in which
  - Inserting can take place at the other end called rear.
  - Deletion can take place only at one end called front.
- Queue principle : FIFO
- Conditions
  - If there is no element in queue/ empty queue
    - FRONT =-1 ; REAR =-1

  - If there is exactly one element in queue
    - FRONT = REAR

- Whenever an element is added to the queue,
  REAR =REAR +1 or REAR++
- If the item being added is the first element, FRONT=0, indicating the queue is no longer empty.
- Four basic operations on queue
  - Enqueue :
    - Queue insert operation is called as enqueue.
    - The data is inserted in the rear.
    - If there is not room to insert another data in the queue , the queue is said to be in an overflow state.
    - Diagram
    - algorithm

      Algorithm Enqueue(struct queue *q, int item)
      To add the data to the queue using array implementation
      pre: struct stack *s : pointer to the queue structure
      Item : data to be pushed in the stack
      post : to add data to the queue

      1.[check if array is full]
      if (q->rear=ARR-1)
      print "\n QUEUE IS FULL"
      2. [increment rear and add item to the array]
      1. q->rear++

2. q->a[q->rear]=item;
3. count++;
3. [if the array is empty
(front =-1) ,set front=0]
if(q->front =-1)
q->front=0

- Dequeue:
  - Queue delete operation is called as dequeue.
  - Data is removed from the front of the queue.
  - If there are no data in the queue, when a dequeue is attempted , then the queue is in an underflow state.
  - Diagram
  - algorithm

Algorithm int dequeue(struct queue *q)
To remove data from the queue using array implementation
pre: struct stack *s : pointer to the queue structure
post:data is removed from the queue and returned to the calling program

1. [if the queue is empty return null]
   if(q->front =-1)
   return NULL
2. [fetch data from the front of the queue]
   1. data=q->a[q->front]
   2. q->a[q->front]=NULL;
3. [if last element was removed from the queue,
   if(q->front=q->rear)
   q->front=q->rear=-1;
   else
   q->front++;
   return data;

- QueueFront:
  - Returns data which is at the front of the queue without changing the contents of the queue.
  - If there are no data in the queue, when a queue front is attempted , then the queue is in an underflow state.
  - Diagram
  - Algorithm
- Algorithm int queuefront(struct queue *q)
- To display the data which is in the front of the queue
- pre: struct stack *s : pointer to the queue structure
- Post : data which is in the front of the queue is returned to the calling program
- QueueRear:
  - Returns data which is at the rear of the queue without changing the contents of the queue.

- If there are no data in the queue, when a queue rear is attempted , then the queue is in an underflow state.
- Diagram
- Algorithm
  - To display the data which is in the rear end of the queue
  - Algorithm int queuerear(struct queue *q)
  - pre: struct stack *s : pointer to the queue structure
  - Post : data which is in the rear end of the queue is returned to the calling program

DISPLAY QUEUE:

- Algorithm displayqueue(struct queue *q)
- To display the contents of the queue

  Pre : struct stack *s : pointer to the queue structure

```
1.[initialize]
x=0
2.[if rear=-1 then the queue is empty]
1.      if(q->rear==-1)
print "QUEUE IS EMPTY"
return;
2. repeat  while(x<=q->rear)
1. printf q->a[x]
2. x=x+1
```

TYPES OF QUEUES:

1.CIRCULAR QUEUE:

- Disadvantage of linear queues:
  - If an item is removed from the linear array, the space remains unutilized. To overcome this, the circular array is used.
  - Algorithm:

    Algorithm ins_circular_q(Struct queue *cq, int item)

    To insert data into circular queue

```
Pre :
Post:
1.  [Check if the queue is full]
                If q->REAR =MAX -1  and
                   q-> Front =0
                        print " circular queue is full"
                         return

2.  If q-> REAR = MAX -1
            q->REAR=0
    else
            increment q->REAR
```

```
if(q->a[q->rear]==0 ||
        q->a[q->rear] ==-999)
          q->a[q->REAR]= item
else
          printf("QUEUE IS FULL");
```

3. [If an item is inserted , then queue is not empty]

```
if q-> front =-1
          q->front =0
```

DELETE:

Algorithm int delete_from_circular_queue(struct queue *q)

1. [Check if the queue is empty]

```
if(q->front ==-1)
      return
```

2. Assign the value of array to the variable data and set the array element to 0

```
data=a[q->front]
a[q->front]=0
```

3.[check if the queue is empty]

```
if(q->front==q->rear)
          q->front=q->rear=-1
else
          if(q->front=MAX-1)
                q->front=0;
          else
                q->front++
return data
```

2.PRIORITY QUEUE:

- Collection of elements where each element is assigned a priority and the order in which elements are deleted and processed.
- The following are the rules:
  - All elements of higher priority is before any element of lower priority.
  - 2 elements with the same priority are processed according to the order in which they are added to the queue.
  - Eg: Time Sharing System
    Programs of high priority are processed first and programs with the same priority form a standard queue
    INSERT:
    Algorithm add_priority_jobs(struct priorityqueue *pq,struct data dt)
    Post: to add jobs in a priority queue

1.[declare variables]

```
struct data temp
```

int i,j

2. if(pq->rear==MAX -1)
   display " QUEUE IS FULL"
   return

3. rear++;
   d[pq->rear]=dt;
   if pq->front==-1
   pq->front =0;

4. Repeat for(i=0;i<=pq->rear;i++)
   1. Repeat for(j=i+1; j<= pq-> rear;j++)
      A. If(priority1 >priority2)
         swap the jobs
      B. If(priority1=priority2)
         1. if(orderno1>orderno2)
            swap the jobs

DISPLAY:

void displaycircularqueue(struct circularqueue *q)

1[initialize]
x=0

2. Repeat while(x<=MAX -1)
   1. display q->a[x]
   2. x++

TYPES OF LINKED LIST:

1.SINGLY

2.DOUBLY

- In linear linked list and circular linked list, one cannot traverse the list backwards.The doubly linked list can be used in such operations.
- Algorithms
- Creating a linked list

Algorithm create_linked_list(struct link *header)

1. [initialize]
   1. header->prev= NULL
   2. header->next= NULL
   3. Point the node to the header
2. Repeat while(ch!='n')
   1. node->next= create a dynamic list of type struct link
   2. node->next->prev=node
   3. node=node->next
   4. Input the data for the node
   5. node->next = null
   6. Ask the user for continuing to create the list

Algorithm display_linked_list(struct link *header)

1. node=point to the first node
2. Repeat while(node)
    1. Display node->info
    2. node=node-> next

Algorithm insert_at_beg(struct link *header)

1. [declare]
    struct link *new1
2. [initialize]
    1. node= point to the first node
    2. prevnode=address of the header
3. new1 =create a dynamic list of type struct link
4. point new1->next to node
   point new1->prev to prevnode
    point prevnode->next to new1
    node->prev=new1
    Enter the value for the new node new1

Algorithm insert_at_end(struct link *header)

1. [Initialize]
    struct link *new1
    node=point to the next node
    prevnode=address of the
        header
2. if(node=NULL)
        display “LIST IS EMPTY"
3. new1= create a dynamic list of type struct link
4. Input the data for the new1 node
5. Repeat while(node)
    1.node=node->next
    2.prevnode=prevnode->next
6. Point new1->next to node
7. Point new1->prev to prevnode
8. Point prevnode->next to new1

Algorithm insert_in_the_middle(struct link *header)

1. [initialize]
    int node_num=1;
    int insert_node;
    node=point to the next node
        prevnode=address of the header
2. Input the location at which the node is to be inserted.

3. if(insert_node<=node_counts)
    1. Repeat while(node)
        if(node_num= insert_node)
            1. new1 =create a dynamic list of type struct link
            2. point prevnode->next to new1
            3. point new1->next to node
            4. point new1->prev to prevnode
                5. point node->prev to new1
6. Enter the value for the new node new1
    else
    1. point node to the next node
    2. point previous to the next node
    node_num++;
    else
display " THERE ARE ONLY 'node_counts' NODES IN THE LIST"

Algorithm delete_first_node(struct link *header)

1. [initialize]
    1. node=point to the next node
    2. prevnode=address of the header
2. If node=NULL
    1. display " THERE ARE NO NODES"
    2. return
  else
    1. prevnode->next= node->next
    2.node->next->prev=prevnode
    3.free the node

3.node=header->next

Algorithm delete_last_node(struct link *header)

1. [initialize]
    node_number=node_counts
    node=pointer to the first node
  prevnode=address of the header
2. If node=NULL
        1. display " THERE ARE NO NODES"
        2. return
  else
    1. Repeat while (node and node_number != 1)
        1. node= point node to the next node
            2. prevnode=point previous to the next node
    2. prevnode->prev->next=node
    3. free the prevnode

Algorithm delete_desired_node(struct link *header)

1.[initialize]
node_number=1
delete_node=0
node=point to the first node
previous=address of the header
2.if(node=NULL)
1. display "LIST IS EMPTY”
3. Enter the node number you want to delete
4. Repeat while(node)
1. if(node_number=
delete_node)
1. previous->next=
node->next
2.node->next->prev
=prevnode
3. free(node)
Else
1. node=point node to the next node
2. previous=point previous to the next node
node_number+3.CIRCUAR:

- Creating a linked list

Algorithm create_linked_list(struct link *header)

1. [initialize]
    1. header->next= NULL
    2. Point the node to the header
2. Repeat while(ch!=‘n’)
    1. node->next= create a dynamic list of type struct link
    2. node=node->next
    3. Input the data for the node
    4. node->next = null
    5. node_counts=node_counts +1
    6. Ask the user for continuing to create the list
3. if(ch=='n')
    1. node->next=header
    2. node=node->next
    3. node->info=node_counts
    4. Display " NO OF NODES IN THE LIST”

Algorithm display_linked_list(struct link *header)

1. node=point to the first node
2. Repeat while(node!=header)
    1. Display node->info
    2. node=node-> next

3. if(node==header)
display “TOTAL NO. OF NODES:”

Algorithm insert_in_the_middle(struct link *header)

1. [initialize]
node_num=1;
node=point to the next node
previous=address of the header
2. Input the location at which the node is to be inserted.
3. if(insert_node<=node_counts)
1. Repeat 1
while(node!=header)
if(node_num= insert_node)
1. new1 =create a dynamic list of type struct link
2. point new1 to node
3. point previous to new1
4. Enter the value for the new node new1
else
1. point node to the next node
2. point previous to the next node
node_num++
2. node=header
node-nfo=node_counts
else
display “ THERE ARE ONLY ‘node_counts’ NODES IN THE LIST”

Algorithm delete_desired_node(struct link *header)

1.[initialize]
node_number=1
delete_node=0
node=point to the first node
previous=address of the header
2. Enter the node number you want to delete
3. Repeat while(node!=header)
1.if node_number=delete_node
1. previous->next= node->next
2. free(node)
else
1. node=point node to the next node
2. previous=point previous to the next node
node_number++

GENERAL TREE:

A general tree(sometimes called as a tree) is defined as a non-empty finite set T of elements, called nodes such that:

The tree contains the root node.

The remaining nodes of the tree form an ordered collection of zero or more disjoint trees T1, T2….TM.

BINARY TREE:

USAGE:

- Used for representing algebraic formulas.
- Used for searching large, dynamic lists.
- Iterative tree traversals

TERMS USED IN BINARY TREE:

- Nodes:
  - Finite set of elements.
- Branches:
  - Finite set of directed lines.
- Degree :
  - Number of branches associated with the node.
- Indegree:
  - Branch directed towards the node.
- Outdegree:
  - Branch directed away from the node.
- Degree of the node= indegree branches + outdegree branches
- The first node is called as the root.
- The indegree of the root is always zero.
- All nodes other than the root must have an indegree of exactly one. i.e they may have exactly one predecessor.
- All nodes in the tree can have zero , one or more branches. i.e they may have outdegree of zero, one or more.
- Leaf node/Terminal Nodes:
  - Any node with an outdegree zero. i.e a node with no successors.
- Internal node:
  - A node that is not a root or a leaf.
- Parent node:
  - If a node has successor nodes. i.e if it has an outdegree greater than zero.
- Child node:
  - If a node has a predecessor. i.e a child node has an indegree one.
- Siblings:
  - Two or more nodes with the same parent.
- Ancestor:
  - any node in the path from the root to the node.
- Descendant:
  - All nodes in the path from a given node to a leaf.

- Path:
  - Sequence of nodes in which each node is adjacent to the next one.
- Level:
  - Distance from the root.
  - Level of the root is zero.
- Height of the tree/depth of the tree:
  - Level of the leaf in the longest path from the root + 1
  - Height of the empty tree is -1.
- Size of the tree:
  - Number of nodes in the tree.

DEFINITION:

- It is a tree in which nonode can have more than 2 subtrees.
- The maximum outdegree for a node is two.
- A node can have zero , one or two subtrees.
- A binary tree may also be defined as follows:-
  - A binary tree is a empty tree
  - A binary tree consists of a node called root, a left subtree and a right subtree both of which are binary trees once again.

PROPERTIES:

- Maximum height:

  $H_{max} = N$ where N = nodes in binary tree.
- Minimum height:

  $H_{min} = [\log_2 N] + 1$
- Minimum nodes:
  - Given the height of the binary tree, H

    $N_{min} = H$
- Maximum nodes:
  - Given the height of the binary tree, H

    $N_{max} = 2^H - 1$
- Balance factor of a binary tree:
  - Difference in height between its left and right subtrees.

    $BF = H_{Left} - H_{Right}$
- In a Balanced binary tree, the height of its subtrees differs by no more than 1(i.e its balance factor is either -1,0,1)

CONVERSION OF GENERAL TREE:

- Identify the branch from the parent to the first child.The branches from parent become the left pointer in the binary tree.
- Connect the siblings with far-left child.
- Remove all unneeded branches from the parent to its children

EXPRESSION TREES:

- An expression is a sequence of tokens
- A token is either an operand or an operator
- An expression tree is a binary tree with the following properties:
  - Each leaf is an operand
  - The root and the internal nodes are operators
  - Subtrees are subexpressions with the root being the operator

AVL TREE:

- Two Russian Mathematicians , G.M Adelson-Velskii and E.M. Landis created the balanced tree known as the AVL tree.
- An AVL tree is a search tree in which the heights of the subtrees differ by no more than 1.
- It is thus a balanced tree.
- An AVL tree is a binary tree that is
  - Either empty or
  - Consists of 2 AVL subtrees T_L and T_R whose heights differ by no more than 1 $| H_L$ and $H_R | <=1$

    Where H_L: height of the left subtree and

    H_R : height of the right subtree

Descriptive identifiers for the balance factors:

- LH : Left High(+1) :
  - Indicates that the left subtree is higher than the right subtree
- EH:Even High(0):
  - Indicates that the left subtree is equal to the right subtree
- RH: Right High(-1):
  - Indicates that the left subtree is Shorter than the right subtree

Balancing trees:

- Whenever a node is inserted/deleted into/from a tree respectively, the resulting tree may become unbalanced.
- Therefore we need to rebalance it.
- Basic Balancing Algorithms:
  - Left of Left:
    - A subtree of a tree that is left high has also become left high
  - Right of Right:
    - A subtree of a tree that is right high has also become right high
  - Right of left:
    - A subtree of a tree that is left high has become right high
  - Left of right:
    - A subtree of a tree that is right high has become left high

Left of Left: When a out-of-balance condition has been created by a left high subtree, balance the tree by rotating the out-of-balance node to the right.

Algorithm rotateRight(root)

set left subtree = right subtree of left subtree

Make left subtree new root

```
AVLNode * rotateRight(AVLNode *root)
    AVLNode *tempptr
tempptr=root->left
    root->left= tempptr->right
    tempptr->right=root
root=tempptr
return
```

Right of right: When a out-of-balance condition has been created by a right high subtree, balance the tree by rotating the out-of-balance node to the left.

Algorithm rotateLeft(root)

set right subtree = left subtree of right subtree

Make right subtree new root

```
AVLNode * rotateLeft(AVLNode *root)
AVLNode *tempptr
tempptr=root->right
root->right= tempptr->left
tempptr->left=root
return
```

Right of left: when a out-of-balance condition is created in which the root is left high and the left subtree is right high,first rotate the left subtree to the left and then rotate the root to the right, making the left node the new root

Pseudocode for balancing left high

Algorithm leftBalance(root)

```
left_subtree=root->left
If(left_subtree high)
    1. rotateRight(root)
    else
    1. rotateLeft(left_subtree)
    2. rotateRight(root)
```

Left of right: when a out-of-balance condition is created in which the root is right high and the right subtree is left high,first rotate the right subtree to the right and then rotate the root to the left, making the right node the new root

Pseudocode for balancing right high

Algorithm rightBalance(root)

```
right_subtree=root->right
If(right_subtree high)
    1. rotateLeft(root)
    else
    1. rotateRight(right_subtree)
    2. rotateLeft(root)
```

INSERTION:

Algorithm AVLInsert(root, newData)

1. if(subtree empty)
    1. Insert newdata at root
    2. return root
2. If(newdata<root)
    1. AVLInsert(left_subtree,newdata)
    2. If(left_subtree taller)
        1. leftBalance(root)

else

    1. AVLInsert(right_subtree,newdata)
    2. If(right_subtree taller)
        1. rightBalance(root)

3. return root

HEAPS

- A heap is a binary tree structure with the following properties:
  - The tree is complete or nearly complete.
  - The key value of each node is greater than or equal to the key value in each of its descendants.
  - Note: whenever the term “heap” is used, it refers to max-heap
  - MAX-A binary tree structure in which the key value in a node is greater than or equal to the key values in all of its subtrees.
- Min-heap:
  - A binary tree structure in which the key value in a node is less than or equal to the key values in all of its subtrees.
- Two basic maintainance operations are performed on a heap
  - Insert a node and
  - Delete a node
- Although it is a tree structure , it is meaningless to traverse it, search it or print it out.
- To implement the insert and delete operations, two basic algorithms are required
  - Reheapup
  - reheapdown
- Reheap Up operation
  - Reorders a “broken” heap by floating the last element up the tree until it is in its correct location in the heap.
  - In this the node must be placed in the last leaf level at the first empty position.
  - If the new node’s key value>key value of the parent, it is floated up the tree by exchanging the child and parent keys
  - Algorithm reheapUp(int values[], int newNode)
  - if(newNode not the root)
  - Parent= (newNode -1)/2
  - If(values[newNode]>values[parent])

- Swap(values[newNode],values[parent])
- reheapUp(values[],parent)

- ReheapDown
  - Reorders a "broken" heap by pushing the root down the tree until it is in its correct position in the heap.
  - This algorithm is used mainly when the root is deleted from the tree.

- Algorithm reheapDown(int values[], int root, int last)

```
1. [declare and initialize]
maxchild, rightchild, leftchild
leftchild=root * 2+1
rightchild=root * 2+2
2.if(leftchild <=last)
        1. if(leftchild ==last)
              maxchild=leftchild
      else
            1. if(values[leftchild]  < values[rightchild])
.                 A. maxchild=rightchild
       else
                  A. maxchild=leftchild
2. if(values[root]<values[maxchild])
        1. swap(values[root],values[maxchild])
        2. reheapDown(values [], maxchild,last)
```

BUILD:

```
Algorithm build_heap(heap , size)
        1.Set walker to 1
        2. Repeat until(walker <size)
                1. reheapUp(heap, walker)
                2. increment(walker)
        INSERT:
        Algorithm insertHeap(heap, last, data)
            1. If (heap full)
                    1. Return false
            2. Increment last
            3. Move data to last node
            4. reheapUp(values[], last)
            5. return last
        DELETE:
        Algorithm deleteHeap(heap, last,dataout)
            1. if(heap empty)
                    1. return false
            2. Set dataout=root data
            3. Move last data  to root
            4. Decrement last
```

5. reheapDown(values[], 0,last)
6. return true

HEAP SORT:

Algorithm heap_sort(A)

1. build_heap(heap,size)
2. Repeat while(n >=0)
    1. Swap(A[0],A[i])
    2. n=n-1
    3. reheapDown(A[],0,n)

M-WAY TREE

- An M-way tree is a search tree in which each node can have from 0 to m subtrees where
  m=order of the tree
- Given a nonempty multiway tree, the following properties:
  - Each node has 0 to m subtrees
  - Given a nod with k<m subtrees, the node contains k subtree pointers, some of which may be null and k-1 data entries
- The key values in the first subtree are all less than the key in the first entry.
- An M-way tree is a search tree in which each node can have from 0 to m subtrees, where m= B-tree order.
- Properties of M-way tree:
  - Each node has 0 to m subtrees
  - The key values in the first subtree < key value in the first entry; key values in the other subtrees >= key value in their parent entry.
  - Keys of the data entries are ordered key1<= key2 <=……key n
  - All subtrees are themselves multiway trees
  - Diagram
- Advantages
  - The height of the tree is greatly reduced.
- Disadvantages
  - it is not balanced.

B TREES

- B-tree is an m-way search tree with the following additional properties:
  - The root is either a leaf or it has 2…m subtrees
  - All internal nodes have atleast m/2 non-null subtrees(i.e minimum) and at most m non-null subtrees
  - A leaf node has atleast m/2 -1 non-null subtrees(i.e minimum) and at most m-1 non-null subtrees

GRAPHS

TERMINOLOGIES:

- Graph:
  - Is a collection of nodes called vertices and a collection of line segments connecting pair of vertices called lines.
- Directed graph or Diagraph:
  - Is a graph in which each line has a direction to its successor. The lines in a directed graph are known as arcs.
- Undirected graph:
  - is a graph in which there is no direction on the lines known as edges.
- Adjacent vertices:
  - Two vertices are said to be adjacent vertices is there exists an edge that directly connects them.
- Path:
  - it is a sequence of vertices in which each vertex is adjacent to the next one.
- Cycle:
  - it is a path consisting of atleast 3 vertices that starts and ends with same vertex.
- Loop:
  - Special case of a cycle in which a single arc begins and ends with the same vertex.
- Strongly connected:
  - a graph is strongly connected if there is a path from each vertex to every other vertex in the digraph.
- Weakly connected:
  - a graph is weakly connected if at least two vertices are not connected.
- Disjoint graph:
  - If two graphs are not connected
- Degree of a vertex:
  - Number of lines incident to it.
- Outdegree of a vertex:
  - In a diagraph, it is the number of arcs leaving the vertex.
- Indegree of a vertex:
  - In a diagraph, it is the number of arcs entering the vertex.

APPLICATIONS:

- Cities and highways connecting them form a graph.
- Components on a circuit board with connections among them form a graph.
- An organic chemical compound can be considered as a graph
- Message transmission in a network

Six primitive graph operations:

- Add vertex

- Delete vertex
- Add edge
- Delete edge
- Find vertex
- Traverse graph

CREATE NODE:

Algorithm create_node return newly created node

```
struct node *create_node()
1.[Declare]
   struct node *temp
2. temp=create a dynamic node
   temp->next=NULL
   temp->edgeptr=NULL
   temp->status=0
   return temp
```

INSERT NODE:

algorithm insnode(char data)

```
1.[ Declare]
        static struct node *temp
2. if(start=NULL)
     1. start=create_node()
     2. start->data=data
     3. temp=start
   else
 1. temp->next=create_node()
   2.temp=temp->next
   3. temp->data=data
```

FIND A NODE:

Algorithm struct node *findnode(char data)

```
1. [Declare and initialize]
     struct node *temp
     temp=start
2. Repeat while(temp->data!=data&&temp!=NULL)
        temp=temp->next
   return temp
```

CREATE EDGE:

Algorithm struct edge *create_edge()

1. struct edge *temp
2. temp=create edge dynamically
3. return temp

INSERT EDGE:

Algorithm insedge(char source,char dest)

1. struct node *locsource,*locdest
2. locsource=findnode(source)
3. locdest=findnode(dest)
4. locsource->edgeptr
   =addedge
   (locsource->edgeptr,locdest)

ADD EDGE:

Algorithm struct edge *addedge
(struct edge* startedge,struct node *locdest)

1. struct edge *temp=startedge;
2. if(temp=NULL)
   1. startedge=create_edge()
   2.startedge->datanode=locdest
   3.startedge->next=NULL
   4. Display
   startedge->datanode->data

   else
   1. Repeat while
   (temp->next!=NULL)
   1. temp=temp->next
   2. temp->next= create_Aedge()
   3. temp=temp->next
   4. temp->datanode=locdest
   5. temp->next=NULL;
   6. Display
   Temp->datanode->data

3. return startedge

DISPLAY GRAPH:

Algorithm display_graph()

1. [Declare]
   struct node *tempnode
   struct edge *tempedge
   tempnode=start
2. Repeat A,B,C,D while(tempnode!=NULL)
   A. Display tempnode->data
   B. tempedge=tempnode->edgeptr
   C. Repeat 1,2while(tempedge!=NULL)
   1. Display tempedge->datanode->data
   2. tempedge=tempedge->next
   D. tempnode=tempnode->next

GRAPH TRAVERSALS:

1.DEPTH-FIRST TRAVERSALS:

- All of a vertex's descendants are processed before we move to an adjacent vertex.
- The preorder traversal of a tree is a depth-first traversal
- Working:
  - The DFS start by processing the first vertex of the graph.
  - Select any vertex adjacent to the first vertex and process it.
  - Repeat until a vertex with no adjacent entries is reached

- This logic requires stack to complete the traversal.

ALGORITHM:

Algorithm dfs(adj[][], vertex)
where adj[][] -adjacency matrix
v = vertex
1. Intialize visted[MAX] =0
2. Enter Initial vertex v
3. Display initial vertex v
4. set visited[v]=1
5. Repeat A. and B. for k=1 to k<vertices
A. Repeat a for j=1 to j<=vertices
a. if(adj[v][j]=1 && visited[j]=0)
1. visited[j]=1
2. stk[top]=j
3. top= top + 1

B. v=stk[top]
top= top -1
display v

2.BREADTH-FIRST TRAVERSALS:

- All adjacent vertices of a vertex are processed before going to the next level.
- The breadth-first traversal of a graph follows the same concept of the breadth-first traversal of a tree.
- Working:
  - Start with a starting vertex and after processing , process all of its adjacent vertices.
  - When all of the adjacent vertices have been processed, we pick the first adjacent vertex and process all of its vertices then the second adjacent vertex and process all of its vertices and so on.
- This logic requires queues to complete the traversal

ALGORITHM:

1. Intialize visted[MAX] =0

2. Enter Initial vertex v.
3. Display initial vertex v
4. set visited[v]=1
5. Repeat A. and B. for k=1 to k<vertices
   A. Repeat a for j=1 to j<=vertices
      a. if(adj[v][j]=1 && visited[j]=0)
         1.visited[j]=1
         2. rear =rear +1
         3. queue[rear]=j
         4. if(front==-1)
            front=0

   B.v=queue[front]
   front= front +1
   display v

REPRESENTATION:

- To represent a graph , two sets are stored:
  - Ist set represents the vertices of the graph
  - IInd set represents the edges of the graph
- The most common structures used to store these sets are
  - Arrays
  - Linked List

ADJACENCY:

- The adjacency matrix uses a vector(one-dimensional array) for the vertices and a matrix (two-dimensional array) to store the edges.
- If two vertices are adjacent i.e there is an edge between, the matrix intersect has a value 1.
- If there is no edge between them , the intersect is to value 0.
- Disadvantage:
  - The size of the graph must be known before the program starts.
  - Only one edge can be stored between any 2 vertices

- void adjacency(adj[M][N], vertices, edges, graphtype)
  1. [Initialize]
     Repeat A. for i=1 to i<=vertices
        A. Repeat for j=1 to j<=edges
           adj[i][j]= 0
  2. Repeat A. for i=1 to i<=edges
     A. Enter start edge and end edge
     B. if(graphtype=1)
        adj[a][b]=1
        adj[b][a]=1
     if(graphtype=2)

adj[a][b]=1

3. Display Adjacency Matrix :

Repeat A. for i=1 to i<=vertices

A. Repeat for j=1 to j<=vertices

display adj[i][j]

ADJACENCY LIST:

- The vertex list is a singly-linked list of the vertices in the list. Depending on the application , it could also be implemented using doubly-linked list or circular linked list.
- The edges are stored in single linked list.
- The pointer at the left of the list links the vertex entries together.
- The pointer at the right in the vertex list is a head pointer to a linked list of edges from the vertex.

NETWORK:

- A network is a graph whose lines are weighted.It is also known as weighted graph.
- The meaning of weights depends on the application.
- Eg:
  - An airline might use a graph where
    - Nodes: cities
    - Edges: routes
    - Edge's weights:miles between 2 cities

SPANNING TREE:

- A spanning tree is a tree that contains all of the vertices in the graph.
- Minimum spanning tree is chosen with following properties:
  - Every vertex is included
  - Total edge weight= minimum cost between vertices
- Minimum spanning tree algorithms:
  - Kruskal's algorithm
  - Prim's algorithm

KRUSKAL'S ALGORITHM:

- The edges considered for the inclusion in the spanning tree is as per the increasing order of the cost of different edges.
- An edge is included in the spanning tree only if it does not form a cycle with the edges that are already present in the spanning tree.
- Algorithm
- Kruskal's algorithm can be shown to run in
- $O(E \log E)$ time, or equivalently, $O(E \log V)$ time, all with simple data structures.

PRIM'S ALGORITHM:

- Prim's algorithm is an algorithm that finds a minimum spanning tree for a connected weighted undirected graph.

- This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Prim's algorithm is an example of a greedy algorithm

The array structure is used to store the cost of each node

Prim(Graph g)
Pre A: Array to store vertices of the graph

1. Set A =Vertices of the graph g
2. Repeat for each vertex u
    1. Cost[u] =∞
    2. Vertex_list=NIL
3. Repeat a and b while array A not empty
    a. Find node with the smallest key and remove from A
    b. Repeat for each vertex v belonging to adjacent[u]
        1. If(weight[u,v]<cost[u] then
            1. Set vertex_list[v]=u
            2. Set cost[u]=w(u,v)

2. A simple implementation using an adjacency matrix graph representation requires O(V^2) running time.
3. Efficiency of Prim's algorithm can be shown to run in time O(E log V) where E is the number of edges and V is the number of vertices.

SHORTEST PATH ALGORITHMS:

- Another common application used with graph requires that we find the shortest path between two vertices in a network
- Shortest path algorithms
    - Dijkstra's algorithm
    - Warshall' algorithm

RELAXATION TECHNIQUE:

- This technique consists of testing whether we can improve the shortest path found so far.
- The relaxation technique may or may not decrease the value of the shortest path estimate.
- Algorithm relax(u,v,w)

1. If (d[u] +cost(u,v)< d[v])
    a. d[v]= d[u] + cost(u,v)
    b. vertex_list[a] = u

DIJKSTRA ALGORITHM:
This algorithm solves the shortest path problem when all edges have non-negative weights.

Dijkstra algorithm(G, cost, S)

1. [initialize]
    set S={} S will contain vertices of final
    shortest path cost from S

Queue Q=V(G) vertices of the graph

2. repeat a,b,c while Q not empty
   a. set u= extract_min(Q) pull out new vertex
   b. set s= s U {u}
   c. repeat 1 for each vertex v adjacent to u
      1. relax(u,v, cost)

Analysis Dijkstra's algorithm runs O(E log V) times.

FLOYD-WARSHALL:

- Basic concept:
  - If for a path(u,v) and its length estimate D[u][v], if we detour(go through) via W and shorten the path, then it should be taken.
  - This translates into the following equation:

  D[u][v]=min(D[u][v], D[u][w] + D[w][v])

- Algorithm floyd-warshall(W)
  1. set n=row[w]
  2. set $D^{(0)}$ =W
  3. repeat A. for k= 1 to n
     A. repeat 1. for i = 1 to n
     1. repeat for j=1 to n
        set $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
  4. return D(n)

ANALYSIS : the algorithm run $O(n^3)$ times