Data Structure

⇒ A data structure is a way to store and organize data in a computer, so that it can be used efficiently.

We talk about data structures

1) Mathematical / Logical models. OR Abstract data types
2) Implementation

⇒ Abstract data types (ADTs)

- define data & operations but no implementation

1) Logical view 2) Operations 3) Cost of operations
4) Implementation

1) ⇒ List as ADT

List
- Store a given number of elements of a given data types
- O(1) Write / modify element at a position.
- Read element at a position (Array).

List
- O(1) empty list has size 0
- O(n) insert
- O(n) remove
- O(1) count
- O(n) Read / modify element at a position
- specify data type

When array is full, create a new larger array, copy previous array into the new array.
Free the memory for the previous array.

Date: / / Page no:

2) Linked List

```
struct node
{
    int data;      // 4 byte
    Node* next;    // 4 byte
}
```

head = first node
⇓
Address of 1st node / head.

⇒ We not access any of the element in constant time. while array requires constant time

⇒ Address of the node give us complete access of the complete list

⇒ Access of n elements ⇒ Tan ⇒ O(n)
Insertion ⇒ O(n), Delete ⇒ O(n)

⇒ Array Vs Linked List

	Array	Linked List
1) cost of accessing an element	constant time ⇒ O(1)	Averge case ⇒ O(n)
2) Memory requirement	fixed size ⇒ memory may not be available as one large block	no unused memory but extra memory needed for pointer variable. memory may be available as multiple small blocks
3) cost of inserting/deleting an element	At beginning - O(n) at the end - O(1) O(n) ⇒ array is full at middle ⇒ O(n)	O(1) ⇒ beginning O(n) ⇒ end O(n) ⇒ middle
4) Easy of use	✓	✗

free

Date: / / Page no:

⇒ Creating an linked list

```
struct Node{
    int data;
    struct Node * Link
}
Node *A
  A = NULL
```

```
⇒ New Node();
Node* temp = (Node*) malloc(size of(Node))
(*temp).data = 2
(*temp).Link = NULL
A = temp
```

⇒ Insertion of Node at End:-

```
temp = (Node*) malloc(size of (Node))
temp → data = 4
temp → lnk = NUL.

Node* temp1 = A
while (temp1 → Link != NULL)
{
    temp1 = temp1 → Link;
}
temp1 → Link = temp
```

} traverse

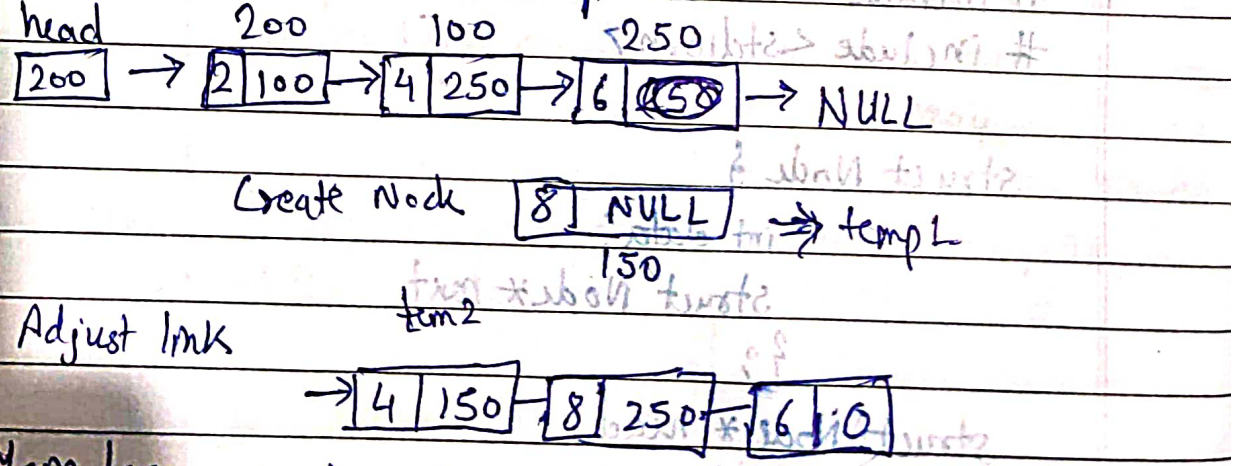
⇒ Insertion of Node at beginning

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

struct Node {
        int data;
        struct Node* next
    };
struct Node * head;
```

```
int main()
{
  head = NULL; //empty list
  printf("How many numbers? \n");
  int n, i, x;
  scanf("%d", &n);
  for(i=0; i<n; i++)
  {
    printf("Enter the number \n");
    scanf("%d", &x);
    Insert(x);
    Print();
  }
void insert(int x)
{
  Node* temp = (Node*) malloc(sizeof(struct Node));
  temp -> data = x
  temp -> next = NULL
  if(head != NULL)
  { temp -> next = head }
    head = temp;
  head = temp;
```

⇒ Insert a node at nth position



Main Logic ⇒ temp1 -> next = temp2 -> next
temp 2 -> next = temp1

Node insertPos

Function :-

```
node insertPos(node x)
{ node a, b;
  int loc, i.
  printf("Enter the location");
  scanf("%d", &loc);
  a = accept();  ⇒ create node
  if (x == NULL)
      x = a;
  else
  { b = x
    for (i = 1; i < loc - 1; i++)
    {
      b = b->next
    }
    a->next = b->next
    b->next = a
  }
  return x;
}
```

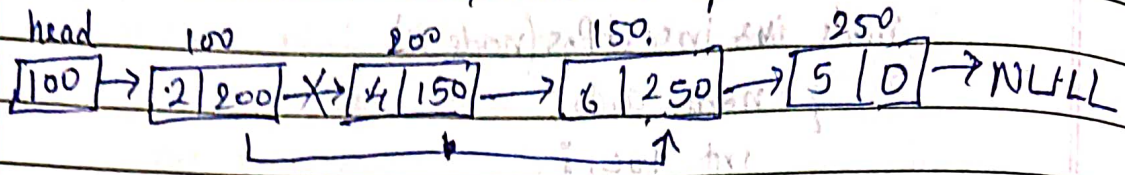
⇒ Print an Linked list

```
Void Print()
{
  struct Node * temp = head;
  while (temp != NULL)
  {
    print("%d", temp->data);
  }
  print("\n");
}
```

Date: / / Page no.

⇒ Delete a node at n^{th} position.

1) fix the links, 2) Free the space



Main logic ⇒ b→next = a→next

a→next = NULL (Not compulsory)

free(a);

```
node delete (node first)
{ node a,b;
  int loc, i;
  printf ("Enter the location:");
  scanf ("%d", &loc);
    a = first
  if (first == NULL)
  { print :=
      print ("List is empty");
      return first;
  }
  else
  { i = 1
    while for (i=1, i<loc, i++)
    {
      b = a
      a = a→next
    } i++;
  }
    free(a)
  b→next = a→next
  a→next = NULL
  free(a);
```

OR

```
for (i=1; i<loc-1; i++)
{
  temp1 = temp1→next
}
struct Node*
temp2 = temp1→next
temp1→next = temp2→next
free(temp2)
```

→ R Reverse a Linked List

1) Iterator

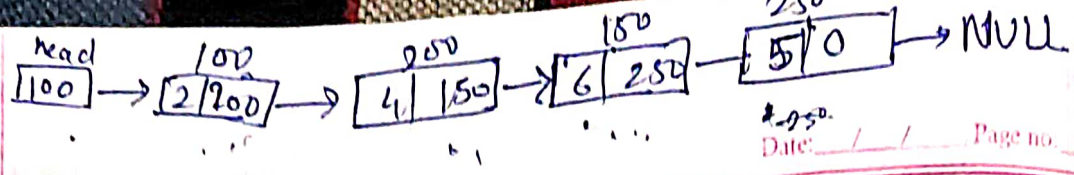
```
Struct Node
  { int data;
    struct Node* next;
  }

Struct Node* head;

struct Node* Reverse(struct Node * head)
  {
    struct Node **current, *prev, *nex;
     current = head
     prev = NULL
      while (current != NULL)
       {
        next = current->next;
        current->next = prev
          prev = current
          current = next
       }
        head = prev;
        return head;
       }
```

ii) Using Recursion

```
struct Node *head;
  Void Reverse(struct Node *p)
    { if (p->next == NULL)
      { head = p;
        return;
      }
```



Reverse (p→next);
struct Node *q = p→next; } ⇒ p→next→next = p;
q→next = p
p→next = NULL

⇒ Print Using Recursion

Forward	Reverse
void Print(struct Node *p){	if(p==NULL) return;
{ if(p==NULL) return;	Print(p→next);
printf("%d", p→data);	printf("%d", p→data);
Print(p→next);	

⇒ Doubly Linked List

```
struct Node
{
  int data;
  struct Node *next;
  struct Node *prev;
};
```

Advantage ⇒ Reverse Look-up (only one pointer)
⇒ 2. Implementing Operation is easy
⇒ .

Disadvantage ⇒ Extra memory for pointer to previous node

Implementation

```
struct Node* head;
```

```
void
struct Node *GetNewNode(int x)
  {
    struct Node* newNode =(struct Node*) malloc(sizeof(struct Node));
      newNode->data = x;
      newNode->prev = NULL;
      newNode->next = NULL;
        return newNode;
  }

void InsertAtHead(int x){
  { struct Node* newNode = GetNewNode(x);
    if(head == NULL)
      { head = newNode;
        return;
      }
    head->prev = newNode
    newNode->next = head;
      head = newNode;
  }

void InsertAtTail(int x)
  {
    struct Node * temp =head;
     struct Node *newNode = GetNewNode(x);
     if(head == NULL)
     {
       head = newNode;
         return;
     }
    while(temp->next != NULL) temp = temp->next
    temp->next = newNode;  newNode->prev =
                             temp;
```

Date: / / Page no:

```
void ReversePrint () {
    struct Node* temp = head;
    if (temp == NULL) return;
    while (temp -> next != NULL) {
        temp = temp -> next; }

    while (temp != NULL)
    {
        print ("%d", temp -> data);
        temp = temp -> prev;
    }
}

void Print () {
    struct Node *temp = head;

    while (temp != NULL)
    {
        print (temp -> data);
        temp = temp -> next;
    }
```

3) Stack

⇒ ADT

→ Stack ⇒ A list with the restriction that insertion and deletion can be performed only from one end, called the top.

First In Last Out (FIFO) or Last In First Out (LIFO)

→ Operation :-

1) Push(x) 2) Pop() 3) Top() 4) Is Empty

↓ O(1).

→ Application → 1) Function call / Recursion

2) Undo in an editor

3) Balanced Parenthesis

⇒ Array Implementation

```
int A[10]
top ← -1  //empty stack
Push(x)
{
  top ← top+1
  A[top] ← x
}
```

A[++top] = x

```
Pop()
{
  top ← top-1
}
```

top--

```
Top()
{
  return A[top]
}
```

```
IsEmpty()
{
  if(top == -1) return true
  else return false
}
```

Limitation ⇒ We can have a situation where stack would consume the whole array so top will be equal to highest index in the array. A further push will not be possible. it will result in

(a) (n) O(n)

overflow. ⇒ create a larger array copy all elements to new array.

⇒ Linked List Implementation

Insert / delete

1) at end of list ✗ [time complexity = O(n)]

⇒

2) At beginning (head) ⇒ (O(1))

```
void Push(int x)
{
  struct Node* temp = (struct Node*) malloc(sizeof(struct Node));
  temp->data = x;  temp->link = top;
  top = temp;
}

void Pop()
{
  struct Node *temp;
  if (top == NULL) return;
  temp = top
  top = temp->link;
  free(temp);
}
```

⇒ O(1) ⇒ does not cause overflow

Date __/__/__ Page no.

⇒ Using Stack to reverse

i) Reverse a string

```
#include <stack>  //stack from standard template library (STL)
void Reverse(char *c, int n)
{
    stack<char> S;
    //loop for push
    for(int i=0; i<n; i++)
    { S.push(c[i]); }

    //loop for pop.
    for(int i=0; i<n; i++)
    {
        c[i] = S.top();  //overwrite the character at index i
        S.pop();  //perform pop.
    }
}
```

```
main() {
  char c[51];
  gets(c);
  Reverse(c, strlen(c));
  printf(" print c
```

Time & Space Complexity = O(n)

⇒ We can also do different approch in which the start & end index in the string initially, in any var store in any variable, & while i < j we can swipe the characters at these position, & increment i, j (these give time complexity = O(n) & space complexity (1)]

Better approach

ii) Reverse a Linked List

i) Iterative Solution ⇒ Time - O(n) Space = O(1)

ii) Recursive Solution ⇒ Time - O(n) Space = O(n) (Implicit Stack)

iii) Explicit Stack

```
void Reverse()
{
  if (head == NULL) return;
  stack <struct Node*> S;
  Node* temp = head;

  while (temp != NULL) {
        S.push(temp);
        temp = temp->next;
  }

  temp = S.top();
  head = temp;
  S.pop();
  while (!S.empty()) {
        temp->next = S.top();
        S.pop();
        temp = temp->next;
  }
  temp->next = NULL;
}
```

⇒ Check for balanced Parentheses [() or [] or {}]

1) Every opening parathesis must find a closing counterpart to its right and every closing parathesis must find a opening counterpart to its left

2) A parenthesis must can close only when all the

parathesis opened after it closed

⇒ ~~last opened first closed~~
⇒ Last unclosed, ~~first~~ etc closed

Sol ⇒ scan from left to right
⇒ if opening symbol, push it into a stack
⇒ if closing symbol & top of stack opening of same type pop
⇒ should end with an empty list

bool AreePair(char open, char close)
{
~~if (opening == '(' && closing == ')') return true;~~
~~else if~~
~~if~~

Code

bool ArePair (char open, char close) ~~re~~
{
if (open == '(' && close = ')') return true
else if (open == '{' && close = '}') return true
else if (open == '[' && close = ']') return true

return false
}

Date: / / Page no

```
bool AreParanthesesBalanced (string exp)
{
    stack <char> S;
    for (int i=0; i< exp.length (), i++)
    {
        if (exp[i] == '(' || exp[i] == '{' ||
            exp[i] == '[')
                S.push (exp[i]);
        else if (exp [i] == ')' || exp[i] == '}' ||
                 exp[i] == ']')
        {
            if (S.empty () || ! ArePair (S.top (), exp [i]))
                return false;

            else S.pop ();
        }

    return S.empty () ? true : false;
```

** Infix, Postfix, Prefix

1) Infix ⇒ <operand> <operator> <operand>

Order of Operations

1) Parentheses
2) Exponents ⇒ (right to left)
3) Multiplication & division ⇒ (left to right) } Associativity
4) Addition & subtraction ⇒ (left to right)

Date: / / Page no.

2) Prefix
(Polish notation) => in 1324

<operator> <operand> <operand>

ex => a+b*c = +a(*bc) a+a*bc

3) Postfix
(Reverse polish notation) in 1950 ; ~~easy~~ easiest to parse & least costly in terms of time & memory.

<operand> <operand> <operator>

ex → a+b*c = abc*+

=> Evalution of Prefix & Postfix Expressions

=> A*B+c*d-e
{(A*B)+(c*d)}-e
{(AB*)+(cd*)}-e
{AB*cd*+}-e
AB*cd*+e-

Evaluate Postfix(exp)
{
Create a stack S.
for i ← 0 to length(exp)-1
{
if exp[i] is operand
Push(exp[i])
else if exp[i] is operator
{
op2 ← pop()
op1 ← pop()
result ← Perform(exp[i], op1, op2)
Push(res)
}
return top of stack
}

Prefix is same as Postfix but we can have to start from right while scanning string & first element pop = first operand

111-20 1816-20

⇒ Code ⇒

```
int EvaluatePostfix(string expression)
{
    stack<int> S;
    for(int i=0; i<expression.length(); i++)
    {
        if(expression[i]==' ' || expression[i]==',')
            continue;
        else if(IsOperator(expression[i]))
        {
            int operand2 = S.top(); S.pop();
            int operand1 = S.top(); S.pop();

            int result = PerformOperation(expression[i],
                                          operand1, operand2);
            S.push(result);
        }
        else if(IsNumericDigit(expression[i]))
        {
            int operand = 0;
            while(i<expression.length() &&
                  IsNumericDigit(expression[i]))
            {
                operand = (operand*10)+(expression[i]
                                        -'0');
                i++;
            }
            i--; //extra increment happend in loop
                 need to adjust
            S.push(operand);
        }
    }
    return S.top();
}
```

```
bool IsOperator (char c)
{
    if (c == '+' || c == '-' || c == '*' || c == '/')
        return true;

    return false;
}

bool IsNumericDigit (char c)
{
    if (c >= '0' && c <= '9') return true;
    return false;
}

int PerformOperation (char operation, int operand1, int operand2)
{
    if (operation == '+') return operand1 + operand2;

    else if (operation == '-') return operand1 - operand2;

    else if (operation == '*') return operand1 * operand2;

    else if (operation == '/') return operand1 / operand2;

    else { cout << "error";
        return -1;
    }
}
```

Date / / Page no.

* Infix to Postfix

⇒

```
string InfixToPostfix(string expression)
{
    stack <char> S;
    stack postfix;
    string postfix = " ";

    for(i=0; i<expression.length(); i++)
    {
        if(expression[i] == ' ' || expression[i] == ',')
            continue;

        else if (IsOperator(expression[i]))
        {
            while(!S.empty() && S.top() != '(' &&
                  HasHigherPrecedence(S.top(), expression[i]))
            {
                postfix += S.top();
                S.top S.pop();
            }
        }
        else if (Isoperand(expression[i]))
        {
            postfix += expression[i];
        }
        else if (expression[i] == '(')
        {
            S.push(expression[i]);
        }
```

```
else if (expression[i] == ')')
    {
    while (!S.empty() && S.top() != '(')
        {
        postfix += S.top();
        S.pop();
        }
    S.pop();
    }
while (!S.empty())
    {
    postfix += S.top();
    S.pop();
    }
return postfix;
}
```

on[i]))

⇒

```
int HasHigherPrecedence (char op1, char op2)
    {
    int op1Weight = GetOperatorWeight(op1);
    int op2Weight = GetOperatorWeight(op2);

    if (op1Weight == op2Weight)
        {
        if (IsRightAssociative(op1)) return false;
        else return true;
        }
    return op1Weight > op2Weight ? true : false
    }
```

Date: / / Page no:

```
int GetOperatorWeight (char op)
{
    int weight = -1;
    switch (op)
    {
        case '+': case '-': weight = 1; break;
        case '*': weight = 2; break;
        case '/': weight = 2; break;
        case '$': weight = 3; break;
    }
    return weight;
}

int IsRightAssociative (char op)
{
    if (op == '$') return true;
    return false;
}
```

Date: / / Page no:

Introduction to ~~Queues~~ Queues

⇒ Queue ADT ⇒ First In Fast Out (FIFO)

↳ A list or collection with the restriction that insertion can be performed at one end (rear) and deletion can be performed at other end (front).

⇒ Operations :-

1) Enqueue (x) or Push (x)
2) Dequeue () or Pop()
3) front () or Peek ()
4) Is Empty ()
~~5) Is fu~~

} constant time or O(1)

⇒ Applications

1) Printer queue
2) Process sheduling
3) simulating wait

⇒ Array Implementation

```
#define Max_Size 101
Class Queue
{ private ~~A[Max_size];~~
  ~~int~~ private:
        int front, rear;
        int A[Max_Size];

  public:
        Queue() { front = -1;
                  rear = -1; }
```

```
bool isEmpty ()
{
    return (front == -1 && rear == -1);
}
bool isFull()
{
    return (rear+1) % Max_Size == front ? true : false;
}
void Enqueue (int x)
{
    if (IsFull())
    {
        cout << " Error : Queue is Full!";
        return;
    }
    if (IsEmpty())
    {
        front = rear = 0;
    }
    else
    { rear = (rear+1) % MaxSize;
    }
    A[rear] = x;
}
```

```
void Dequeue()
{
  if (IsEmpty()):
  {
    cout<<"Error";
    return;
  }
  else if (front == rear) rear = front = -1;
  else front = (front+1) % Max_Size;
}
int Front()
{
  if (front == -1) { cout<<"error"; return -1; }
  return A[front];
}
```

⇒ Linked List Implementation

```
struct Node {
    int data;
struct Node *next;
}
```

```
Struct Node *front = NULL
struct Node *rear = NULL
```

```
void Enqueue(int x) {
struct Node *temp = (struct Node*)
        malloc(sizeof(
        struct Node));
temp->data = x;
temp->next = NULL
if (front == rear == NULL)
{
  front = rear = temp;
  return
}
```

Date: / / Page no:

```
rear -> next = temp;
  rear = temp;
}

Void Dequeue() {
    struct Node * temp = front;
    if (front == NULL) {
        print (list is Empty);
        return;
    }
    if (front == NULL && rear == NULL)
        front = rear = temp NULL
    if (front == rear)
    { front = rear = NULL }
    else { front = front -> next; }
    free (temp);
```

Indr Introduction to Trees

⇒ How to decide which data structure to use?
i) What needs to be stored?
ii) Cost of operations.
iii) Memory usage.
iv) Ease of implementation

⇒ Tree
root = no parents
Parents
Children.
Sibling = have same parents.
Leaf = have no child.

Subtree
Tree
↳ recursive data structure
- N nodes has n-1 edges.

⇒ Depth of x = length of path from root to x OR
Number of edges in path from root to x.

⇒ Height of x = No of edges in longest path from x to a leaf

⇒ Binary tree = have at most 2 children.

Application = 1) Storing naturally hierarchical data
eg → file system

2) Organise data for quick search, insertion, deletion.
3) Trie - dictionary
4) Networks routing algorithms

Date: / / Page no:

Binary trees

i) Binary tree = each node can have at most 2 children.

ii) Strict/Proper Binary tree.
Each node can have either 2 or 0 children

iii) Complete Binary tree
All levels except possibly the last are completely filled and all nodes are as left as possible.

Max no. of nodes at level $i = 2^i$ [$i = 0, 1, 2, 3 \ldots$ root $i = 0$]

iv) Perfect Binary tree
All levels are completely filled & all leaf node are at the same level.

⇒ Maximum no. of nodes in a Binary tree with h

$= 2^0 + 2^1 + \ldots + 2^h$

$= 2^{h+1} - 1$

$n = 2^{(\text{no of level})} - 1$

Level start with zero
height of root is 0.
height of empty tree is -1

$\therefore h = \log_2(n+1) - 1$

$\therefore$ height $= \log_2(n+1) - 1$ (Perfect Binary tree)

Height of complete binary tree $= \lfloor \log_2 n \rfloor$

Height of empty tree = -1,

⇒ Min-height = $\lfloor \log_2 n \rfloor$

⇒ Max-height = $n-1$

⇒ Height of Perfect Binary tree = $\log(n+1) - 1$.

⇒ Balanced Binary tree. ⇒ The difference between height of left and right subtree for every node is not more than K (mostly 1).

⇒ Implement Binary tree.

a) dynamically created nodes.

b) array (used for heap)

↓

For Binary tree for node at index i

left child index = $2i + 1$

right child index = $2i + 2$

~~Parents = $\lfloor 1/2 \rfloor$~~ ~~Parents = $\lfloor i/2 \rfloor - 1$~~

Parents = $(i-1)/2$

Binary search ⇒ O(logn)
O(logn).

Date / / Page no.

Binary Search Tree

Binary Search Tree.

A binary tree in which for each node, value of all the nodes in left subtree is lesser or equal and value of all the nodes in right subtree is greater.

Time ~~com~~ Complexity ⇒ Average = O(Logn)
Worst = O(n)

⇒ Implementation in C/C++

```
Struct Node {
    int data;
    Node *left;
    Node* right;
}
```

Nodes will be created in heap using malloc function in 'C' or new in C++

```
Node * root;
root = NULL;
```

Code ⇒

```
struct BstNode {
    int data;
    BstNode* left;
    BstNode* right;
}
```

```
BstNode* CreatNewNode(int data)
{   BstNode* newNode = new BstNode();
    newNode->data = data;
    newNode->left = NewNode->right = NULL;
}   return newNode;
```

```
BstNode* Insert (BstNode *root, int data).
{
    if (root == NULL) {
            root = GetNewNode(data);
        }

// if data to be inserted is lesser, insert in left subtree.
    else if (data <= root->data)
        {
            root->left = Insert (root->left, data);
        }
    else {
            root->right = Insert (root->right, data);
        }
        return root
}.

bool Search (BstNode *root, int data){
        if(root == NULL) {
                return false;
        }
        else if (root->data == data) {
                return Search (root->left, data);
        }
        else {
                return Search (root->right, data);
            }
    }
```

Date: / / Page no.

⇒ Find Max & Min in BST

i) iterative

```
int FindMin(BstNode* root){
    if (root = NULL) {
        cout<< "Error: Tree is empty \n";
        return -1;
    }
    BstNode* current = root;

    while (current->left != NULL) {
        current = current->left;
    }
    return current->data;
}
```

ii) Recursive

```
FindMax (BstNode * root) {
    if (root == NULL) {
        cout<< "Error : Tree is empty \n";
        return -1;
    }
    else if (root->right == NULL) {
        return root->data;
    }
    // Serach in the right subtree
    return FindMin(root->left);
}
```

⇒ Find height of a binary tree.

```
FindHeight(root)
  {
    if (root is NULL)
        return -1.
    }
  leftHeight ← FindHeight(root→left)
  rightHeight ← FindHeight(root→right)

  return max(leftHeight, rightHeight) + 1
  }
```

⇒ Binary Tree Traversal — Reading/Processing.

Tree Traversal ⇒ process of visiting each node in the tree exactly once in some order.

Tree Traversal
- → Breadth-first — level-order
- → Depth-first
  - root left right — Preorder
  - left root right — Inorder
  - left right root — Postorder

Inorder traversal of BST give you sorted data in increasing order.

Date: / / Page no:

☆ ⇒ Level Order Traversal.

⇒ Using queue

⇒ Time complexity = O(n) ⇒ all cases
Space complexity = O(1) ⇒ best case
O(n) ⇒ worst / Avg.

```
void LevelOrder(Node *root) {
    if (root == NULL) return;
    queue<Node*> Q;

    Q.push(root);

    While (!Q.Empty()) {
        Node* current = Q.front();
        Q.pop();
        cout << current->data << " ";
        if (current->left != NULL)
            Q.push(current->left);

        if (current->right != NULL)
            Q.push(current->right);
    }
}
```

Date: / / Page no:

⇒ Depth Order Traversal

Time complexity : O(n)
Space complexity : O(h)
worst : O(n) ⇒ O(n-1)
Best/Average : $O(\log_2 n)$

1) Preorder
root left right

visit i) visit the root
ii) visit left subtree
iii) visit right subtree.

```
void Preorder (struct Node* root) {
        if(root == NULL) return;
        printf("%c", root->data);
        Preorder (root->left);
        Preorder (root->right);
    }
```

~~void Inorder (Node *~~

ii) Inorder

```
void Inorder (struct Node* root) {
    if(root == NULL) return;

    Inorder (root->left);
    printf("%c", root->data);
    Inorder (root->right);
}
```

iii) Postorder

```
void Postorder (struct Node* root) {
    if(root == NULL) return;

    Postorder(root->left);
    Postorder (root->right);
```

```
    printf("%c", root->data);
  }
```

Check if a given ~~Binary~~ Binary tree is BST.

```
bool IsBinarySearchTree(Node* root, int minValue, int MaxValue)
{
  if(root == NULL) return true;

  if(data->data > minValue &&
     root->data < maxValue &&
     IsBinarySearchTree(root->left, minValue, root->data)
     IsBinarySearchTree(root->right, root->data, MaxValue))
  return true;

  else
    return false;
}
```

Date / / Page no.

⇒ Delete a node from BST

→ Case 1) No child
Case 2) One child (copy child to delete position & delete child)
Case 3) 2 children.

↓ Find min in right copy the value in targeted node. Delete duplicate from right subtree

→ Find max in left copy the value in targetted value. Delete duplicate from left subtree.

Code

```
struct Node* Delete(struct Node *root, int data) {
    if(root == NULL) return root;
    else if(data < root->data)
        root->left = Delete(root->left, data);
    elsif(root
```

Code

```
struct Node* Delete(struct Node *root, int data) {

    if(root == NULL) return root;
    else if (data < root->data)
            root->left = Delete(root->left, data);

    elseif (data > root->data)
            root->right = Delete(root->right, data);

    else {
        if(root->left == NULL && root->right == NU
            delete root;
            root = NULL;
```

Case 1

right = 300
root = 300
temp = 200

Date: / / Page no:

```
// case 2 : One children
else if ( root -> left == NULL)
    { struct Node * temp = root;
      root = root -> right;
      delete temp;
    }

else if ( root -> right == NULL)
    { struct Node * temp = root;
      root = root -> left;
      delete temp;
    }

// case 3 : 2 children
else {
      struct Node * temp = FindMin ( root -> right);
      root -> data = temp -> data;
      root -> right = Delete ( root -> right, temp -> data);
    }
}

return root;
}
```

⇒ Inorder Successor in a BST

Time Complexity = O(h) h = height

Case 1) Node has right subtree ⇒

└ Go deep to leftmost node in right subtree
OR
Find min in right subtree

Case 2) No right subtree

└ Go to the nearest ancestor for which given node would be in left subtree

Code

```
struct Node* GetSuccessor(struct Node* root, int data) {
    struct Node* current = Find(root, data);

    if(current == NULL) return NULL;
    if(current->right != NULL) {
        return FindMin(current->right);
    }

    else {
        struct Node* successor = NULL;
        struct Node* ancestor = root;

        while(ancestor != current) {
            if(current->data < ancestor->data)
            {
                successor = ancestor;
                ancestor = ancestor->left }
```

Date: / / Page no:

```
else
{ancestor = ancestor->right;}
}
return successor;
}
}
```

Date: / / Page no:

Graph

Graph ⇒ A graph G is an ordered pair of set V of vertices and a set E of edges.

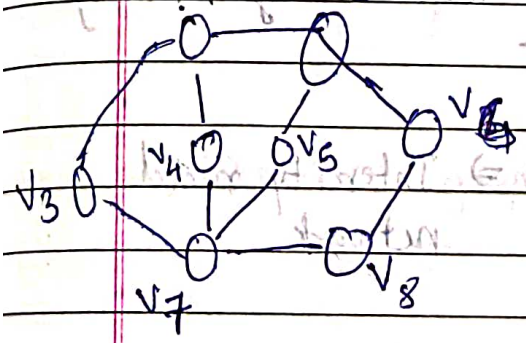
$$G = (V, E)$$

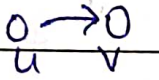
~~order~~

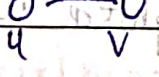
ordered pair ⇒ $(a, b) \neq (b, a)$ if $a \neq b$.

Unordered pair :- $\{a, b\} = \{b, a\}$

Note ⇒ Tree is a special kind of ~~tree~~ graph where if N ~~=~~ nodes then (N-1) edges one each for each parent-child relationship (exactly one path from root to ~~note~~ node).



edges ⇒  directed

 undirected

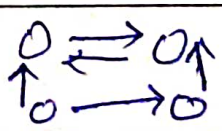
Edge e_{ij} associated with ordered pair (v_i, v_j)

$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$

$E = \{\{v_1, v_2\}, \{v_2, v_6\}, \{v_1, v_3\} \ldots\}$

→ Directed vs Undirected

↳ diagraph



Date __/__/__ Page no

→ Application

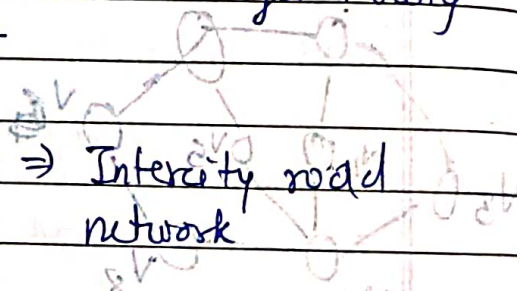
1) Social Network ⇒ find mutual friend of a friend (Length of shortest path from nodes equal to 2).

2) World wide web ⇒ directed graph
⇒ web crawling (graph traversal)

⇒ Weighted vs Unweighted

↳ If each edge or each vertex or both are associated with some +ve no. then the graph is called as weighted graph

Unweighted ⇒ a weighted graph with all edges having weights = 1 unit



Weighted undirected graph ex ⇒ Intercity road network

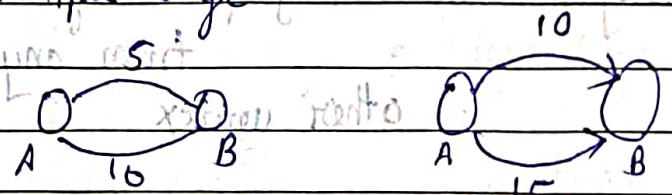
⇒ Properties of Graph

|V| = no of vertices

|E| = no. of edges.

⇒ Self loop :- If the end vertices of any edge are same the edge is called as self loop

⇒ Multiedge or Parallel edge :- If there are more than one edge is associated with given pair of vertices then those edge called as Parallel edge or multiple edge



⇒ Simple graph ⇒ no self loop or parallel edge.

└ No. of edges :-

min = 0 ⇒ $E = \phi$

⇒ directed graph :-

$$\max = (\text{no. of vertices}) \times (\text{no. of vertices} - 1)$$
$$= n(n-1)$$

$\therefore 0 \leq |E| \leq n(n-1)$, if directed

⇒ $\therefore 0 \leq |E| \leq \frac{n(n-1)}{2}$, if undirected assuming no self loop or multiedge

Dense graph = too many edges

sparse graph = too few edges

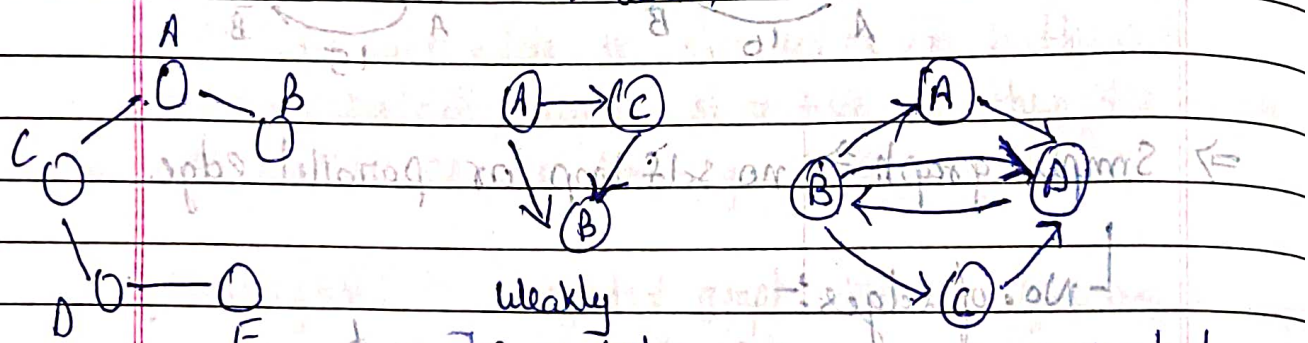
Date / / Page no:

⇒ Path or Walk :- A sequence of vertices where each adjacent pair is connected by an edge.

i) Simple path :- no vertices (or and thus no edges) are repeated.

ii) Trail :- no edges are repeated.

⇒ Strongly Connected graph :- If there is a path from any vertex to any other vertex



Connected

Weakly Connected

[No path for B to A or B to C]

Strongly connected

Cycle

⇒ Closed Path/Walk ⇒ start and ends at same vertex at length > 1

⇒ Simple Cycle ⇒ no repeatition other than start and end.

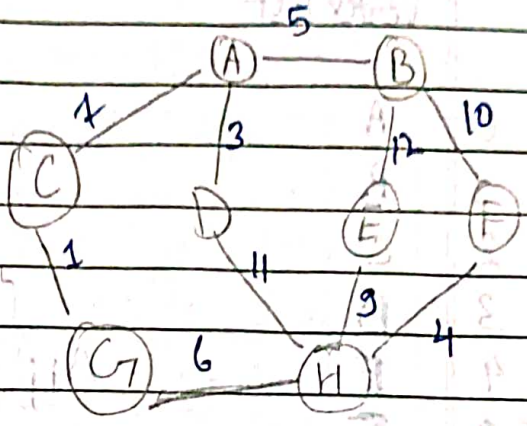
⇒ Acyclic graph ⇒ a graph with no cycle

ex → tree.

→ Directed Acyclic graph (DAG).

⇒ Graph Representation

1) Edge List



Vertex list

A
B
C
D
E
F
G
H

$O(|V|)$

Edge List

A	B	5
A	C	7
A	D	3
B	E	12
B	F	10
C	G	1
D	H	11
E	H	9
F	H	4
G	H	6

$O(|E|)$

```
Class Edge
{
    string startVertex;
    string EndVertex;
    int weight;
};
```

Space Complexity = $O|V| + O|E| = O(|V| + |E|)$

Time Complexity

Operation	Running time
finding adjacent nodes	$O(|E|)$
check if given nodes are connected	$O(|E|)$
	$O(|E|) \approx O(|V^2|)$

Not okay.

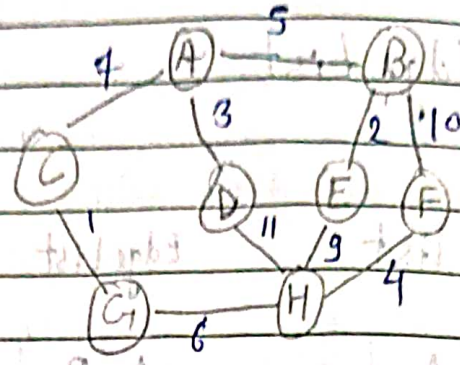
Symmetry $A_{ij} = A_{ji}$

Date / / Page no

2) Adjacency ti Matrix

Vertex list

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H



$$A_{ij} = \begin{cases} 1, & \text{if } \exists \text{ edge from } i \text{ to } j \\ 0, & \text{otherwise} \end{cases}$$

Adjacency Matrix

	0	1	2	3	4	5	6	7
0	∞	5	7	3	∞	∞	∞	∞
1	5	∞	∞	∞	2	10	∞	∞
2	7	∞	∞	∞	∞	∞	1	∞
3	3	∞	∞	∞	∞	∞	∞	11
4	∞	2	∞	∞	∞	∞	∞	9
5	∞	10	∞	∞	∞	∞	∞	4
6	∞	∞	1	∞	∞	∞	∞	6
7	∞	∞	∞	6	11	9	4	∞

Matrix is Symmetry

Time Complexity

finding adjacent nodes $O(V)$

finding if two nodes are connected $O(V)$

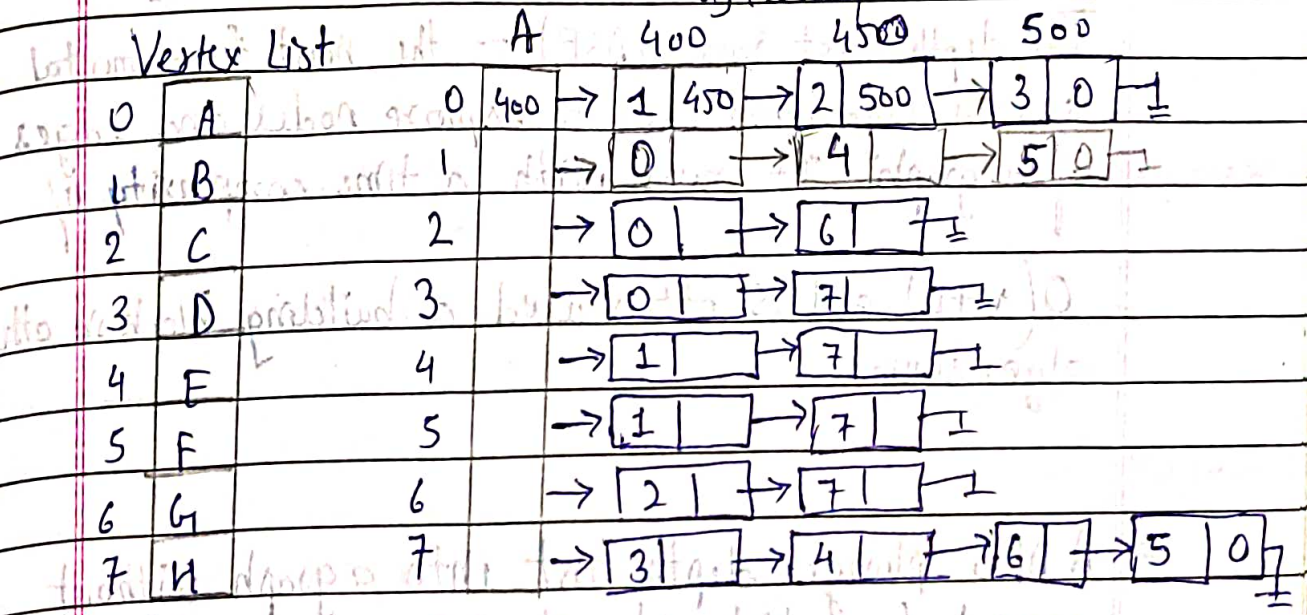
Space Complexity $= O(V^2)$

Good if graph is dense or V^2 is too less to matter.

O(E) Space complexity

Date: / / Page no:

3) Adjacency List




```
Struct Node{
    int data;
    Struct Node* next;
    }
struct Node*A[s];
```

Space Complexity = O(|E| + |V|)

Most graphs are sparse with |E| << |V|*|V|

⇒ Time Complexity

find adjacent nodes	O|V|
find if two nodes are connected	O|V|

⇒ Depth first Traversal

The depth first search (DSF) is the most fundamental search algorithm used to explore nodes and edges of a graph. It runs with a time complexity of $O(V+E)$ and is often used a building block in other algorithum.

⇒ A DFS plunges depth first into a graph without regard for which edge it takes next until it cannot go any further at which point it backtracks and continues

```
# Code in Python
#Global or class scope variable
n = no. of nodes in graph
g = adjacency list representing graph.
visted = [false, ..., false] #size n.

function dfs(at):
    if visited[at]: return
    visited[at]: true

    neighbours = graph[at]
    for next in neighbours:
        dfs(next)

# Start
start_node = 0
dsf(startnode).
```

s the most fundamental
plore nodes and edges
a time complexity of

a building block in other

o a graph without
s next until it
t which point it

abvgement DFS algo. to:

Application :- 1) Compute a graph's minimum spanning tree
2) Detect & find cycles in a graph
3) Check if a graph is bipartite
4) Find strongly connected components

many more...

⇒ C++ code dsf (node * temp, int month)

```
for (i = 1 to total user)
  { visited [i] = 0 }

visited [temp -> index] = 1;
temp = temp -> next;
while (temp != NULL)
  {
    if (visited [temp -> index] != 0)
        dfs ( temp , month)
        temp = temp -> next;
    }
}
```

→ condition for print

```
if (month == temp -> mm)
  { cout << . . . .
  };
```

⇒ ~~BSF (Bredth First Traversal)~~
⇒ ~~BFS (Breadth First Search)~~

abugement DFS algo. to:

Application :- 1) Compute a graph's minimum spanning tree.
2) Detect & find cycles in a graph.
3) Check if a graph is bipartite.
4) Find strongly connected components.

⇒ C++ code dsf (node * temp, int month)

```
for (j=1 to total user)
  { visited [j] = 0 }
visited [temp -> index] = 1;
temp = temp -> next;
while (temp != NULL)
  {
    if (visited [temp -> index] == 0)
        dfs (temp, month)
        temp = temp -> next;
    }
}
```

→ condition for print

```
if (month == temp -> mm)
  { cout << ---
  };
```

⇒ ~~BSF (Bredth First Traversal)~~
⇒ ~~BFS (Breadth First Search)~~

Date: / / Page no:

⇒ Breadth first Search (BFS).

Time complexity = O(V+E)

⇒ Useful in finding the shortest path on unweighted graph

⇒ A BFS starts at some arbitary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours.

⇒ The BFS algorithm uses a queue data structure to track which node to visit next. Upon reaching a new node the algorithm add ~~its~~ it to the queue to visit it later.

```
BFS (){
{ queue q;
  int 0,
int index=0= index1 ;
 int max_comment =0= min_comment;
  for (nt i=0 to total_user)
       {
          visited [i] = 0
       }

node *temp, *p = qr[0]
max_comment = min_comment = p->tot_comm;
visited [p->index] = 1;
 index = p->index;
 index1 = p->index;
  q.enqueue(p);
```

```
while(!q.isEmpty())
{
    temp = q.dequeue();
    p = temp->next;

    while(p != NULL)
    {
        if(visited[p->index] == 0)
        {
            max_comment = p->tot_comm;
            index = p->index;
        }
        if(p->tot_comm < min_comment)
        {
            min_comment = p->tot_comm;
            index = p->index;
        }

        visited[p->index] = 1;
        q.enqueue(p);
    }
    p = p->next;
}
}

cout << gr[index]
cout << gr[index] -> tot_comm.
}
```

Date: / / Page no:

⇒ Dijkstra's shortest Algorithm.

⇒ Single Source Path (SSI shortest Path algorithum

⇒ Time complexity = $O(E * \log(V))$ // depends on implementation & data structure used

⇒ Constsaint
1) Only contain non-negative weights

⇒ Greedy Algorithm.

⇒ Types ⇒ 1) Lazy Dijkstra's Algorithm
2) Eager Dijkstra's Algorithm.
3) Heap-optimization with D-ary heap.

⇒ Algorithm
1) Create a set spt sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this is set is empty.

$O(V^2)$ //

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE.
Assign distance value as 0 for the source vertex so that it is picked first.

Date: / / Page no:

⇒ Dijkstra's shortest Algorithm.

⇒ Single Source Path (SSI shortest Path algorithm

⇒ Time complexity = $O(E * \log(V))$ // depends on implementation & data structure used

⇒ Constraint
1) Only contain non-negative weights

⇒ Greedy Algorithm.

⇒ Types ⇒ 1) Lazy Dijkstra's Algorithm
2) Eager Dijkstra's Algorithm.
3) Heap-optimization with D-ary heap.

⇒ Algorithm
1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

$O(V^2)$

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE.
Assign distance value as 0 for the source vertex so that it is picked first.

3) While sptSet doesn't includes all vertices
⇒ a) Pick a vertex u which is not there in sptSet and has minimum distance value.
⇒ b) Include u to sptSet.
⇒ c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

{ if (d(u) + c(u,v) < d(v)) { d(v) = d(u) + c(u,v) } }

Date: / / Page no:

⇒ Spanning Tree

A spanning tree is a subset of graph G, which has all the vertices covered with minimum possible number of edges (V-1) {vertices-1}.

⇒ ~~Minimum~~ Hence, a spanning tree does not have cycle and it cannot be disconnected.

⇒ Minimum Spanning Tree ⇒ total weight is smallest among all the spanning tree of graph.

⇒ Prim's Algorithm

1) greedy algorithm

types ⇒ i) lazy version $O(E * \log(E))$ } using advanced data structure
ii) Eager version $O(E \times \log(V))$

Time complexity = $O(V^2)$ in normal case by using array & adjacency matix

⇒ Algorithm

1) Create a set mstSet that keeps track of vertices already included in MST.

2) Assign a key value to all vertices in the input graph. Initialize all key values as Infinite. Assign key value as 0 for the first vertices so that it is picked first.

Date: / / Page no:

⇒ Spanning Tree

A spanning tree is a subset of graph G, which has all the vertices covered with minimum possible number of edges (V-1) {vertices-1}

⇒ ~~Minimum~~ Hence, a spanning tree does not have cycle and it cannot be disconnected.

⇒ Minimum Spanning Tree ⇒ total weight is smallest among all the spanning tree of graph.

⇒ Prim's Algorithm

1) gruedy algorithm

types ⇒ i) lazy version $O(E * \log(E))$ } using advanced data structure
ii) Eager version $O(E \times \log(V))$

Time complexity = $O(V^2)$ in normal case by using array & adjacency matix

⇒ Algorithm

1) Create a set mstSet that keeps track of vertices already included in MST.

2) Assign a key value to all vertices in the input graph. Initialize all key values as Infinite. Assign key value as 0 for the first vertices so that it is picked first.

3) While mstSet doesn't includes all vertices
...a) Pick a vertex u which is not there in mstSet and has minimum key value.
...b) Include u to mstSet
...c) Update key value of all adjacent vertices of u. To To update the key value, iterate through all adjacent vertices. For every adjacent vertex V, if weight of edge u to V is less than the previous key of V, update the key value as weight of u to V.

⇒ Kruskals algorithm.
1) greedy algorithm
2) Use Set & Union-find Algorithm (path compression)

⇒ Time Complexity ⇒ O(E Log E) or O(E log V)

⇒ Algorithm
1) Sort all the edges in non-decreasing order of their weight.

2) Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else discard it.

3) Repeat step 2) until there are (V-1) edges in the spanning tree.

Step 2) use the Union-find algorithm to detect cycles.

Date: / / Page no:

⇒ Topological Sort

Real time Problem → i) Program build dependencies
ii) College class prerequisites
iii) Event scheduling
iv) Assembly instructions

Topological Ordering is an ordering of the nodes in a directed graph where for each directed from node A to node B, node A appears before node B in the ordering.

Note ⇒ Topological ordering are not unique.

Algorithm ⇒ i) Modified Depth first Traversal using stack
ii) Kahn's algorithm

Time complexity = O(V+E)
Space complexity = O(V)

⇒ Only Directed Acyclic Graph (DAG's) have a topological orderings.

→ Kahn's Algo ⇐ i) Compute in-degree (number of incoming edges) for each of the vertex present in the DAG and initialize the count of visited node as 0.

ii) Pick all the vertices with in-degree as 0 and add them into a queue. (Enqueue operation).

3) Remove a vertex from the queue. (Dequeue Operation) then:-

1) Increment count of visited nodes by 1.
2) Decrease in-degree by 1 for all its neighboring nodes.
3) If in-degree of a neighboring nodes is reduced to zero, then add it to the queue.

4) Repeat step 3 until the queue is empty.

5) If count of visited nodes is not equal to the number of nodes in the graph then the topological sort is not possible for the given graph.

Hash Tables

⇒ A Hash Table (HT) is a data structure that provides a mapping from key to values using a technique called hashing.

⇒ In key-value pairs keys must be unique, but values can be repeated & keys need to be hashable

⇒ HT often used to track item frequencies.

⇒ A hash function H(x) is a function that maps a key 'x' to a whole number in a fixed range

ex ⇒ $H(x) = (x^2 - 6x + 9) \bmod 10$ maps all integers keys to the range [0,9]

⇒ If H(x) = H(y), then object x & y might be equal, but if $H(x) \neq H(y)$ then x & y are certainly not equal.

⇒ For files we used cryptographic hash functions also called checksums.

⇒ Hash function H(x) must be deterministic

means H(x) = y, then H(x) must always produce y. & never another value

good

⇒ Properties of Hash function

1) Make hash function uniform to avoid collisions

2) Keys used in our functions hash table are immutable. data types. to enfore hashable property.

3) Time complexity = O(1) * for insertion, lookup & removal of data from hash table.

* the constant time behaviour attributed to hash table is only true if you have a good uniform hash function

⇒ Hash Collision :- The situation where newly inserted keys maps to an already occupied slot in hash table is called collision

Hash Collision resolution Techiques

1) Separate Chaining
2) Open addressing
3) Bucket hashing

1) ~~Separate~~ Separate chaining deals with hash collisions by maintaining a data structure (usually a linked list) to hold all this different values which hashed to a particular value.

2) Open addressing deals with hash collisions by finding another place within the hash table for the object

Hash Collision ⇒ Hash functions are not perfect, therefore sometimes two keys K_1, K_2 ($K_1 \neq K_2$) has to the same value.

$H(K_1) = H(K_2)$

to go ~~the~~ offsetting it from the position to which it hashed to.

Complexity :-

Operation	Average	Worst
Insertion	O(1)*	O(n)
Removal	O(1)*	O(n)
Search	O(1)*	O(n)

⇒ Note → Separate Addressing is not limited to a ~~linked~~ Linked List Only, we can also use other data structure like self-balancing trees and etc

Separate Chaining Ex →

0	
1	Mayank, 26 → Sahu, 18
2	AAB, 18
3	Will, 21 → Finn, 22
4	
5	Leah, 18 → Lasa, 34 → Mark, 40

⇒ Once HT contains a lot of elements you should create a New HT with a larger capacity & rehash all the items inside the old HT and disperse them throughout the new HT at different locations.

⇒ Open addressing

└ Key - value pairs are stored in the table (array) itself.

Load factor = $\frac{\text{item in table}}{\text{size of table}}$ (generally 0.7, 0.6)

Probing Sequences

offset = i
$\pm i^2$
$i \times H_2(k)$

i) Linear probing ⇒ $P(x) = ax + b$
a & b are constant

ii) Quadratic probing ⇒ $P(x) = ax^2 + bx + c$
a, b & c are constant

iii) Double hashing
$P(k, x) = x * H_2(k)$, where $H_2(k)$ is a secondary hash function

iv) Pseudo random number generator
$P(k, x) = x * RNG(H(k), x)$, where RNG is random generator function seeded with $H(k)$

⇒ General Idea
x = 1, ~~to hash~~
KeyHash = H(k), index = KeyHash

```
while table[index] != null:
    { index = (KeyHash + P(K,x)) mod N
      x = x+1 }
insert (k,v) at table[index].
```

Date: / / Page no:

⇒ Randomly selected probing sequences modulo N will produce a cycle shorter than the table size.
└ This leads to problem of infinite loop.

⇒ ~~We~~ ~~th~~ to avoid this we choose probing functions to those which produce a cycle of exactly length N* (few exception are there)

1) Linear Probing (LP)

LP is a probing method which probes according to a linear formula, specifically:-

$P(x) = ax + b$, where $(a \neq 0)$, b are constants.
b is obsolete

⇒ Which value(s) of the constant a in $P(x) = ax$ produce a full ~~value~~ cycle modulo N? (soo that we avoid infinite loop)
Happend when

Ans → A~~#~~ a & ~~b~~ N are relatively prime (i.e. their gratest common factor (GCD) is equal to one).

⇒ A common choice for P(x) is $P(x) = 1x$, since GCD $(N, 1) = 1$, no matter the the choice of N (table size)

⇒ Once elements is greater than threshold value create new hash table generally double the size of original one whatever you do make sure GCD $(N, a) = 1$ still holds

α = threshold value = load factor × size of table (N)

2) Quadratic Probing (QP)

QP is a probing method which probes according to a quadratic formula, specifically:-

$P(x) = ax^2 + bx + c$, where a, b, c are constant & $a \neq 0$; (constant c is obsolete)

⇒ Decide Probing function

i) Let $P(x) = x^2$, keep the table size a prime number > 3 and also keep $\alpha \leq 1/2$

ii) $P(x) = (x^2 + x)/2$, keep the table size of power of two.

iii) $P(x) = (-1^x) * x^2$, keep the table size a prime N where $N \equiv 3 \bmod 4$. [ex N = 23]

3) Double Hashing (DH).

DH is a probing method which probes according to a constant multiple of another hash function, specifically.

$P(k, x) = x * H_2(k)$, where $H_2(k)$ is a second hash function.

H_2 must have same type of keys as $H_1(k)$

⇒ To fix the issue of cycles pick the table size to to be a prime number and also compute the value of δ

$\delta = H_2(k) \bmod N$.

Date: / / Page no:

If $\delta = 0$, then set $\delta = 1$.

Notice :- $1 \leq \delta < N$, i.e. $GCD(\delta, N) = 1$. since N is prime.

⇒ To resize one strategy is compute $2N$ and find the next prime above this value.

⇒ Removing a element from Open Addressing

- The solution is to place a unique Number called a tombstone instead of null to indicate that a (k,v) pair has been deleted and that the bucket should be skipped during a search.

Tombstones count as filled slots in the HT, so they increase the load factor and will be removed when the table is resized.
Also, when inserting a new (k,v) pair you can replace buckets with tombstones with the new key-value pair.

Heap Data Structure

⇒ Binary Heap

|- 1) Max Heap :- In a Max-heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub trees in that binary tree.

OR ⇒ for every node i, the value of node is less than or equal to its parent value.

$$A[Parents(i)] \geq A[i] \quad \text{for every } i$$

2) Min Heap :- In a Min-Heap the key present at the root must be minimum among the keys present at all of its children. The same property must be recursively true for all sub tree in the binary tree.

OR

for every node i, the value of node is greater than or equal to its parent value

~~A[Parents[i]~~

$$A[Parents(i)] \leq A[i].$$

Note :- Binary heap is used complete binary tree (All levels except possibly the last are completely filled and all nodes are as left as possible

⇒ Array based Representation (index starts at 0)

Parents node ⇒ (i-1)/2
left child ⇒ (2*i)+1.
Right child ⇒ (2*i)+2.

⇒ Complexity = O(nlogn).

⇒ Applications:-
1) Heap Sort
2) Priority Queue
3) Graph Algorithms

⇒ Heap Sort Code:- (Max Heap gives in increasing order).

// Heapify Method ⇒ Complexity ⇒ O(logN)

```
void heapify (int arr[], int n, int i)
{
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    if (l < n && arr[l] > arr[largest])
        largest = l;
    if (r < n && arr[r] > arr[largest])
        largest = r;
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
```

Left Leaf nodes starts from floor (n/2) [Index starts = 0].

```
void build-maxheap (int Arr[])          //complexity → O(n)
{
    for (i = N/2-1; i >= 0; i--)
    {
        maxheapify (Arr, i);
    }
}
```

~~i = N/2 to 0~~
i = N/2 - 1 to 0

```
void heap-sort (int Arr[])
{
    int heap_size = N;
    build_maxheap (Arr);
    for (int i = N-1; i > 0; i--)
    {
        swap (arr[0], arr[i]);
        heapify (arr, i, 0);
    }
}
```

O(n log n)