

Data Structure

Date _____ Page No. _____

⇒ 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 ~~an ADT~~ (structure) \leftrightarrow List (Implementation)

List

↳ Store a given number of elements of a given data type.

O(1) ↳ Write / modify element at a position.

(n) ↳ Read element at a position.

(Array)

List

O(1) ↳ Empty list has size 0.

O(n) ↳ Insert extra element at any pos.

O(n) ↳ Remove any element from list.

O(1) ↳ Count

O(n) ↳ Read / modify element at a position.

Specify data type of elements.

1) $b \leftarrow [n]$

(2) O(b) times list to

list with b elements

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

Free the memory for the previous array.

⇒ Linked List

create at few no. of structure which is node
 with two field next & data. so, structure is as follows:

```

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

head = first node

⇒ We can access any element of the list in constant time.
 while array requires constant time

⇒ Address of the node give us complete access of
 either complete list or part of list.

⇒ Access of n elements $\Rightarrow T \propto n \Rightarrow O(n)$

Insertion $\Rightarrow O(n)$, Delete $\Rightarrow O(n)$

→ Array Vs Linked List (Answe: Which is better?)

→ Array vs. Linked List

1) Cost of accessing an element

constant time $\Rightarrow O(1)$

Average case $\Rightarrow O(n)$

2) Memory requirement

fixed size \Rightarrow memory will 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 $\Rightarrow O(n)$

at the end $\Rightarrow O(1)$

$O(1) \Rightarrow$ beginning

$O(n) \Rightarrow$ end

$O(n) \Rightarrow$ middle

4) Easy of use

to insert & delete without reprogramming

⇒ Creating an linked list

struct Node {

 int data; // data part

 struct Node *Link; // pointer

}

Node *A

A = NULL

→ New Node(),
(constructor)

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 → Link = NULL

(constructor)

(constructor)

(constructor)

Node *temp1 = A

while (temp1 → Link != NULL) { }

{

 temp1 = temp1 → Link;

}

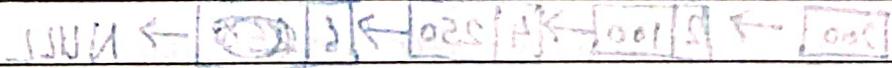
temp1 → Link = temp1 → Link → next;

→ next = NULL

⇒ Insertion of Node at beginning

#include <stdlib.h> // for to alloc a mem

#include <stdio.h>



struct Node {

 int data; // data part

 struct Node *next; // next part

};

struct Node *head; // head part

temp → snext = curr → snext ← snext ← snext → head

snext = curr + snext

```
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)
```

```
{
```

A = Insert method

```
Node* temp = (Node*) malloc(sizeof(struct Node));
```

```
temp → data = x
```

```
temp → next = NULL; //not = 1 point
```

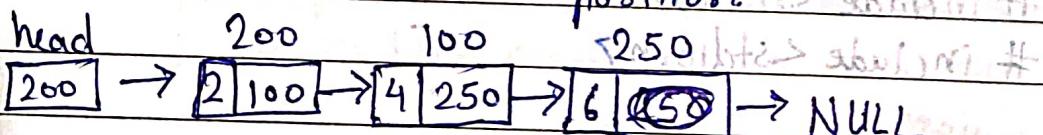
```
if (head != NULL)
```

```
{ temp → next = head; //not = 1 point
```

```
head = temp;
```

```
head = temp; //assigned to head for insertion
```

⇒ Insert a node at nth position



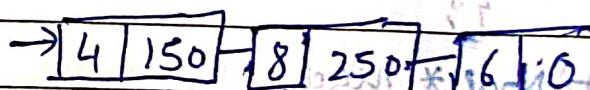
Create Node

[8] NULL

150

↓ next pointer
temp1

Adjust links



Main logic ⇒ temp1 → next = temp2 → next
temp2 → next = temp1

node insertPos (inserting data to chosen position) :-

Function :- node* insertPos(node* head, int loc)

```

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; // first element
    else
        b = x;
    for (i = 1; i < loc - 1; i++)
        b = b->next;
    a->next = b->next;
    b->next = a;
    return x;
}

```

⇒ Print a linked list (i.e.) root value

Void Print()

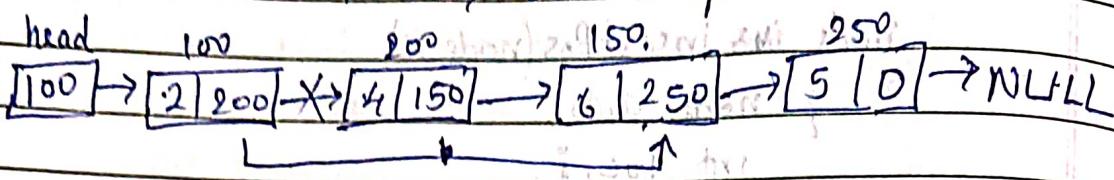
```

{
    p = d;
    struct Node* temp = head;
    while (temp != NULL)
    {
        cout << temp->data;
        temp = temp->next;
    }
    cout << endl;
}

```

⇒ Delete a node at nth position.

- 1) fix the links, 2) Free the space



Main logic: $b \rightarrow next = a \rightarrow next$

$a \rightarrow next = NULL$ (Not compulsory).

After step 1 free(a); (If head == a)

Node delete (node first): D = x

{ node a,b;

int loc,i;

(++) printf("Enter the location: ");

scanf("%d", &loc);

a = first - d = d

if (first == NULL)

{ printf("List is empty");

return first;

else

{ i=1;

while (for (i=1, i<loc-1, i++))

{ b = q

OR

for (i=1; i<loc-1; i++)

{

temp1 = temp1->next

}

struct Node*

temp2 = temp1->next

temp1->next = temp2->next

free(temp2);

b->next = a->next

a->next = NULL

free(a);

→ R Reverse a Linked List (using q) (reverse q) reversal

i) **Iterator** q → i

struct Node

{ int data;

 struct Node* next; }

} reverse

vector (int n = -1); vector (int n = -1) for base case

struct Node* head; vector (int n = -1) for base case

{ (base case)

struct Node* Reverse (struct Node* head);

{

 struct Node** current, *prev, *nex;

 current = head; hosted variable

 prev = NULL;

 while (current != NULL) { for loop

 {

 nex = current->next;

 current->next = prev;

 prev = current->next;

 current = nex; if

 }

 return prev; return previous A

 } return head; G

}

ii) Using Recursion from question 1 & question 2

struct Node* head; base

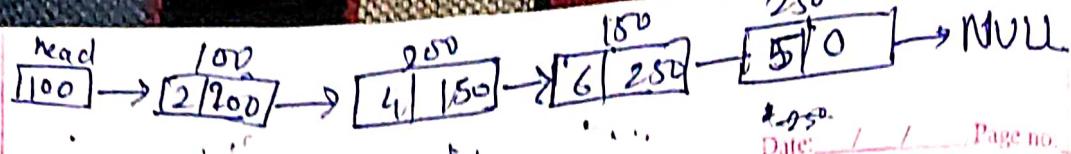
Void Reverse (struct Node* p)

{

 if (p->next == NULL) root of empty tree

 { head = p; }

 return;



Date: / / Page no. /

Reverse ($p \rightarrow \text{next}$); *(This is backward in normal flow)*

struct Node* $q = p \rightarrow \text{next}$; $\{ \Rightarrow p \rightarrow \text{next} \rightarrow \text{next} = p;$
 $q \rightarrow \text{next} = p$
 $p \rightarrow \text{next} = \text{NULL}$ *about to exit?*
return for ?

\Rightarrow Print Using Recursion with *traverse*

Forward

```
void Print(struct Node* p)
{
    if (p == NULL) return;
    printf("%d", p->data);
    Print(p->next);
}
```

Reverse

```
if (p == NULL) return;
Print(p->next);
printf("%d", p->data);
}
```

\Rightarrow Doubly Linked List *about to discuss*

struct Node *UN = forward*

{ *UN = reverse* }

int data; *reverse = UN*

struct Node* *next*;

struct Node* *prev*;

{ *UN = previous* }

Advantage \Rightarrow Reverse Look-up (only one pointer)

\Rightarrow D. Implementing Operation is easy

D. Disadvantage \Rightarrow Extra memory for pointer to previous

*node * prev; about to discuss*

(q->prev) = previous link

Implementation

```
struct Node* head;
if (q->prev == head) {  

    q = head;  

    cout << "new tail";  

}
```

```

void { (1) function b/w
struct Node* GetNewNode(int x); // New Function
{
    struct Node* newNode = (struct Node*) malloc(sizeof(
        struct Node));
    newNode->data = x;
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}
    { (2) function b/w
        insert <- grant == grant
    }
}

```

```

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) temp = temp->next;
temp->next = newNode;
newNode->prev = temp;

```

void ReversePrint () {

struct Node *temp = head;

if (temp == NULL) return;

while (temp->next != NULL) {

temp = temp->next; }

if (temp != NULL)

{

print ("%d", temp->data);

temp = temp->prev; }

}

void Print () {

struct Node *temp = head;

while (temp != NULL)

{

temp->data print(temp->data);

temp = temp->next; }

}

(External) External link

function not found function not found

if (head == prev->shallow + write)

if (shallow + write) = shallow + shal + write

(JLUN == head) +

if (shallow == head)

return

next->prev = prev (JLUN) + next->prev)

= next->shallow { shallow +1 = &next->prev

next

3) Stack

- ADT (Abstract Data Type) ~~for stack~~ ~~for queue~~
- Stack \Rightarrow 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)

Operations:-

- (1) Push(x)
- (2) Pop()
- (3) Top()
- (4) IsEmpty

Application \rightarrow Function call / Recursion

3) Balanced Parenthesis

Array Implementation

int A[10]

top $\leftarrow -1$ //empty stack

Push(x)

Top()

{

A[++top] \leftarrow x

= x

A[top] \leftarrow x

Pop()

{

top \leftarrow top - 1

?

IsEmpty()

{

if (top == -1) return true

else return false

?

imitation

\Rightarrow 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 on

overflow. \Rightarrow create a larger array. Copy all elements.

new array will have M new array. Then find a suitable place to insert them between old and new arrays.

Linked List Implementation

Insert / delete $O(n)$ time complexity

1) at end of list \rightarrow [time complexity = $O(n)$]



2) At beginning (head) \Rightarrow ($O(1)$)

void Push(int x)

{

 struct Node *temp = (struct Node*) malloc(sizeof(struct Node));

 temp->data = x; temp->link = top;

 top = temp; top = temp; //translating words

}

void Pop()

{

 if struct *Node *temp;

 if (top == NULL) return; l = got = got;

 temp = top;

 top = temp->link;

 top = temp->Link;

 free (temp);

 got = got;

\Rightarrow $O(1)$ \Rightarrow does not cause overflow because it is a word not array.

int main() { int top = 0; int l = 0; int got = 0; }

 for (i = 0; i < 10; i++) { l = l + 1; got = got + 1; }

⇒ 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]);
    main() {
        char c[51];
        gets(c);
        Reverse(c, strlen(c));
        printf("%s", c);
    }
}
```

// loop for pop.

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

;() qot-2 = front to qot

Time & Space Complexity ($\Rightarrow O(n)$)

⇒ We can also take different approach in which the start & end index of the string initially in any var store in any variable, & while $i < j$ we can swipe the characters at these position, & increment i, j

Better approach

(these give time complexity = $O(n)$ & space complexity (1))

thus giving overall time $O(n)$ & space $O(1)$

ii) Reverse a Linked List from head to tail (i.e.)

i) Iterative Solution \Rightarrow Time = $O(n)$ & Space = $O(1)$

ii) Recursive Solution \Rightarrow Time = $O(n)$ Space = $O(n)$

with (Implicit Stack), time in O(n) & space $O(1)$

(iii) Explicit Stack

possible at any time - parallel
processes or threads

void Reverse()

{

if (head == NULL) return; // handle situation if

stack <struct Node*> s; // push elements here

Node* temp = head;

do

while (temp != NULL) { // do something with

s.push(temp); i = i + 1; // i++

temp = temp->next; i++; }

temp = s.top(); // do something with

head = temp; i++; // i++

stack.pop(); i++; }

while (!s.empty()) { // do something with

temp->next = s.top(); // do something with

s.pop(); i++; }

temp = temp->next;

return head; // return modified linked list

if stack is empty, then temp->next == NULL; otherwise it is a

new value for next. i.e. i > f. When all elements are

processed, then i = n. Return control to main.

\Rightarrow Check for balanced Parentheses [() or [] or { }].

[() pair]

1) Every opening parenthesis must find a closing counterpart to its right and every closing parenthesis must find a opening counterpart to its left [()]

() -> () -> () -> () -> ()

() -> () -> () -> () -> ()

2) A parenthesis must cancel close only when all the

parathesis opened after its closed then what

→ 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

$\{ \} \neq []$ & $\{ \} \neq ()$

bool ArePair (char open, char close)

~~if (open == '}' & close == '()')~~ return true;
~~else if (open == '[' & close == ']')~~ return true;

if

code

bool ArePair (char open, char close)

if (open == '(' & close == ')') return true

else if (open == '{' & close == '}') return true

else if (open == '[' & close == ']') return true

return false

3. $(f1) \text{ or } (f2) \leftarrow$

$\{ (\text{topic of } f1) \leftarrow \text{variable's name is } f1 \}$

$(\text{topic of } f2) \leftarrow \text{variable's name is } f2 \}$

bool Are Parentheses Balanced (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() || !ArePairs(s.top(), exp[i])) {
            return false;
        }
        s.pop();
    }
}
return s.empty() ? true : false;

```

return s.empty() ? true : false;

not correct ('(' = symbol, ')' = sign) ↗
 Infix, Postfix, Prefix ↗

2) Infix ↗ [operator] > [operator] > [operator] > [operator]

Order of Operations

1) Parentheses ↗

2) Exponents ↗

3) Multiplication & division ↗

4) Addition & subtraction ↗

what -> result ↗

→ (right to left)

→ (left to right)

} Associativity

27 Prefix

(Polish notation) \rightarrow int 1324 student 2 sum

<operator><operand><operator>

~~(++i : ()#tand, colspc) > (; 0 = i + 1) #t~~

$$\Rightarrow ab \times c = +a(bc) a+a \times bc$$

3) Postfix

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

$\text{f}(\text{gof}, \delta) + \text{f}(\text{gof}, \delta) = \text{f}(\text{hanging}, \delta)$

<operand> <operator> <operand>

$$\text{Litterex} \rightarrow a+b \times c = a \cdot b \cdot c +$$

Strong & Weak

Evaluation of Prefix & Postfix Expressions

$\Rightarrow A * B + C * D - E$. tip: Evaluate Postfix(exp)

$$\{ (A \neq B) + (c \times d) \} - e$$

$S = \text{browsing } T^N$ Create a stack S .

$\{ (AB*)^i + (cd*)^j \mid i < 0 \text{ to } \text{length}(exp) - 1$

$$((i_1, \dots, i_n), \{AB\}cd\{+,\}) - e_{initial}$$

1000000000

Push($\exp(i)$)

Prefix is same as Postfix
but we don't have to start

get from right while it acts like

Scanning string of

first element pop = (first operand) pg. 3

十一

else if $\exp[i]$ is operator

3

$Op2 \leftarrow pop()$

$op1 \leftarrow pop()$

result \leftarrow Perform(exp[1],

, Push (res)

Op_1, Op_2

return top of stack. \leftarrow L.

\Rightarrow Code =

int EvaluatePostfix(string expression)

{

 stack<int> S; // stack <int> S; stack >

 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); // result to stack

 }

 else if (IsNumericDigit(expression[i]))

 {

 1.

 2. start a string int operand = 0;

 3. if next char is not operand or 0 -> i++ while (i < expression.length() &&

 4. IsNumericDigit(expression[i])

 5. operand = (operand * 10) + (expression[i]

 - '0')

 6. if operand & 1000 -> i++ else

 7. if

 x1000 is what it is till

 test of mod was in bud

 8. i--; // extra increment happened in loop

 9. S.push(-) // operand to adjust max

 10. S.push(operand) = 999 + terminal test

 11. (999, 100)

 12. (898, 100)

 13. f. if two digits S.top() > current

```

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;
    }
}

```

* Infix to Postfix

```

→ string InfixToPostfix(string expression)
{
    stack<char> S;
    stack<char> postfix;
    string postfix = " ";
    for (i=0; i<expression.length(); i++)
    {
        if (expression[i] == ')' || expression[i] == ',')
            continue; // not operator
        else if (IsOperator(expression[i]))
            while (!S.empty() && S.top() != '(' &&
                   HasHigherPrecedence(S.top(), expression[i]))
                postfix += 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(); // pop opening parenthesis
}

while (!S.empty())
{
    if (isdigit(S.top()))
    {
        postfix += S.top();
        S.pop();
    }
    else
    {
        if (operatorWeight[S.top()] > operatorWeight[S.top() + 1])
        {
            return postfix;
        }
    }
}

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;
}

```

int getOperatorWeight (char op) {

{ int weight = (1 & op == '+') || (1 & op == '-') ? 1 : 2; }

switch (op)

{ (op == '+' || op == '-') ? 1 :

case 'b': break; weight = 1; break;

case '-': weight = 1; break;

case '*': weight = 2; break;

case '/': weight = 2; break;

case '\$': weight = 3; break;

} (1 & op == '.') ? 0 : weight

return weight;

{ (op == '+' || op == '-') ? 1 :

case 'b'

int IsRightAssociative (char op) {

{ xiting nature

if (op == '\$') return true;

return false;

{ op1, op2, op3 } whether op3 is left or right associative

{ (op1 + op2) + op3 = op1 + (op2 + op3) }

{ (op1 * op2) * op3 = op1 * (op2 * op3) }

(op1 * op2) * op3 = op1 * (op2 * op3)

{ op1 * op2 = op2 * op1 }

Introduction to Queues

\Rightarrow Queue ADT \Rightarrow First In First Out (FIFO)

\hookrightarrow 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).

\Rightarrow Operations

- | | |
|---|---|
| 1) Enqueue (x) or Push (x)
2) Dequeue () or Pop ()
3) front () or Peek ()
4) IsEmpty ()
5) Is full | } Constant time or $O(1)$
using head
(() != tail) |
|---|---|

\Rightarrow Applications

- 1) Printer queue
- 2) Process scheduling
- 3) simulating Wait

\Rightarrow Array Implementation

```
#define Max_Size 101
```

Class Queue

```
{ private A[Max_Size]; }  

private:  

    int front, rear;  

    int A[Max_Size];
```

public:

```
Queue() { front = -1;  

          rear = -1; }
```

`bool isEmpty()`

{

`return (front == -1 && rear == -1);`

}

task: `bool IsFull()` return whether queue is full A

Logic (base) { if we want to insertion, add more cells in array }

→ If true \Rightarrow `bool IsFull()` to check front and rear position

{

`return (rear+1) % MaxSize == front ? true : false;`

}

(2) To insert elements) Max size () initial {

`bool void Enqueue (int x)` if $x \geq 0 \& front <$

{

`if (IsFull())`

{

"cout << " Error: Queue is Full"; return A

}

{

`if (IsEmpty())` front position {

{

`front = rear = 0;` initial position {

}

else

`int val = x & rear;`

{

`rear = (rear+1) % MaxSize;`

{

`A[rear] = x;` A starting {

: having rear {

: [size - x] A A + rear

: adding {

: J = front } (initial)

{ J = rear }

void Dequeue()

```

    {
        if (IsEmpty()):
            cout << "Error"; // Output error
            return;
        else if (front == rear):
            cout << "Error"; // Output error
            return;
        else:
            front = (front + 1) % Max_Size;
    }

```

int Front()

```

    {
        if (front == -1):
            cout << "Error"; return -1;
        else:
            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
}

```

$\text{rear} \rightarrow \text{next} = \text{temp};$

$\text{rear} = \text{temp};$

}

() removal b/w ?

(()) addition b/w +; ?

?

Void Dequeue() {

struct Node* temp = front;

if (front == NULL) {

I- = front = print (list is empty); if else

return;

else { if (front & rear) = front } if

if (front == NULL & rear == NULL)

front = rear = temp NULL

if (front == rear)

{ front = rear = NULL }

else { front = front \rightarrow next; }

free (temp);

} () removal b/w

{ which front

; static tm

; front & which front

?

which front

NULL = front & which front

which front = front & front

NULL = rear & which front

Unit 1 Introduction to Trees

⇒ How to decide which data structure to use? (Ans) (Q)

i) What needs to be stored?

ii) Cost of operations.

iii) Memory usage (Ans: more data)

iv) Ease of implementation (Ans: easier to implement)

⇒ Tree (Ans: A tree is a non-linear data structure)

children root = no parents (Ans: tree is a tree)

Parents (Ans: children nodes can have recursive data structure)

Sibling = have same parents (Ans: - N nodes has

Leaf = have no child. n-1 edges.

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

Ans: Number of edges in path from root to x .

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

Ans: leaf (Ans: leaf is a node with no children)

⇒ Binary tree = have at most 2 children.

Ans: 2 children

Application = 1) Storing naturally hierarchical data

Ans: file system

2) Organise data for quick search, insertion, deletion.

Ans: search, insert, delete

3) Trie (dictionary) = Ans: search, insert, delete

Ans: search, insert, delete

4) Network routing algorithms

Ans: search, insert, delete

Binary trees

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

↳ Left child & right child (i)

ii) Strict/Proper Binary tree. \Rightarrow no node has zero children.

Each node can have either 2 or 0 children.

iii) Complete Binary tree

\Rightarrow 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, ...)

Left child

Right child, root = 2ⁱ⁺¹ - 1 [root i=0]

iv) Perfect Binary tree \Rightarrow if all leaf nodes = root then

All levels are completely filled & all leaf nodes are at the same level.

⇒ if all leaf nodes are at height h = root = 2^{h+1} - 1

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

$$\text{Ans} \leq 2^0 + 2^1 + 2^2 + \dots + 2^h \text{ Ans} = 2^{h+1} - 1$$

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

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

$$\text{Ans} = \log_2(n+1) - 1 \text{ Ans} = \lceil \log_2(n+1) \rceil$$

$$\therefore \text{height} = \log_2(n+1) - 1 \quad (\text{Perfect Binary tree})$$

Additional properties about it (A)

$$\text{Height of complete binary tree} = \lceil \log_2 n \rceil$$

Height of empty tree = -1,

Date _____ Page no. _____

$\Rightarrow \text{Min-height} = \lceil \log_2 n \rceil$

$\Rightarrow \text{Max-height} = n-1$

$\Rightarrow \text{Height of Perfect Bmry tree} = \lceil \log(n+1) - 1 \rceil$

\Rightarrow Balanced Bmry tree. \Rightarrow The difference between height of left and right subtree for every node is not more than k (mostly 1).

\Rightarrow Implement Bmry tree.

a) dynamically created nodes

b) array (used for heap)

leftchild index = $2i + 1$ rightchild index = $2i + 2$

For Bmry tree for node at index i

leftchild index = $2i + 1$

rightchild index = $2i + 2$

Parents = $\lceil \frac{i}{2} \rceil$

Parents = $\lceil \frac{i}{2} \rceil - 1$

Parents = $\lceil \frac{(i-1)}{2} \rceil$

(\in go)

like if max value is 8 then total tree size = 8

total children = 7

total parents = 3

total leaf nodes = 4

(total tree), shall we take $\lceil \frac{n}{2} \rceil - 1$

$\lceil \frac{n}{2} \rceil - 1 = \lceil \frac{n}{2} \rceil - 1 = \lceil \frac{n}{2} \rceil - 1$

total = total \leftarrow shall we

$\lceil \frac{n}{2} \rceil - 1 = \lceil \frac{n}{2} \rceil - 1 = \lceil \frac{n}{2} \rceil - 1$

shall we again

Binary search \Rightarrow often

$O(\log n)$.

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 complexity \Rightarrow Average = $O(\log n)$
Worst = $O(n)$

\Rightarrow Implementation: $\text{main} / \text{C} + \text{functions}$

Struct Node {

Nodes will be created

int data; store root in heap using malloc

Node *left; is = return function in 'C' or

Node *right; = return hline with C++

}

Node *root;

root = NULL; $\backslash (i-1) \cdot N = \text{answer}$

Code \Rightarrow

struct BstNode {

int data;

BstNode *left;

BstNode *right;

}

BstNode * CreateNewNode(int data);

{ BstNode * newNode = new BstNode();

newNode \rightarrow data = data;

newNode \rightarrow left = newNode \rightarrow right = NULL;

return newNode;

BstNode* Insert (BstNode* root, int data);

{

if (root == NULL) {

root = (newNode(data));

{

// if data to be inserted is lesser, insert in left subtree.

else if (data <= root->data)

{

root->left = Insert (root->left, data);

{

{ (LUNI = ! (left < tree)) alidul

. else { (LUNI < tree) = tree;

root->right = Insert (root->right, data);

{

return root

{

bool Search (BstNode *root, int data);

if (root == NULL) {

return (false); = tree); fi

; "n/ ptnm"; P.ort; > tree;

else if (root->data == data) {

return Search (root->left, data);

{ (LUNI == tree) = tree) fi

else { (LUNI < tree) = tree;

return Search (root->right, data);

{

{ (LUNI & tree) = tree); //

(LUNI < tree) without tree;

{

\Rightarrow Find Max & Min in BST

i) iterative

```

int FindMin(BstNode* root) {
    BstNode* curr = root;
    while (curr != NULL) {
        if (curr->left == NULL)
            return curr->data;
        curr = curr->left;
    }
}

```

ii) Recursive

```

FindMax(BstNode* root) {
    if (root == NULL)
        cout << "Error : Tree is empty \n";
    else if (root->right == NULL)
        return root->data;
    else if (root->right != NULL)
        return FindMax(root->right);
}

```

→ Find height of a binary tree. [Recursion, height = 0]

FindHeight(root)

{

if (root is NULL) (0) → leftmost node

return -1; (1) → rightmost node

else return (leftHeight + 1)

leftHeight ← Find Height (root → left)

rightHeight ← Find Height (root → right)

return max (leftHeight, rightHeight) + 1

}

; (base) case, 2

→ Binary Tree Traversal (Inorder, Preorder, Postorder)

(Inorder, LNR) = traversal * reading / processing.

Tree Traversal → process by visiting each node in the tree

(Inorder = LNR) >> establishing order in some order

(Inorder = LNR) >> traversal

Tree Traversal (Inorder, Preorder, Postorder)

→ Breadth-first traversal

→ Node traversal 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.

$\star \Rightarrow$ Level Order Traversal, ~~random or by depth first~~

\Rightarrow Using queue

\Rightarrow Time Complexity = $O(n)$, \Rightarrow all cases

Space Complexity = $O(1)$ \Rightarrow best case

$O(n) \Rightarrow$ worst / Avg.

& void ~~LevelOrder~~ (Node *root) {
if (root == NULL) return;

 if (queue.empty()) return;
 queue.push(root);

 Q.push(root);

 while (!Q.empty()) {
 Node *current = Q.front();

 cout << current->data << " ";
 Q.pop();

 if (current->left != NULL)
 Q.push(current->left);

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

}

} \Rightarrow left-right

subroot \rightarrow type, flag, root

subroot \rightarrow type, flag, root

subroot \rightarrow type, flag, root

also notes may use T.R.B traversal scheme

\Rightarrow Depth Order Traversal

\rightarrow Time Complexity: $O(n)$ \rightarrow Space Complexity: $O(h)$ \rightarrow Worst: $O(n) \Rightarrow O(n^2)$ \rightarrow Best/Average: $O(\log n)$
--

i) Preorder

root left right

visit i) visit the root

ii) visit left subtree (leftmost node in left branch)

iii) visit right subtree.

```
void Preorder (struct Node* root) {
```

```
    if (root != NULL) {
```

```
        & l. visit root > visit left
```

```
< visit left > visit (printf ("%c", root->data));
```

```
        Preorder (root->left);
```

```
        Preorder (root->right);
```

```
}
```

```
{ visit root }
```

```
void Inorder (Node* -
```

ii) Inorder

{ visit root }

```
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);
```

Date: 11.11.2023

```

    printf("%d\n", root->data); // prints data
    {
        if (root->left == NULL && root->right == NULL) // leaf node
            return true;
        else
            return false;
    }
}

bool IsBinarySearchTree(Node* root, int minValue, int maxValue)
{
    if (root == NULL) return true;
    if (root->data < minValue || root->data > maxValue) // violates rule
        return false;
    else
        return IsBinarySearchTree(root->left, minValue, root->data) && // left subtree
        IsBinarySearchTree(root->right, root->data, maxValue); // right subtree
}

```

→ Done

⇒ Delete a node from BST

→ Case 1) No child

Case 2) One child

Case 3) 2 children

Find min in right

copy the value in target

targetted node. Delete target

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;
    if (data < root->data) {
        if (root->left == NULL) {
            struct Node* temp = root;
            root = root->right;
            delete temp;
        } else {
            root->left = Delete(root->left, data);
        }
    } else if (data > root->data) {
        if (root->right == NULL) {
            struct Node* temp = root;
            root = root->left;
            delete temp;
        } else {
            root->right = Delete(root->right, data);
        }
    }
}
```

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);

else if (data > root->data)

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

else {

if (root->left == NULL && root->right == NULL)

Case 1

delete root;

root = NULL;

500
right = 300
root = 300
temp = 200

Date: / / Page no.:

11 case 2: One children

else if (root → left == NULL)

{ struct Node *temp = root; }
if (root == root → right) { }
delete temp;

? delete temp; }
else if (root → right == NULL)

{ struct Node *temp = root; }

if (root == root → left) { }
else if (root == root → right) { }
delete temp; }

else if (root → right == NULL)

{ struct Node *temp = root; }

if (root == root → left) { }
else if (root == root → right) { }
delete temp; }

? { (else if (root → left == NULL)) { } }
else { } }

; else { } }

{ struct Node *temp = FndMn (root → right); }

if (root → right == NULL) { }
else if (root → right == temp → data) { }
root → right = Delete (root → right, temp → data);
temp → data =

? { } }

? { } }

? { } }

? { } }

? { } }

? { } }

? { } }

? { } }

? { } }

? { } }

? { } }

? { } }

? { } }

\Rightarrow Inorder Successor in a BST

Time Complexity = $O(h)$ $n = \text{height}$

Case 1) Node has right subtree \Rightarrow

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

```
3) struct Node* Getsuccessor(struct Node* root, int data) {
    struct Node* current = FindMin(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 }
```

else

$\text{push } \langle \text{left}, \text{right} \rangle \text{ onto stack}$

{ ancestor = ancestor \rightarrow right; }

? begin node = (left) \rightarrow (right), next

return successor;

} \rightarrow visit left node and right (2n)

visit distance above current level of push on

return node = (left) \rightarrow (right), next

visited before in visited

return trip off (Cn)

new node distance \rightarrow return current left off on

the distance off in set block above

the distance off in set block below

? (push, take block & travel) \rightarrow (push, *block & travel + (block, travel) \rightarrow travel *block & travel.

? JUH visitor ($\text{JUH} = \text{travel}$) ?

? ($\text{JUH} = (\text{trip} \times \text{travel})$) ?

? ($\text{trip} \times \text{travel}$) \rightarrow MTHN \rightarrow MTHN

{ }

? $\text{JUH} = \text{medium} * \text{block & travel}$

? $\text{block} = \text{reference} * \text{block & travel}$

? ($\text{travel} = \text{prefixed}, \text{global}$)

Graph

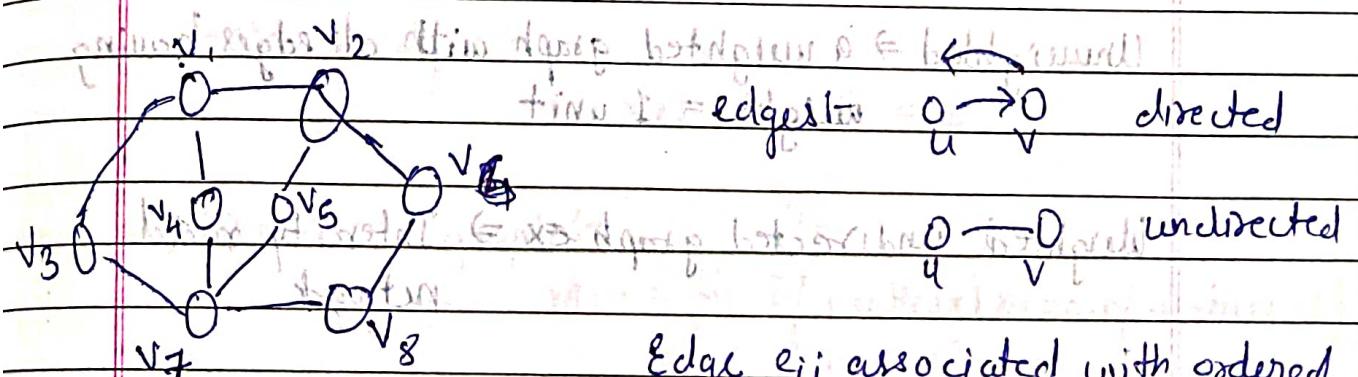
Graph \Rightarrow A graph G is an ordered pair of set V of nodes (vertices) and a set E of edges (set of edges).

$$G = (V, E)$$

order of edges in graph \Leftarrow data about direction of edges
 (ordered) pair $\Rightarrow (a, b) \neq (b, a)$ if $a \neq b$.

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

Note \Rightarrow Tree is a special kind of tree graph where
 we find if N nodes then $(N-1)$ edges one
 node with rest each form exactly one parent-child relationship
 (exactly one path from root to node).

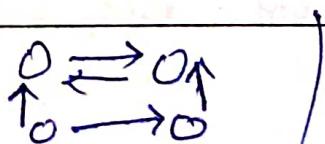


Edge e_{ij} associated with ordered pair (v_i, v_j)

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\} \quad |V| = 8$$

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

\Rightarrow Directed vs Undirected
 ↳ diagraph



+ Application

(application)

- (1) Social Network \rightarrow find mutual friend of a friend
length of shortest path from nodes equal to 2

$(E, V) = \emptyset$

- 2) World wide web \rightarrow directed graph

data structure \rightarrow web crawling, (graph traversal)
 $\{s, d\} = \{s, d, n\} \rightarrow$ using breadth first search

⇒ Weighted vs Unweighted

Weighted graph is a graph having edges with weight

↳ If each edge or each vertex has both are associated with some value, then the graph is called as weighted graph.

Unweighted \Rightarrow a weighted graph with all edges having

weights = 1 unit

Weighted undirected graph ex \Rightarrow Intercity road network

Unweighted undirected graph ex \Rightarrow Social network

⇒ Properties of Graph

$|V| = \text{no. of vertices}$ $V = \{v_1, v_2, v_3, v_4, v_5\} \Rightarrow |V| = 5$

$|E| = \text{no. of edges}$ $E = \{e_1, e_2, e_3\}, \{v_1, v_2\}, \{v_2, v_3\} \Rightarrow |E| = 3$

backtracking in traversing

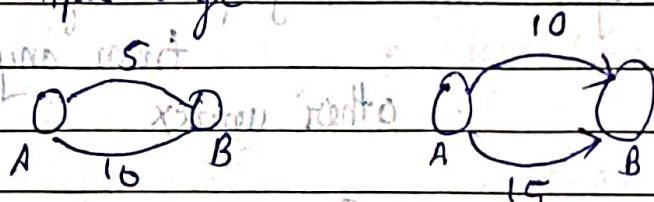
depth first



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



⇒ Multiedge or Parallel edge :- If there are more than one edge between two vertices then those edges called as parallel edges or multiple edges.



⇒ Simple graph ⇒ no self loop or parallel edge.

No. of edges :-

$$\begin{aligned} \text{Min} &= 0 \\ \text{Max} &= (\text{no. of vertices}) \times (\text{no. of vertices} - 1) \\ &= n(n-1) \end{aligned}$$

⇒ directed graph :-

$$\text{Min} = 0$$

$$\text{Max} = n(n-1)$$

$$\text{Outdegree} \leq |E| \leq \text{Indegree}$$

$$\Rightarrow 0 \leq |E| \leq n(n-1), \text{ if undirected}$$

$$\text{and } 0 \leq |E| \leq n(n-1), \text{ if directed}$$

assuming no self loop or multiedge

Dense graph = too many edges

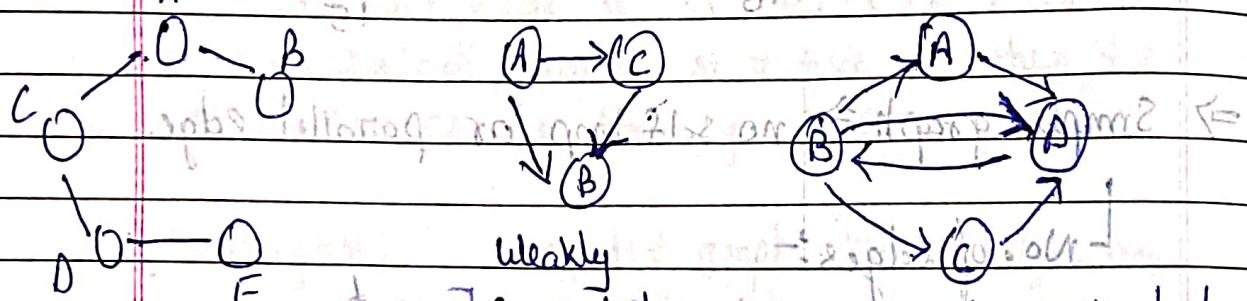
Sparse graph = too few edges.

⇒ 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



$\phi = \exists \text{ Connected} = \forall v \in V \text{ Strongly connected}$

[No path for

B to A or
B to C]

Cycle

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

Condition: $(I-n)n > |E| \geq 0$

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

⇒ Acyclic graph ⇒ a graph with no cycle

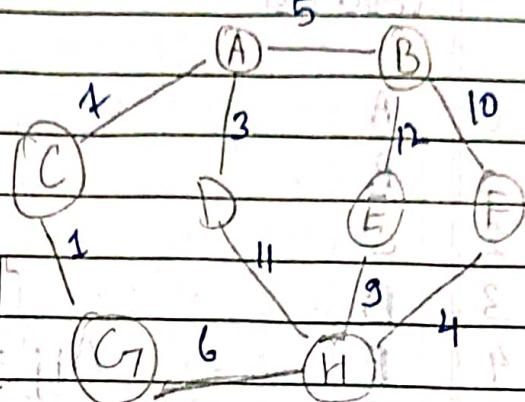
⇒ Directed Acyclic graph (DAG), ex → tree.

Graph Representation

1) Edge List

Vertex list

Edge List



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

$$\text{Space Complexity} = O(|V| + O(|E|) = O(|V| + |E|)$$

Time Complexity

Operation

Running Time

O(|V|) in finding adjacent nodes with a graph

(check if given nodes are part of same connected component)

$O(|E|) \approx$

$(|V|)^2 = \text{worst case } O(|V|^2)$
Not okay.

→ Need to figure it out

bottom of page out of $|V|$

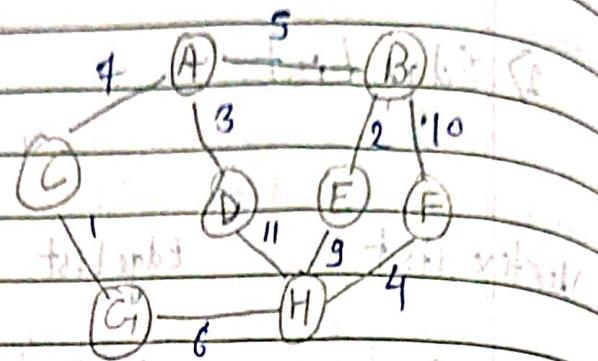
Symmetry $A_{ij} = A_{ji}$

Date _____ Page no. _____

2) Adjacency Matrix

Vertex List

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



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

Adjacency Matrix

Intermediate points	0	1	2	3	4	5	6	7
Matrix is initialized	0	∞	5	7	3	∞	∞	∞
Symmetry	1	5	∞	∞	∞	2	70	∞
	2	7	∞	∞	∞	∞	1	∞
	3	3	∞	∞	∞	∞	∞	11
	4	∞	2	∞	∞	∞	∞	9
(i) + (ii)	5	∞	10	∞	1	∞	∞	4
	6	∞	∞	1	∞	∞	∞	6
	7	∞	∞	6	11	9	4	∞

Time Complexity

(i) finding adjacent nodes $\Theta(V^2)$ $\Theta(V)$

(ii) finding if two nodes are connected $\Theta(V^2)$ $\Theta(V)$

are connected

Space Complexity

Space Complexity = $\Theta(V^2)$

Good if graph is dense or

V^2 is too less to matter.

$O(E)$ space complexity

Date: / / Page no: _____

4.3) Adjacency List

Implementation of Adjacency List

Adjacency List

Vertex List	Adjacency List
0 A	0 400 → 1 450 → 2 500 → 3 0 ↘
1 B	1 0 → 4 0 → 5 0 ↘
2 C	2 0 → 0 ↘
3 D	3 0 → 0 ↘
4 E	4 1 → 7 1 ↘
5 F	5 1 → 7 1 ↘
6 G	6 2 → 7 1 ↘
7 H	7 3 → 4 1 → 6 1 → 5 0 ↘

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

struct Node {
 int data;
 struct Node* next;

};

struct Node* A[8];

Most graphs are sparse

Unknown with $|E| \ll |V| \times |V|$

dropped in when $|V| \gg |E|$

⇒ Time Complexity

find adjacent nodes

$O(|V|)$

find if two nodes
are connected

: $O(|V|)$ worst case

worst : $O(|V|)$ best case

FindAdjacent = pre-adjacency

: adjacent after visit

(true) ↗

→ true ↗

$O = \text{short trade}$

→ Depth first Traversal

The depth first search (DFS) 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 algorithms.

→ 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 to search the

Coding Pythonic tools

Global or class scope variable

n = no. of nodes in graph

g = adjacency list representing graph. using list not dict

visited = [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

dfs(startnode).

s the most fundamental
place nodes and edges
a time complexity of
a building block in other

a graph without
next until it
which point it

with the
which point it
shah

graph with

the first
the first
the first

abgerument 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 `dfs (node *temp, int month)`

for ($i = 1$ to total user)

{ visited [i] = 0 }

→ condition for print

visited [$temp \rightarrow index$] = 1; if ($month == temp \rightarrow mm$)

$temp = temp \rightarrow next;$

{ cout << --- }

while ($temp \neq \text{NULL}$)

{ } ;

{ if (visited [$temp \rightarrow index$] == 0)

{ } ;

dfs ($temp$, month)

$temp = temp \rightarrow next;$

{ }

{ }

⇒ BFS (Breadth First Traversal) $l[0] = 0 = \text{shah} \rightarrow \text{tri}$

⇒ BBS (Breadth First Search) $l[0] = 0 = \text{shah} \rightarrow \text{tri}$

(total node at 0 = 1 tm) not

$o = [i] \text{ btsiv}$

{ }

$l[0] = q = 4^{\infty}, \text{ first } \times \text{ short}$

$\text{mm} \rightarrow l[0] = q = \text{triumm} \rightarrow \text{num} \rightarrow \text{triumm} \rightarrow \text{num}$

$l[1] = [xshah \leftarrow q] \text{ btsiv}$

$xshah \leftarrow q = shah$

$xshah \leftarrow q = shah$

$l[0] = \text{num}$

Implementation of Augmented DFS algo.

- Application :-
- i) Compute a graph's minimum spanning tree.
- ii) Detect & find cycles in a graph.
- iii) Check if a graph is bipartite.
- iv) Find strongly connected components.

Implementation can also be done with without it but many more.

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

for (j = 1 to total user)

{ if (temp->visited[j] == 0) { visit node j }

→ condition for print

 temp->visited[j] = 1; // visited node j
 if (month == temp->month) { cout << j }

 temp = temp->next;

{ cout << -- }

 while (temp != NULL) { visit node j }
 if (temp->visited[j] == 0) { visit node j }

 temp->visited[j] = 1; // visited node j
 dfs (temp, month)

 temp = temp->next;

}

}

⇒ BFS (Breadth First Traversal)

⇒ BPS (Breadth First Search)

o = [] btree

F0778 = q = front * short

front = tot - q = maximum - min = maximum - k + m

{ i = [x] btree

x = init - a = sub n

\Rightarrow Breadth first Search (BFS). (1 -> vector list)

Time complexity = $O(V+E)$

\Rightarrow Useful in finding the shortest path on unweighted graph

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

\Rightarrow The BFS algorithm uses a queue(data) structure to track which node to visit next. Upon reaching a new node, the algorithm adds it to the queue to visit it later.

BFS (\$).

{ queue q;

int Q,

int index = 0 = index1;

int max_comment = 0 = min_comment;

for (mt i = 0 to total_user)

{

visited[i] = 0

}

node *temp, *p = gr[0];

max_comment = min_comment = p \rightarrow tot_comm;

visited[p \rightarrow index] = 1;

index = p \rightarrow index;

index1 = p \rightarrow index;

q.enqueue(p);

while (!q.isEmpty()) { // fast node visited }

{

temp = q.dequeue();

p = temp \rightarrow next;

while (p != NULL)

{

if (visited[p \rightarrow index] == 0)

{

max-comm = p \rightarrow tot-comm;

index = p \rightarrow index;

}

if (p \rightarrow tot-comm < min-comm)

{

min-comm = p \rightarrow tot-comm;

index1 = p \rightarrow index;

} // if (min-comm - max-comm <=

visited[p \rightarrow index] = 1;

(fast node visited) q.enqueue(p); // state G

tot-comm++; // fast node visited by direct enqueue fast

next pointer p = p \rightarrow next; // fast node

visited[p]++; // fast node visited by direct enqueue

}

gr[index]

cout << "at index " << gr[index] << tot-comm

total-comm << " at index " << gr[index] << tot-comm.

37/31/31

if not 0 in value means repeat

txt having at least one value

⇒ Dijkstra's shortest Algorithm

⇒ Single Source Path (SSP) 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.

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.

- ⇒ Dijkstra's shortest Algorithm
- ⇒ Single Source Path (SSP) shortest Path algorithm
- ⇒ Time complexity = $O(E * \log(V))$ // depends on implementation & data structure used
- ⇒ Constraint
 - 1) Only contains 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 spSet (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.
 - 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 include 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 ($d(u) + c(u,v) < d(v)$) } $d(v) = d(u) + c(u,v)$ }

Ex. 3) $\{ (3, \text{pol} \times 3) \} \cap \text{minimum weight edges}$

$\{ (V, \text{pol} \times 3) \} \cap \text{minimum weight edges}$

pol \Rightarrow common in $\{ (3, V) \} \cap \text{minimum weight edges}$

unvisited & present price \rightarrow min price \rightarrow min price

item \rightarrow min price \rightarrow min price

\Rightarrow 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}

\Rightarrow Minimum Hence, a spanning tree does not have cycles and it cannot be disconnected.

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

\Rightarrow Prim's Algorithm

i) greedy algorithm

types \Rightarrow i) lazy version $O(E * \log(E))$

ii) eager version $O(E * \log(V))$ {using advanced data structure}

Time complexity = $O(V^2)$ in normal case by

using array & adjacency matrix

\Rightarrow Algorithm.

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

ii) 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 vertex so that it is picked first.

⇒ 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 cycles and it cannot be disconnected.

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

⇒ Prim's Algorithm

i) greedy algorithm

types → i) lazy Version $O(E * \log(E))$

ii) eager version $O(V * \log(V))$

{using advanced data structure}

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

⇒ Algorithm

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

ii) 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 vertex so that it is picked first.

3) While mstSet doesn't include 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 update the key value, iterate through all adjacent vertices. For every adjacent vertex v, if weight of edge $u \rightarrow v$ is less than the previous key of v, update the key value as weight of $u \rightarrow v$.

\Rightarrow Kruskal's algorithm.

1) greedy algorithm (O(E log E))

2) use set & Union-find Algorithm (path compression)

common for both first O(n^2) then O(n log n) then O(n log n)

\Rightarrow Time Complexity $\Rightarrow O(E \log E)$ or $O(E \log V)$

\Rightarrow 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.

otherwise discard it. else add it to the spanning tree.

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

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

⇒ 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 orderings 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 Graphs (DAGs) 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 m-degree by 1 for all its neighboring nodes.
- 3) If m-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.
- ⇒ Key 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 use ~~what are~~ 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

~~hash function must be good~~

the hash function must be 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 enforce hashable property.

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

~~hash function must be uniform & hashable~~

* 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 Techniques

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

1) Separate Chaining deals with hash collisions by maintaining a data structure (usually a linked list) to hold all the different values which hashed to a particular value. ~~with two pointer start~~

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

Hash Collision \Rightarrow Hash functions are not perfect, therefore sometimes two keys k_1, k_2 ($k_1 \neq k_2$) has to the same value.

$$H(k_1) = H(k_2)$$

Date: / / Page: / / Date: / / Page: / /

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

Separate chaining or insertion without direct shifting

Complexity:- Operation Average Worst

Insertion $O(1)^*$ $O(n)$

Removal $O(1)^*$ $O(n)$

Search $O(1)^*$ $O(n)$

Separate chaining is not limited to a linked list of nodes in memory

linked list of nodes in memory

\Rightarrow Note \rightarrow Separate Addressing is not limited to a linked

list of nodes in memory

separate addressing is not limited to a linked list of nodes in memory

and etc

but as Separate Chaining Ex \Rightarrow it's not limited to a linked list of nodes in memory

bring in separate addressing of separate parts

Address	Value	Address	Value
0	Magnik, 26	1	Sahu, 18
2	AAB, 18	3	Will, 21
4		5	Leah, 18
			Lara, 34
			Mark, 40

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

→ Open addressing: avoiding bottleneck phenomena

→ 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)

→ Grid size information (2D array)

Probe Probing Sequences

i) Linear probing $\Rightarrow P(x) = ax + b$

offset = $i \pm j^2$ where a & b are constant

ii) Quadratic probing $\Rightarrow P(x) = ax^2 + bx + c$

offset = $(5+i)^2$ where a, b & c are constant

iii) Double hashing

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

iv) Pseudorandom number generator

$P(k, x) = x * \text{RNG}(H(k), x)$, where RNG is random generator

function seeded with $H(k)$

⇒ General Idea: $x \rightarrow P(x)$ not directly possible

$x = 1, \text{to hash}$ $\rightarrow x \rightarrow \text{keyHash} \rightarrow \{1, 2, \dots, N\} \rightarrow \{1, 2, \dots, N\}$

KeyHash = $H(k)$, index = keyHash

while $\text{table}[index] \neq \text{null}$:

loop to size { index = $(\text{keyHash} + P(k, x)) \bmod N$

$x = x + 1$

insert (k, v) at $\text{table}[index]$.

- ⇒ Randomly selected probing sequences modulo N will produce a cycle shorter than the table size
 ↳ This leads to problem of infinite loop!
- ⇒ We fix it by avoiding this by choosing probing functions to those which produce a cycle of exactly length N^* (few exceptions are there)

⇒ Linear Probing (LP)

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

$$P(x) = ax + b, \text{ where } (a \neq 0), b \text{ are constants.}$$

b is obsolete

- ⇒ Which value(s) of the constants a in $P(x) = ax$ produce a full value cycle modulo N ? (so that we avoid infinite loop)

Ans → If $a \& b \mid N$ are relatively prime (i.e. their greatest common factor (GCD) is equal to one).

- ⇒ A common choice for $P(x)$ is $P(x) = 1x + 0$, since $\text{GCD}(N, 1) = 1$, no matter 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 $\text{GCD}(N, a) = 1$ still holds.

$\alpha = \text{threshold value} = \text{load factor} \times \text{size of table (N)}$

2) Quadratic Probing (QP)

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

$$P(x) = ax^2 + bx + c, \text{ where } a, b, c \text{ are constant}$$

~~and condition is $a \neq 0$; (constant c is obsolete)~~

→ Decide probing function with formulae or algorithm.

i) Let $P(x) = ax^2$, to keep the table size a prime

so that size of table is a number ≥ 3 and also keep $a \leq 1$,
to avoid left side has bit shift instead of mod (n, a).

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

iii) $P(x) = (x^2 + x) * N$, to keep the table size a prime N where $N \equiv 3 \pmod{4}$.

and this probing (QP) will be continuous modulo [ex: $N = 23$]

and after 23rd mark after starting sequence

3) Double Hashing (DH)

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

$$P(k, x) = x * H_2(k), \text{ where } H_2(k) \text{ is a second hash function}$$

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

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

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

If $S=0$, then set $S=1$.

Notice $1 \leq S \leq N$, i.e., $\text{GCD}(S, N) = 1$ since N is prime.

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

\Rightarrow Removing an element from Open Addressing

1- 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 both 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.

All 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 subtrees in that binary tree.

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

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

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 subtrees in the binary tree.

OR

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

$$A[\text{Parents}(i)] \leq A[i].$$

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). It is also called min-heap.

⇒ Array based Representation [index starts at 0]

Parents node $\Rightarrow (i-1)/2$

left child $\Rightarrow (2*i) + 1$

right child $\Rightarrow (2*i) + 2$

⇒ Complexity = $O(n \log n)$

⇒ Applications

1) Heap Sort (Heap means not \leq 90)

2) Priority Queue

3) Graph Algorithms

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

task sort to traverse not sort heapify not \Rightarrow max heap

// Heapsort Method \Rightarrow Complexity $\geq O(n \log n)$

function void heapify (int arr[], int n, int i)

{

int largest = i;

int left = 2*i + 1; int mid = 2*i + 1;

int right = 2*i + 2; int s = 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[r], arr[largest]);

 heapify (arr, n, largest);

};

};

left Leaf nodes starts from floor($n/2$) [Index starts = 0]

void build-maxheap (int Arr[]) // complexity $\rightarrow O(n)$

{

for ($i = N/2-1$; $i \geq 0$; $i--$)

{

maxheapify (Arr, i);

}

void heap-sort (int Arr[])

int heap_size = N;

build_maxheap (Arr);

$O(n \log n)$

for (int i=N; i>

for (int i=N-1; i>0; i--)

{

swap (arr[0], arr[i]);

maxheapify (arr, i, 0);

}

}

left Leaf nodes starts from floor($n/2$) to constuct maxheap

from bottom to top

remove and arr . popping off the

bottom of the maxheap with the

maxval (or root) back to the