

Introduction to Data Structures

- * Data - Raw facts
- * Information - Meaningful data.

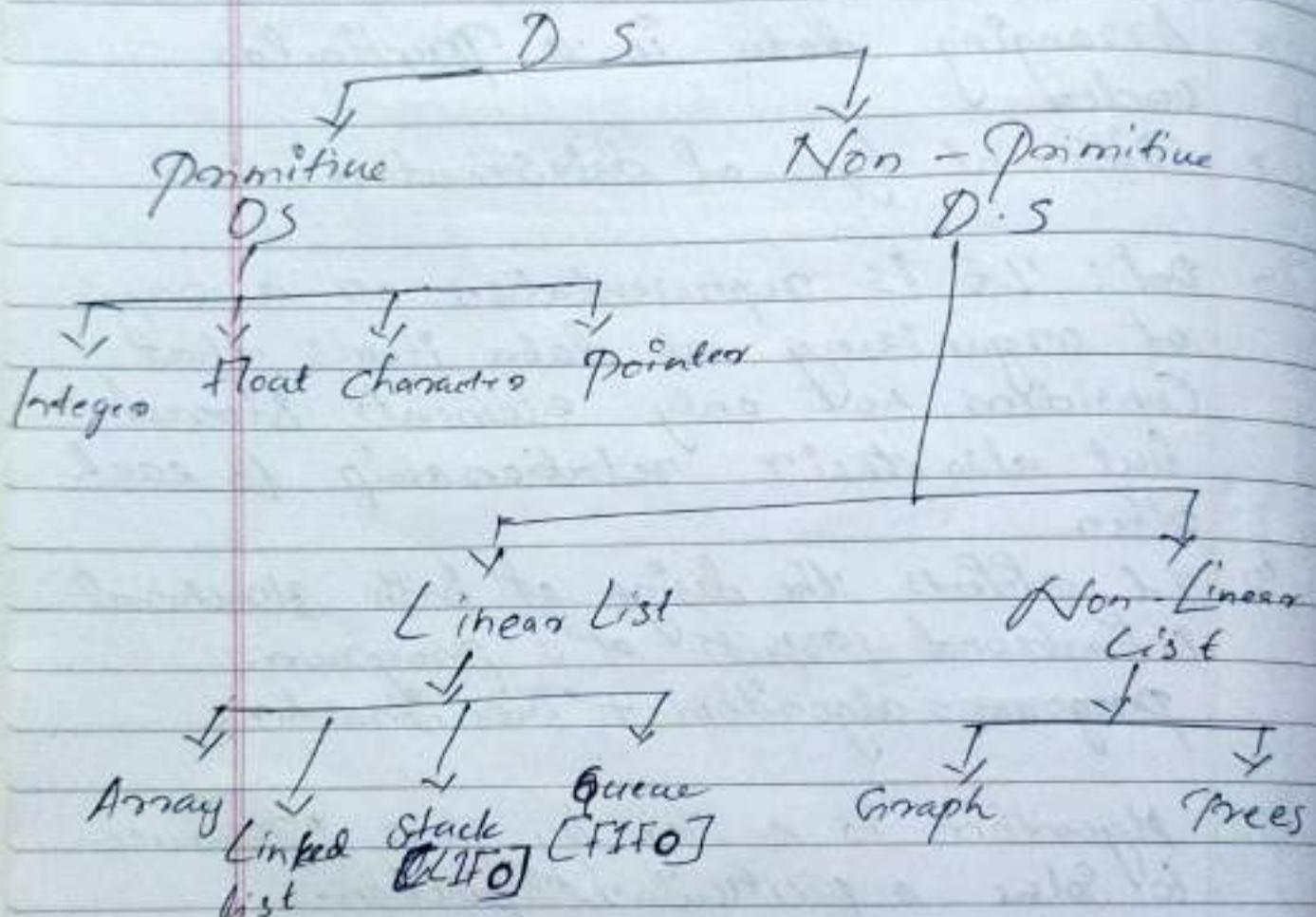
Data Structures

- * Arranging data in a particular order.
 - * Different types of Data Structure.
- ↳ Def: It is representation or a way of organizing all data items that considers not only elements stored but also their relationship to each other.
- ↳ It affects the design of both structural & functional aspects of a program.
program = algorithm + Data Structure
- ↳ Algorithm is a step by step procedure to solve a particular function.
- ↳ To develop a program or an algorithm we should select an appropriate data str

for that algorithm

Classification of Data Structure

- ↳ Primitive D.S
- ↳ Non-Primitive D.S



Primitive DS

- * Data
- * Basic Structure that directly operate upon the machine instructions.
- * Built-in or predefined datatypes and can be used directly by user to declare variables.
- * Non Primitive
- * More Sophisticated Data Structure.
- * Derived from primitive D.S
- * Emphasize on Structuring of a group of homogeneous or heterogeneous (diff type) data items.

Arrays and Pointers

Arrays

- * An array is a data-structure containing a no. of data values all of which are of same type.

Visualize Array

$$A = \boxed{1 \ 1 \ 1 \ 1}$$

- Collection of smaller block of memory and each block capable of storing a data value of same type.

One dimensional array

Variable
 $a [5 | 10 | 15 | 20 | 25 | 30]$

Syntax for declaration

datatype name of array [no. of elements]

ie; $\text{int arr}[5]$

ie; $\text{arr} [\quad | \quad | \quad | \quad |]$

- It means that compiler allocate 5 location.

Size = $5 \times \text{size of } (\text{int})$

ie; Integer takes 4 bytes of location.

Variant of declaring

ie $\text{int arr}[a, b]$

- * The length of array cannot be negative.

Note: Specifying length

```
#define N 10 // Macro for finding
int arr[N]; // size limit
```

Accessing

arr[0], arr[1], arr[2]

Initializing

```
y a[5] = {1, 2, 3, 4, 5}
y a[] = {1, 2, 3}
```

* for loop -

```
int arr[5];
for(i=0; i<5; i++)
    scanf("%d", &a[i])
}
```

Designated Initialization

```
int arr[10] = { [0]=1, [5]=2, [6]=3 };
```

Size Calculation

* Size of (name of array[0])

↳ for finding the size of array

$$\text{length} = \frac{\text{size of (name of array)}}{\text{size of array}[0]}$$

Multidimensional array

* array of array

datatype name of arr [size₁,][size₂] ...
size_n

Size calculation: Multiplying the size of all dimensions.

ex: size of a[10][20]

$$= 10 \times 10$$

$$= 20$$

$$= 20 \times 4 (\text{size})$$

$$= 80 \text{ bytes}$$

Two dimensional array

Int a[4][5]

↳ 4 rows, 5 columns

Size of a[4][5]

$$= 20 \times 4 = 80 \text{ bytes}$$

size
of
element

Initialize

Int a[2][3] = {1, 2, 3, 4, 5, 6}

Access 2D array

- * Using row index and column index
ie. a[0][1] = 2

	0	1	row	cols
0	1	2	/	
1	3	4	/	

Looping

2 loops

i - rows

j - column

Three dimensional Column

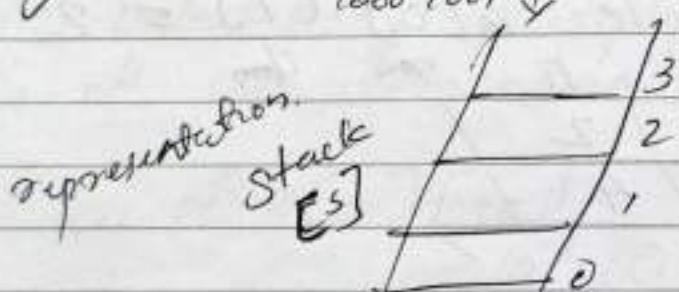
on file a [1][2][3]

25/11 Stack LIFO

- * Type of data structure
- * operation is ~~push~~ up and up

Implementation of Stack using array

Array a[5] = 



4 operations

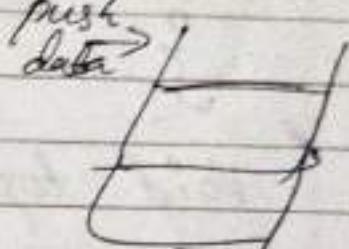
Void main ()

```
{
    push()
    pop() // using switch (ch)
    top()
    display()
}
```

Case 1 : push
 Case 2 : pop
 ...

define N 5 ← Macro

```
int stack [N]
int top = -1;
void push()
{
    int a;
    printf("enter data")
    scanf("%d", &a);
    if (top == N-1)
    {
        printf("overflow")
    }
    else
    {
        top++;
        stack [top] = a;
    }
}
```



Date _____
Page _____

2. void pop ()

{ int item;

if (top == -1) // No data in the
list

{ under flow

}

else

{ item = stack[top];

top --;

printf ("%d", popped item, item);

}

3. void top () // getting the top
most element.

{ if (top == -1) // empty

{ stack is empty

}

else { printf ("%d", stack[top]);

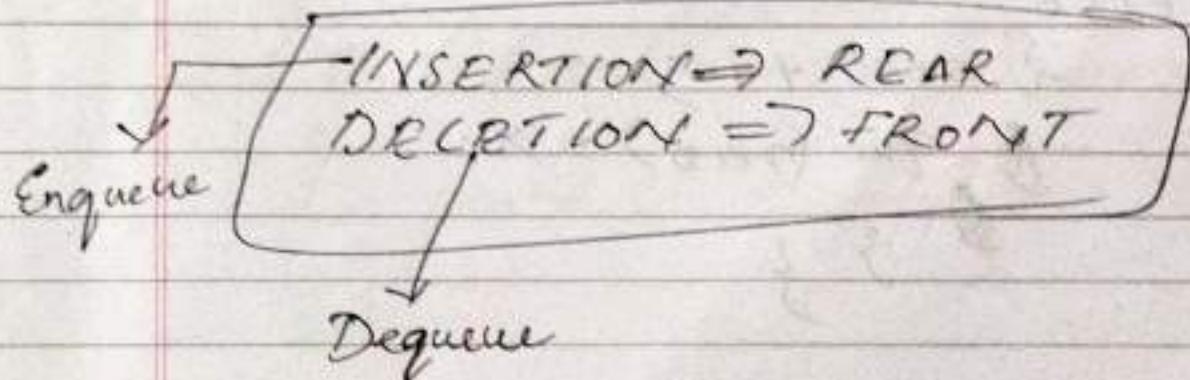
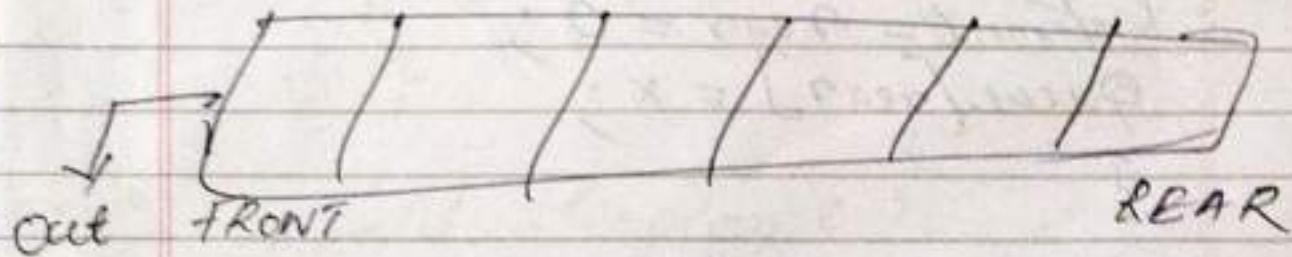
}

```

4. void display()
{
    int i
    for(i=top; i>0; i--)
    {
        printf("%d", stack[i]);
    }
}

```

II Queue [FIFO]



classmate
Date _____
Page _____

	0	1	2	3	4	5
int arr[6];	a	/	/	/	/	/

```

#define N 5
int queue[N]
int front = -1, rear = -1; // queue empty
Enqueue()
void enqueue (int x)
{
    if (rear == N-1)
    {
        printf ("Queue is full");
    }
    else if (front == -1 && rear == -1)
    {
        front = rear = 0;
        queue[rear] = x;
    }
    else
    {
        rear++;
        queue[rear] = x;
    }
}

```

dequeue()

```

void dequeue()
{
    if (front == -1 && rear == -1)
        { cout << "underflow" }
    else if (front == rear)
        { front = rear = -1; }
    else
        {
            cout << "element deque" << queue[front];
            front++;
        }
}

```

display()

```

void display()
{
    if (front == -1 && rear == -1)
        { cout << "empty" }
}

```

else

```
{ for (int i = front; i < rear + 1;  
      i++)  
{ printf("%d", queue[i]);  
}
```

~~top()~~

Void top()

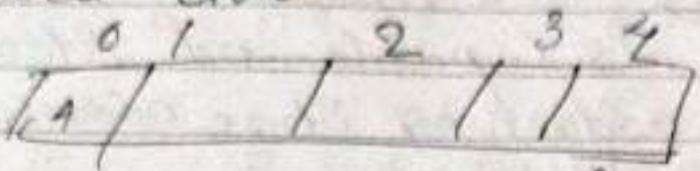
```
{ if (front == -1 && rear == -1)  
{ printf("empty")  
}
```

else

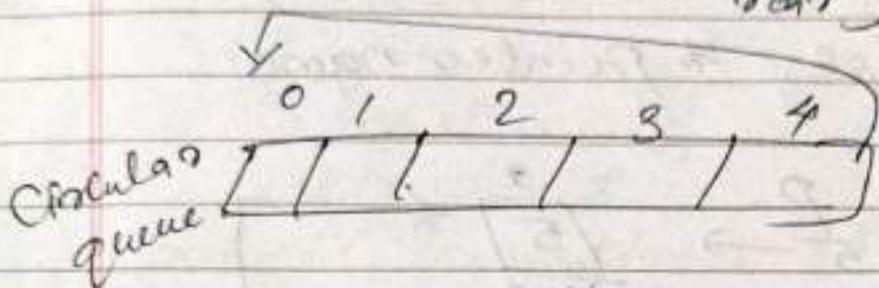
```
{ printf("%d", queue[front]);  
}
```

Drawback

- * we can also represent it using linked list.



front } enqueue call
rear } $\rightarrow (N-1)$ no space



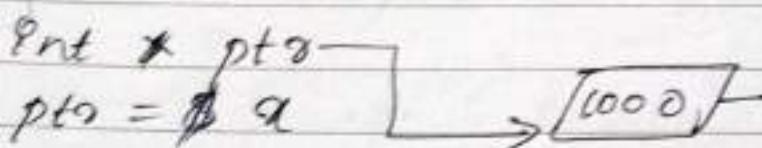
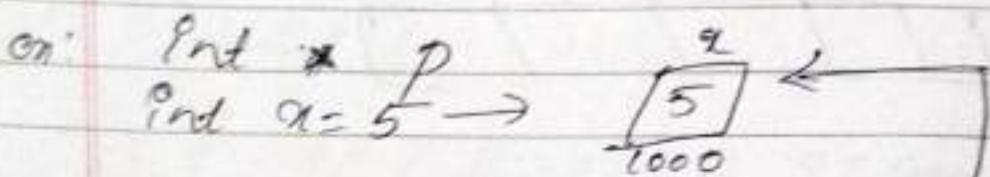
~~30/11/06~~

Array & Pointers

Pointers is a special variable which is capable of storing some address or store addresses rather than values.

Syntax of declaration

- * Datatype * Pointer name



- * Value of operation (*) / induction operator used to access the value stored at the location.

- * (*) is called dereference operator. It operates on a pointer and gives the value stored in that pointe.

int $a = 5$

int *ptr

ptr = & a

printf("%d", *ptr);

i.e; point the value of ptr
dereference operator

on: int main()

{
 int $i = 10$;

 int *p = & i ;

 printf("The address of variable i is
 %p", p);
 return 0;

}

on: int $b = 10$

int *p;

$p = \& b$;

① printf ("%d", b)

O/P = 10

② printf ("%d", *p)

$\&$ = address
 $*$ = referencing operator /
 value of pointer

3. $\text{printf}(\&p, \&b) / o/p = 200$
4. $\text{printf}(\&p, p) / o/p = 200$

\Rightarrow If we want to print the address of an array it points the base address.

Put $a[] = \{5, 2, 3\}$.

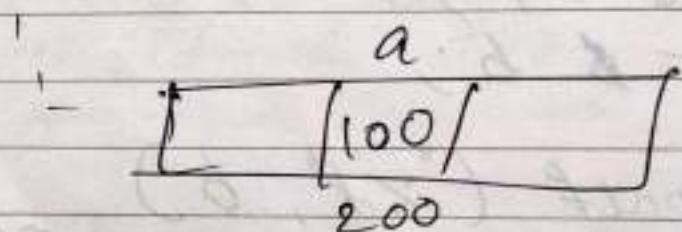
Put $*q$

$q = a$ // q contains base address of $a[0]$ i.e; $q = \&a[0]$ // q contains address 100

$a[0] \quad a[1] \quad a[2]$

1	2	3
100	101	108

address 108



1/2/20

Relationship between Arrays and Pointers

- An array is a block of sequential data.

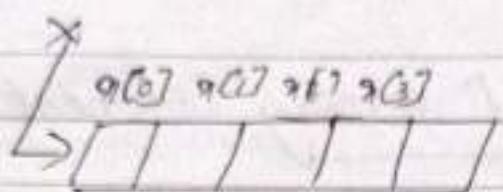
Addresses of array elements

```
#include <stdio.h>
int main() {
    int x[4];
    int i;
    for (i=0; i<4; i++)
    {
        printf("x[%d] = %p\n", i, &x[i]);
    }
    printf("Address of array x: %p", x);
    return 0;
}
```

O/P $\&x[0]$ = 1450734448
 $\&x[1]$ = 1450734452
 $\&x[2]$ = 1450734456
 $\&x[3]$ = 1450734460

Address of array x: 1450734448

- The address of $\&x[0]$ and x is same, but it is bar the variable name x points to the first element of the array.



- $x[0]$ is equivalent to $*x$
 $x[1]$ " " " $*(\text{?}+1)$
 $x[i]$ " " " $*(\text{?}+i)$

ex:-

Pointers and Arrays

```
#include <stdio.h>
int main()
{
    int i, *x[6], sum = 0;
    printf("Enter 6 numbers:");
    for (i=0; i<6; i++)
    {
        scanf("%d", &x[i]);
        // Equivalent to scanf("%d", &x[i]);
    }
}
```

```

    sum += *(s+i);
}
printf ("Sum = %d", sum);
return 0;
}

```

Q/P

Enter 6 numbers : 2

3

4

4

12

12

Sum = 29

Qn2

Arrays and Pointers

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

```
int main ()
```

```
{
```

```
int a[5] = {1, 2, 3, 4, 5};
```

```
int *ptr;
```

// ptr is assigned the address of
the third element.

```
ptr = &a[2];
printf("%*ptr=%d\n", *ptr);
printf("%*(ptr+1)=%d\n", *(ptr+1));
printf("%*(ptr-1)=%d", *(ptr-1));
return;
```

O/P

```
*ptr = 3
*(ptr+1) = 4
*(ptr-1) = 2
```

C Pass Addresses and Pointers

- In C programming, it is also possible to pass addresses as arguments to functions.
- To accept these addresses in the function definition, we can use pointers. It's been pointers are used to store addresses.

```
ex. #include <stdio.h>
Void swap (int *n1, int *n2);
Int main ()
{
    int num1 = 5, num2 = 10;
    // address of num1 and num2 is passed
    swap (&num1, &num2);
    printf ("num1 = %d\n", num1);
    printf ("num2 = %d\n", num2);
    return 0;
}

Void swap (Int *n1, Int *n2)
{
    Int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

O/P

num1 = 10

num2 = 5

Dynamic Memory Allocation

- * Dynamically allocate memory in your program using Standard Library functions:
 - (1) malloc()
 - (2) calloc()
 - (3) free()
 - (4) realloc()

These functions are defined in the `<stdlib.h>` header file.

i) malloc()

- * The name "malloc" stands for "my allocation".
- * It reserves a block of memory for the specified no. of bytes. And it returns a pointer of void which can be casted.

into pointers of any form.

Syntax

`ptr = (CastType *) malloc (size);`

Ex: `ptr = (float *) malloc (100 * sizeof (float));`

- ↳ The above statement allocates 100 bytes of memory. Because the size of float is 4 bytes. And the pointer `ptr` holds the address of first byte in the allocated memory.

Calloc()

- The name "calloc" stands for Contiguous allocation.
- The malloc() function allocates only and leaves the memory uninitialized. Whereas, the calloc() function allocates only and initializes all bits to zero.

C realloc()

- If the dynamically allocated my is insufficient or more than required, you can change the size of previously allocated my using the realloc() function.

Syntax: `ptr = realloc(ptr, x);`

```
ex: int main()
{
    int *ptr, i, n, n2; printf("Enter size");
    scanf("%d", &n);
    ptr = (int*)malloc(n * sizeof(int));
    printf("Addressess of previously allocated my:");
    for(i=0; i<n; i++)
        printf("%u\n", ptr+i);
    printf("Enter new size");
    scanf("%d", &n2); //reallocating the my
    ptr = realloc(ptr, n2 * sizeof(int));
    printf("Addressess of newly allocated my:");
    for(i=0; i<n2; i++)
        printf("%u\n", ptr+i);
    free(ptr);
    return 0;
}
```

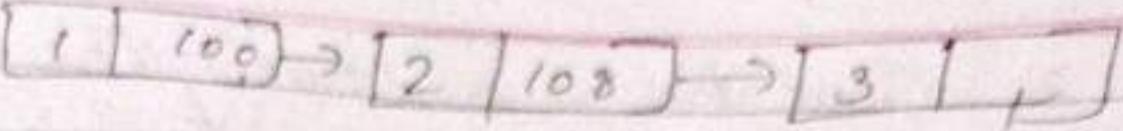
Linked List

memory

`int a;` // Compiler allocates []
`int a[3]` → [] Continuous location
`int a[100]` → Compiler allocates 100 * 4
100 bytes of memory.

- * If the programmer use only 10 location only, then remaining locations are wasted.
Another program cannot allocate these memory in future. This is the drawback of array.
To resolve this linked list introduced.
- * List is collection of items in which data items are not allocated in a continuous way.
- * To connect 1 with 2 then add extra space needed to store address

ex:



↓
data item
↓
address of next data item

- * we can perform:
 - ⇒ insert node at end
 - ⇒ insert node at middle
 - ⇒ insert node at beginning

? If we can add new node.

Struct node
{ int a;

 Data address

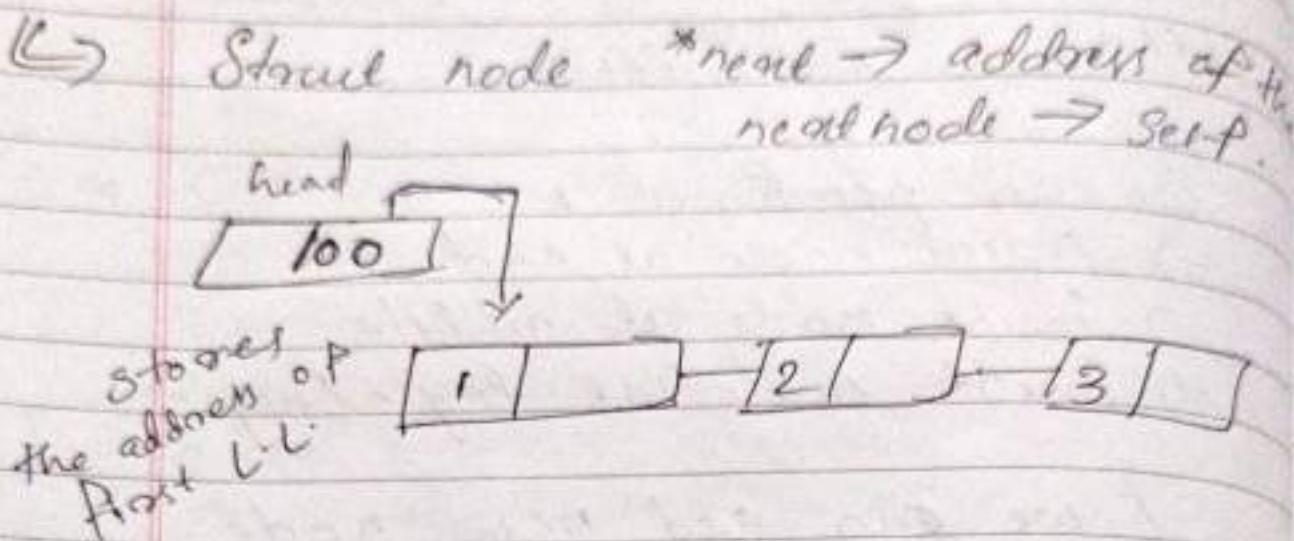
Struct node *next;
}

 ↳ self referential structure

* int *p; // 'p' stores the address of an integer value.

* char *s; // 's' stores " " of a float value.

* float *a ; it stores the address of a float value



Types of Linked list or Variants

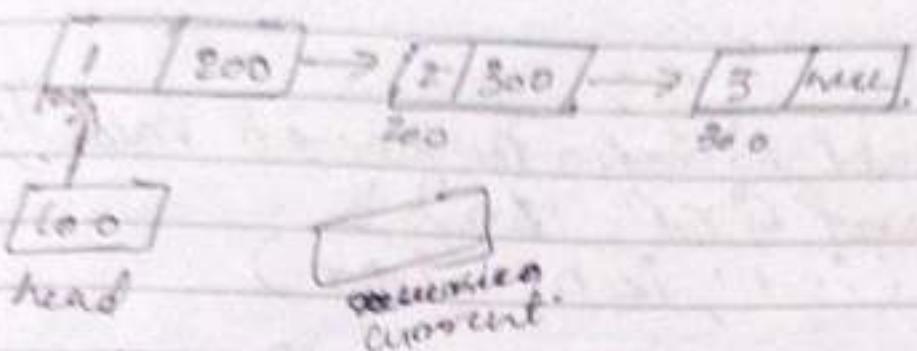
1. Single linked list
2. Doubly linked list
3. Circular linked list

2/12/20

classmate

for
topic
classmate

Singly Linked list



- ① 1) First step create node of the list

```
Struct node { [data/next]  
    int data;  
    Struct node *next  
};
```

- * Our aim is to create 3 such nodes
so ~~to~~ link them together.

```
#include <stdio.h>  
#include <stdlib.h>  
main()  
{  
    Struct node  
    {  
        int data;  
        Struct node *next;  
    };
```

• Struct node *newnode, *current,
* head = NULL

int i, n;

printf ("Enter the no. of values");

scanf ("%d", &n);

for (i=1; i<n; i++)

{ newnode = (struct node *) malloc (sizeof
(struct node));

scanf ("%d", &newnode->data);

newnode->next = head = NULL

if (head == NULL)

{ head = newnode;

current = newnode;

}

else

{ current->next = newnode

current = newnode;

}

Solved

```

printf ("Linked list")
for (current = head; current != NULL;
     current = current->next)

```

```

printf ("%d->%d", current->data);
printf ("NULL");

```

Traversal

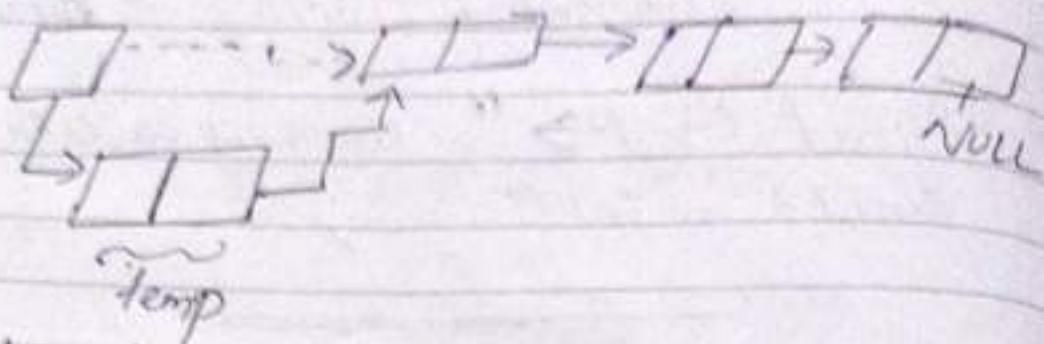
- * Moving through the list & visiting each node exactly once.

Traverse (head)

1. Set current = head
2. Repeat steps 3 to 4
while current ≠ null
3. Apply process (pointing) to DATA
(current)
4. Set current = Next (current)
5. exit .

3/12/20

Insertion in linked list



- * 'temp' is pointer which points to node which has to be created or inserted

$\text{temp} \rightarrow \text{info} = \text{data};$

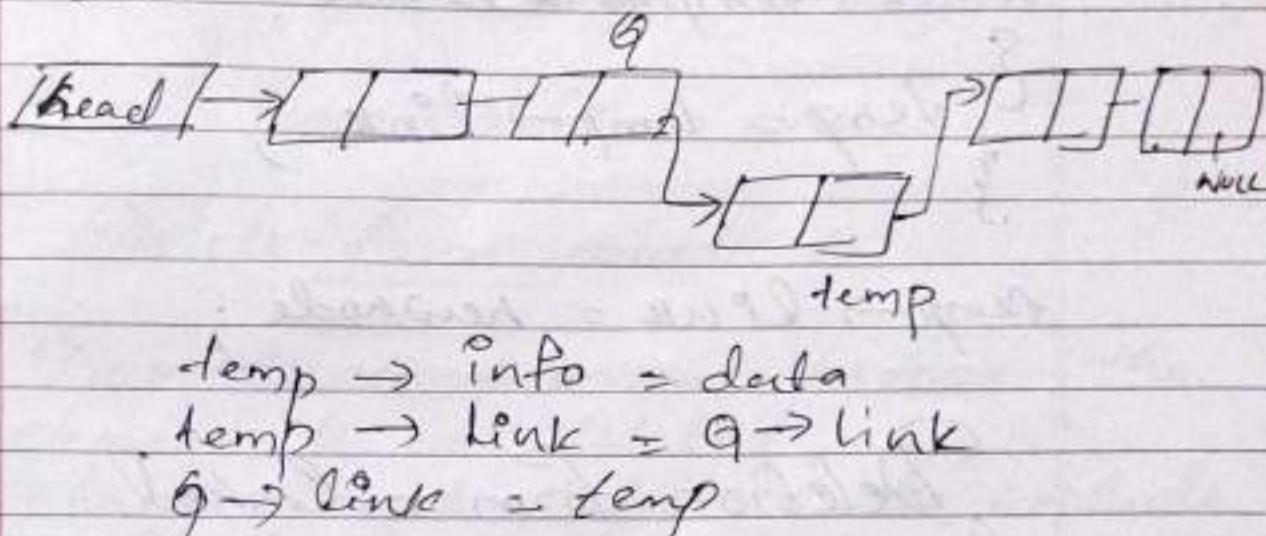
- * 'Start' points to the first element of linked list. For insertion at beginning, we assign the value of Start to link of temp node.

$\text{temp} \rightarrow \text{link} = \text{Start};$

- * Now inserted node is first node of L.L so Start reassigned as $\text{Start} = \text{temp}$

Case IIInsertion in between

- * First we traverse the linked list for obtaining the node . we obtain a pointer Q which points to the element after which we have to insert .

Case IIIInsertion at End

- * Allocate mly for new node
- * Store data
- * Traverse to last node
- * Change next of last node to recently

created node.

ex: Struct node * newnode;
 newnode = malloc (size of (struct node))
 $\text{newnode} \rightarrow \text{data} = 4;$
 $\text{newnode} \rightarrow \text{link} = \text{NULL};$

Struct node * temp = head;
 while ($\text{temp} \rightarrow \cancel{\text{link}} \neq \text{NULL}$)
 {
 $\text{temp} = \text{temp} \rightarrow \text{link};$
 }

$\text{temp} \rightarrow \text{link} = \text{newnode};$

Deletion from a Linked List

2

Delete from beginning

* point head to the second node

$\text{head} = \text{head} \rightarrow \text{link};$



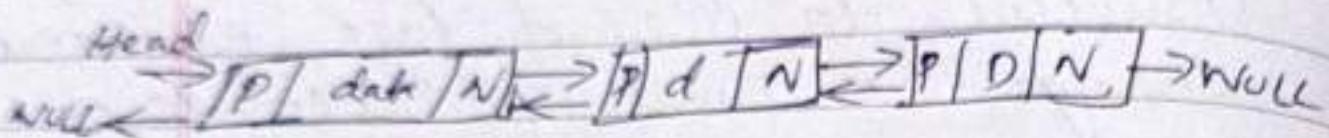
II Delete from end

- x Traverse to second last element
 - x Change its next pointer to null
- ex. Struct node *temp = head;
while (temp → link → link != NULL)
{
 temp = temp → link;
}
temp → link = NULL;

III Delete from middle

- x Traverse to element before the element to be deleted.
 - x Change next pointers to exclude the node from the chain.
- ex. for (int i=2; i< position; i++)
{
 if (temp → link != NULL)
 {
 temp = temp → link;
 }
 temp → link = temp → link → link;

Doubly Linked List



- * We add a pointer to the previous node in a doubly-linked list. Thus, we can go in either direction: forward or backward.

ex. Struct node {

 int data;

 Struct node * next;

 Struct node * prev;

}

Operations in D.L.L

1. Traverse

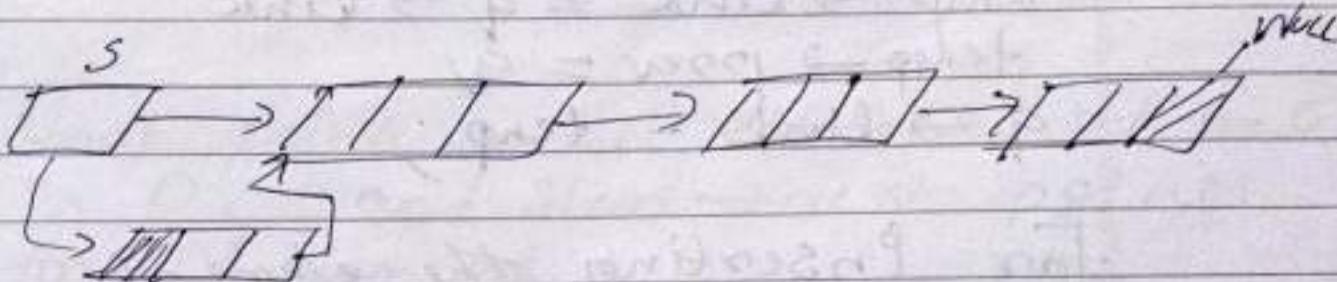
- * We take ptr as pointer variable.

- 6) Here Start points to the first element of list. Initially we assign the value of start to ptr so ptr also points to the first node of list.
- 7) for processing the next element, we assign the address of next node to ptr as
- $$\text{ptr} = \text{ptr} \rightarrow \text{next}$$
- 8) we can traverse until ptr has NULL value.

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

$\text{ptr} = \text{ptr} \rightarrow \text{next}$

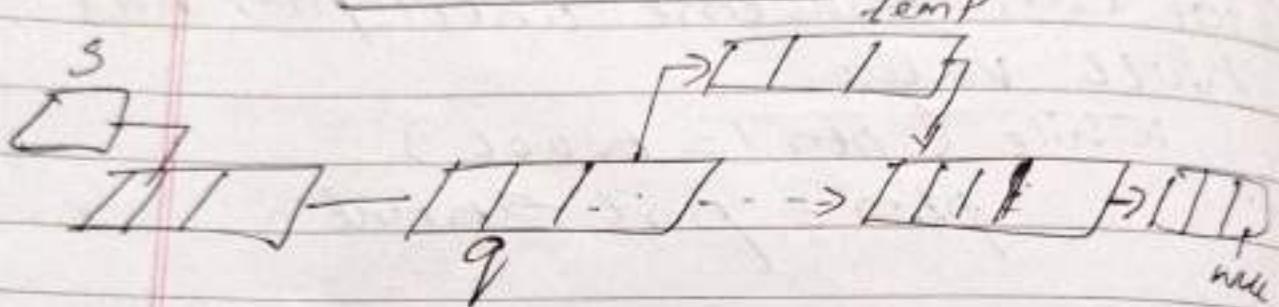
2. Insertion Into D.L.L



ex Start node * temp
 $\text{temp} = \text{malloc}(\text{sizeof(Start node)})$
 $\text{temp} \rightarrow \text{info} = \text{data}$
 $\text{temp} \rightarrow \text{prev} = \text{NULL}$
 $\text{temp} \rightarrow \text{Next} = \text{start}$
 $\text{start} \rightarrow \text{prev} = \text{temp}$
 $\text{start} = \text{temp}$

Case 2

Insertion in Middle



- * $q \rightarrow \text{link} \rightarrow \text{prev} = \text{temp}$
- * $\text{temp} \rightarrow \text{link} = q \rightarrow \text{link}$
- * $\text{temp} \rightarrow \text{prev} = q$
- * $q \rightarrow \text{link} = \text{temp};$

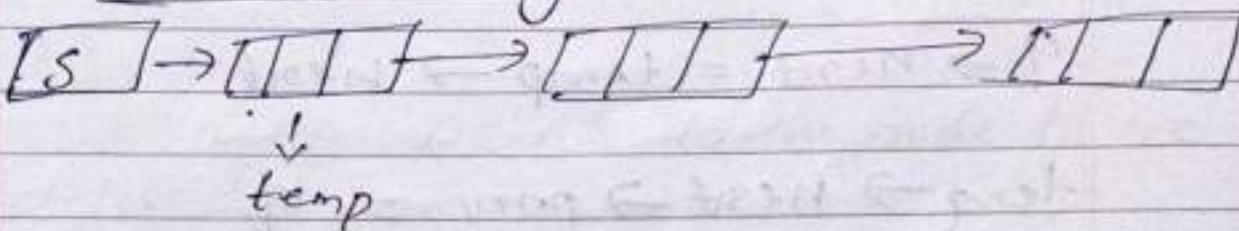
For inserting the element after the node we assign the address of inserted node to

the prev part of next node. Then we assign the next part of previous node to the next part of inserted node.

- * Address of previous node assigned to prev part of inserted node and address of inserted node will be assigned to next part of previous node.

(3) Deletion

Case: i) Deletion at beginning

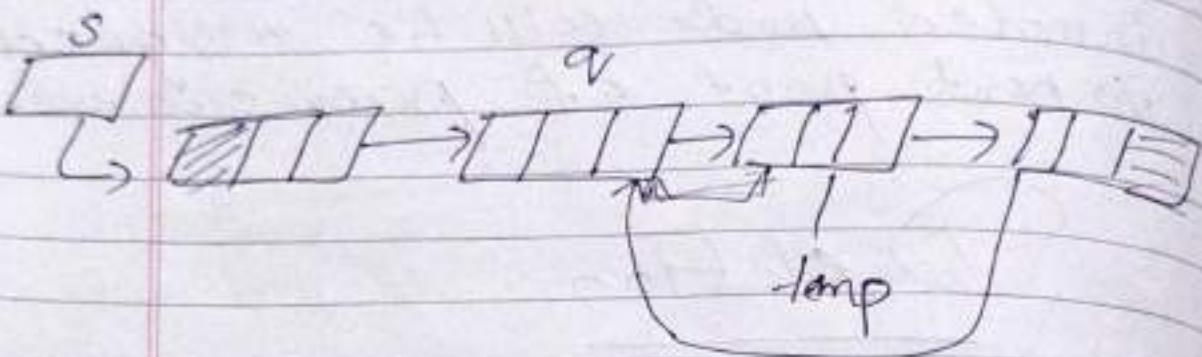


- * Start points to 1^{st} element of node in D.L. and Start \rightarrow next points to 2^{nd} element

$$\text{temp} = \text{start} \left(1^{\text{st}} \text{ element} \right)$$

start = start \rightarrow next (1st element)
start \rightarrow prev = null
free(temp);

Case 2 Deletion From Middle



temp = q \rightarrow next

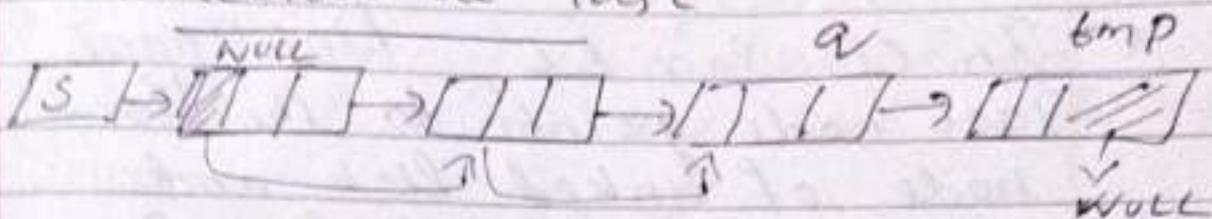
q \rightarrow next = temp \rightarrow next

temp \rightarrow next \rightarrow prev = q

free(~~q~~temp)

Case 3

Deletion at last



1) $\text{temp} = q \rightarrow \text{next}$

$\text{free}(\text{temp})$

$q \rightarrow \text{next} = \text{NULL}$

x 2) 'q' pointing to the previous node of node to be deleted.

After start .

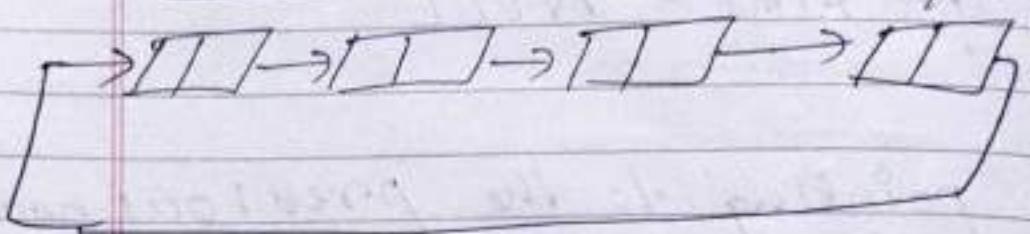
1. temp points to the node to be deleted
2. last node will be deleted
3. Second last node become the last node.

Circular Linked List

- * In Circular L.L., Link part of last node points to the first node of linked list and represent linked list in a Circular way -

Creation

first



- * New element can be added as

$\text{temp} \rightarrow \text{link} = \text{last} \rightarrow \text{link}$
 $\text{last} \rightarrow \text{link} = \text{temp}$
 $\text{last} = \text{temp}$

- * After that, link of added node will point to the first node of list.

3. Link of previous node added will point to added node and after Step ①, pointer variable "last" will point to added node which is now the last node of list.

Operations

Traversal

Traversal need a pointer variable which now points to 1st node of list. In circular L.L we maintain the pointer *last*, which points to last node. Link part of last pointer points to first node, so we assign the last link value to pointer variable, which point to 1st node.

$$q \rightarrow \text{last}$$

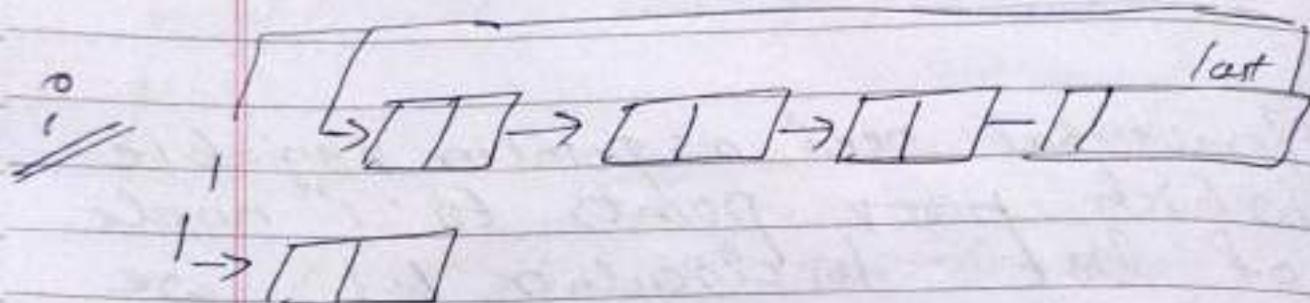
$$q - \text{last} \rightarrow \text{link}$$

Date _____
Page _____

```
while (q != last)
{
    q = q->link
}
```

② Insertion

- i Insertion at beginning
- ii Insertion in between.



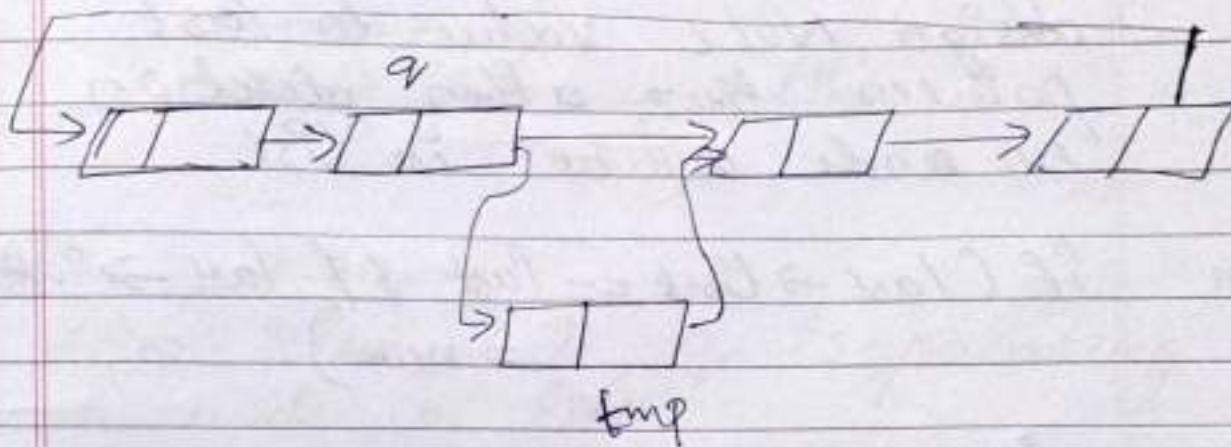
$\text{tmp} \rightarrow \text{data} = \text{data}$
 $\text{tmp} \rightarrow \text{link} = \text{last} \rightarrow \text{link}$
 $\text{last} \rightarrow \text{link} = \text{tmp}$

- x After Statement 2 inserted node points to previous node of list - and After Statement ③ 'link' part of last node

points to inserted node, which is
1st node of circular L.L.

Case 2

Insertion In between



$\text{tmp} \rightarrow \text{link} = q \rightarrow \text{link}$
 $\text{tmp} \rightarrow \text{info} = \text{num}$
 $q \rightarrow \text{link} = \text{tmp};$

3

Deletion - 4 Cases

- i If list has only one element
- ii Node to be deleted is 1st node
- iii Deletion In between

iv Node to be deleted is last node.

Case 1

Here we check condition for only one element of list then assign NULL value to last pointer bcz after deletion no node will be in list.

en if (last->link == last && last->info
= num)

{
 tmp = last;
 last = NULL;
 free (tmp);
}

Case 2: Delete at first

Here we assign the Link Part of deleted node to the Link Part of pointer last. So now

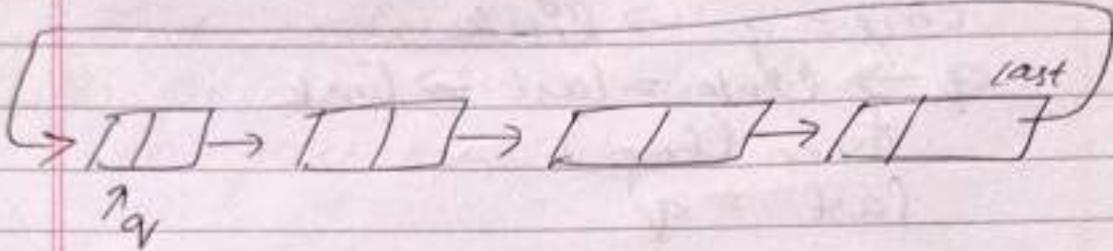
Link part of last pointer will point to the next node which is now first node of list after deletion

```

q = last->link
if (q->info == num)
{
    tmp = q;
    last->link = q->link;
    free(tmp);
}

```

After Statement ①, q is pointing to
1st node of List.



Case 3

Deletion in between

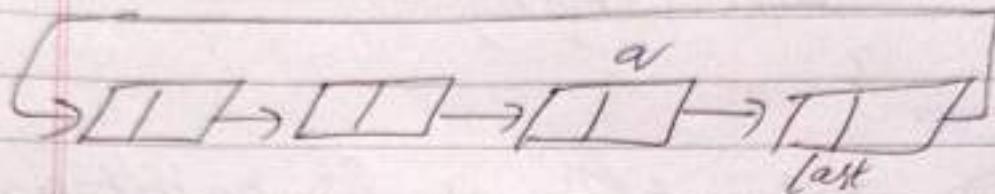
while ($q \rightarrow \text{link}! = \text{last}$)
{

classmate
Date _____
Page _____

if ($q \rightarrow \text{link} \rightarrow \text{info} == \text{num}$)
{
 tmp = $q \rightarrow \text{link}$
 $q \rightarrow \text{link} = \text{tmp} \rightarrow \text{link}$
 free (tmp)}

Case 4

Deletion at last



$\text{tmp} = q \rightarrow \text{link}$
 $q \rightarrow \text{link} = \text{last} \rightarrow \text{link}$
free (tmp)
 $\text{last} = q$

- * Deletion of last node requires to assign the link part of last node to link part of previous node.
- * So link part of previous node

will point to the first node of list.
Then assign value of previous node
to the pointer variable 'last' but
after deletion of last node pointer
variable 'last' should point to the
previous node.

- * Here q is pointing to previous node.

After Stmt ① : tmp will point to last node.

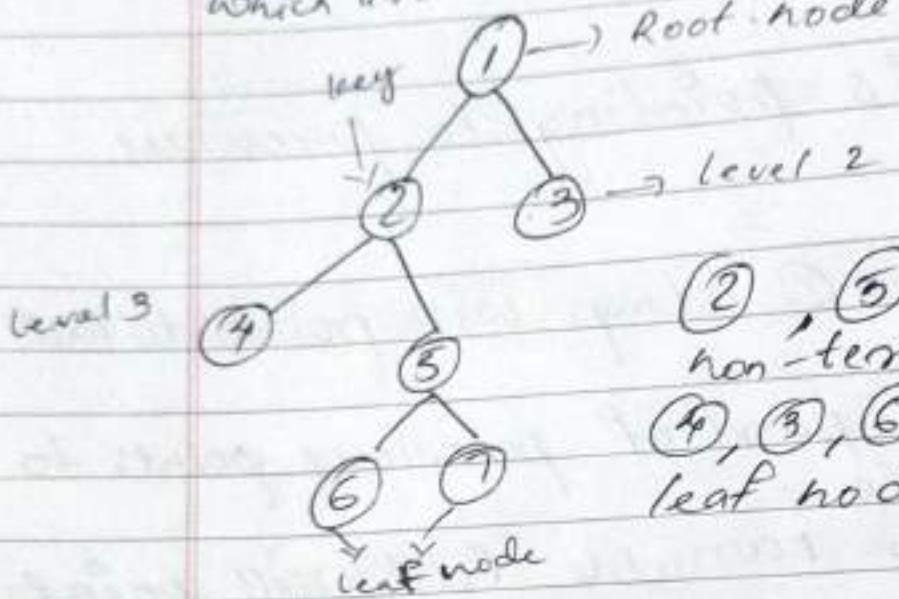
2 : Link part of previous points to first node.

3. pointer variable last will point to previous node.

~~14/12/10~~

Tree Data Structure

A tree is a non-linear hierarchical data structure that consists of nodes connected by "edges". which are either directed or undirected.



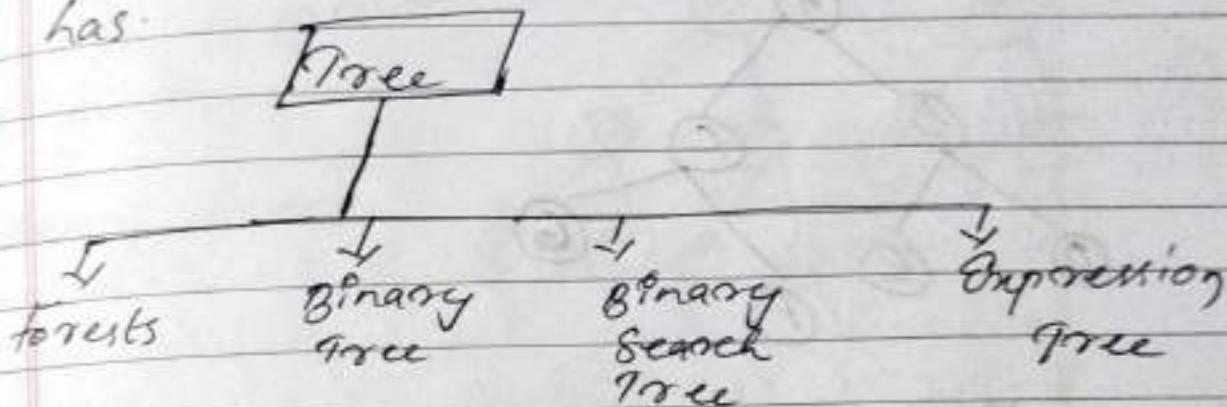
(2), (3) are non-terminal nodes
(4), (5), (6), (7) are leaf nodes

Node

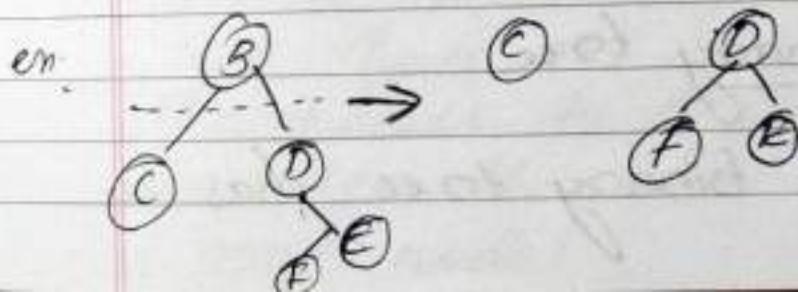
- i Root node: - Top most node in the hierarchy.
- ii Leaf node: It is bottom most node in a tree hierarchy.
- iii Parent node: Any node except the root

node that has a child node and an edge upward towards the parent.

- b) key: It represents the value of a node.
- b) path: The path is a sequence of consecutive edges.
- b) Degree: Degree of a node indicates the number of children that a node has.



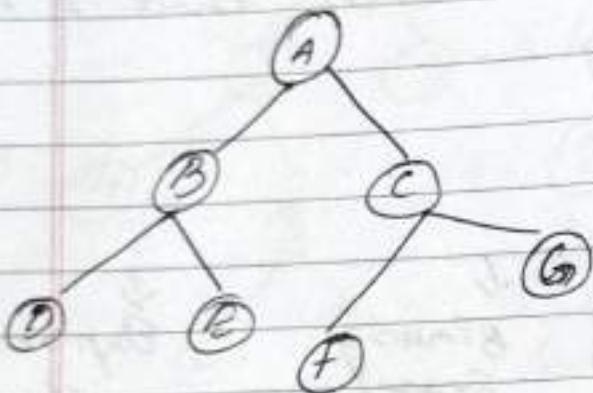
- 1) Forests: whenever we delete the root node from the tree and the edges joining the next level elements and the root, we obtain set of disjoint ~~sets~~ trees.



2 Binary Tree

A tree data structure in which each node has at most two child nodes is called a binary tree.

Applications: Expression evaluation and databases.



• Full binary tree

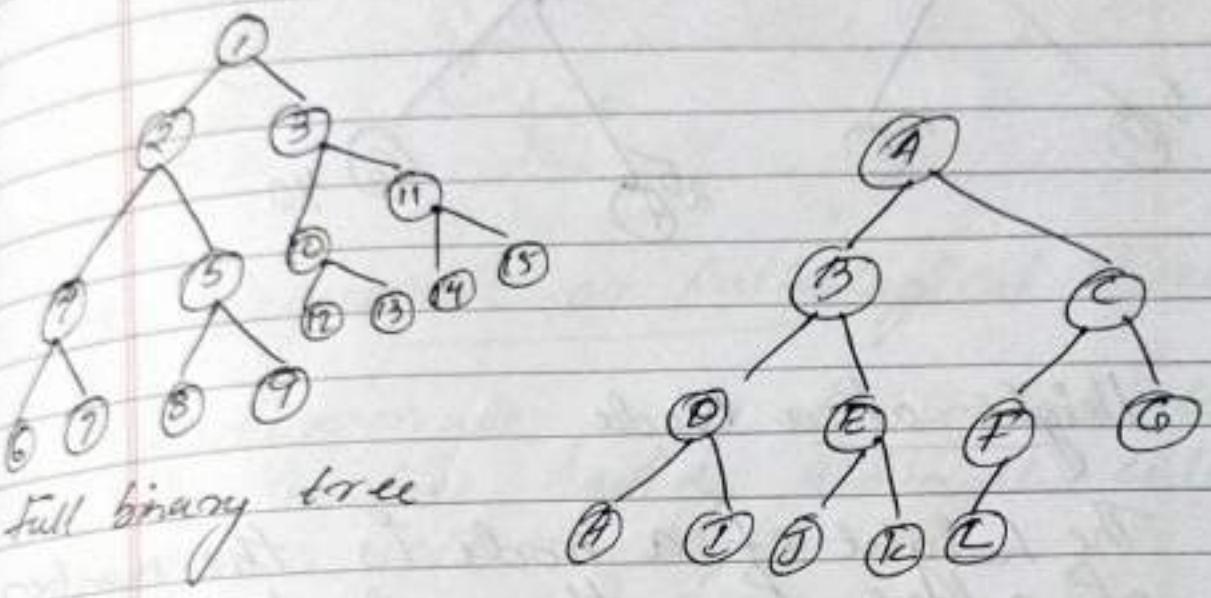
A binary tree in which each node has exactly zero or two children is called a full binary tree.

Complete binary tree

A complete binary tree has a

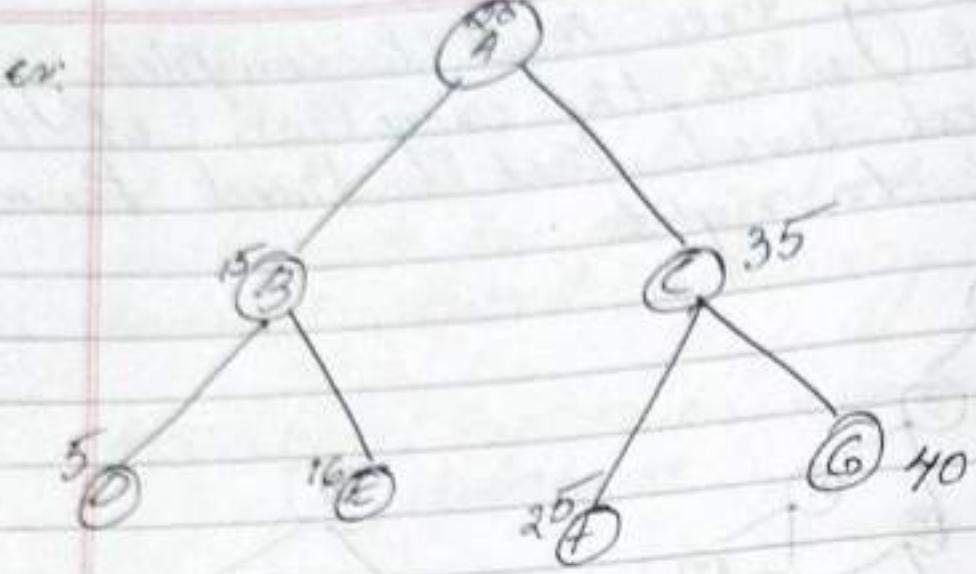
classmate
Date _____
Page _____

binary tree that is completely filled (with the exception of the lowest level) that is filled from left to right.



3. Binary Search Tree

The binary search tree is the binary tree that is ordered. The nodes to the left are less than the root node while the nodes to the right are greater than or equal to the root node.



Height of a Node

The height of a node is the number of edges from the node to the deepest leaf (i.e. the longest path from the node to a leaf node).

Depth of a Node

The depth of a node is the number of edges from the root to the node.

15/10/00

Binary Search tree

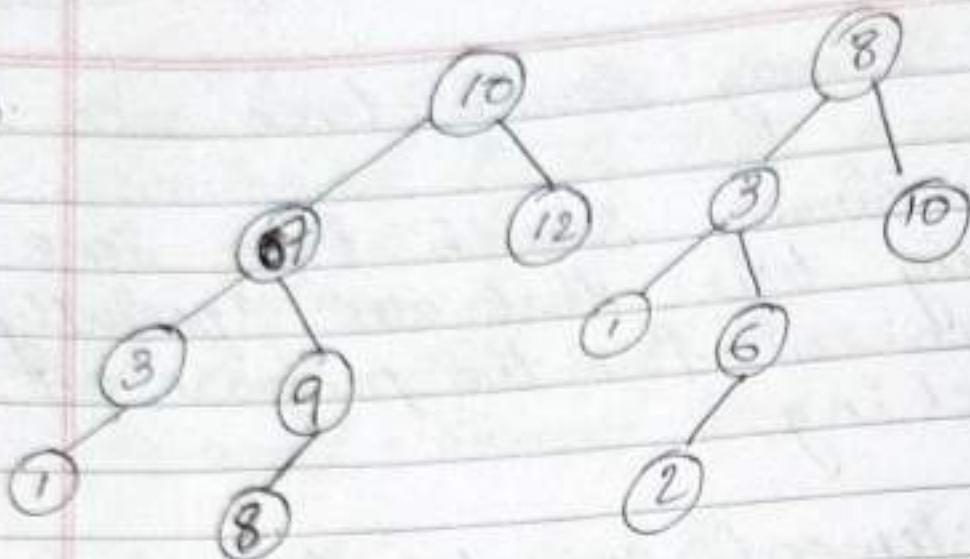
Binary Search trees are binary trees that are specially organized for the purpose of searching.

An element can be searched in average $O(\log N)$ time where N is no. of nodes.

A binary Search tree is a binary tree that may be empty and if it is not empty then it satisfies 2 properties

1. All the keys in the left subtree of root are less than the key in the root.
2. All the keys in the right subtree of root are greater than the key in the root.
3. Left and right subtrees are also binary search tree.

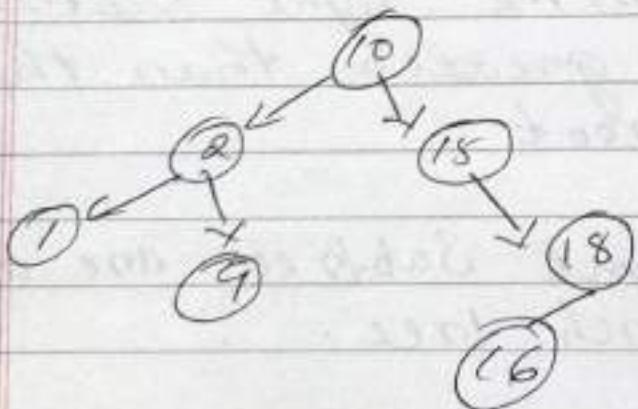
e9



X

✓

- * First one is right because the left subtree have small values than the root and right subtree has large values
- * Second one is wrong because in the subtree with root 3 the right subtree have less value like ②



- Different traversals are used in BST: (pre-order, In-order, post-order)

Searching (Non-recursion)

The function passed the address of the root of the tree and the key to be searched; It returns address of the node having the desired key or NULL, if such a node is not found.

```

Code struct node *Search (struct node *ptr,
                      int key)
{
    while (ptr != NULL)
        if (key < ptr->info)
            ptr = ptr->lchild;
        else if (key > ptr->info)
            ptr = ptr->rchild;
        else
            return ptr;
    }
    return NULL;
}

```

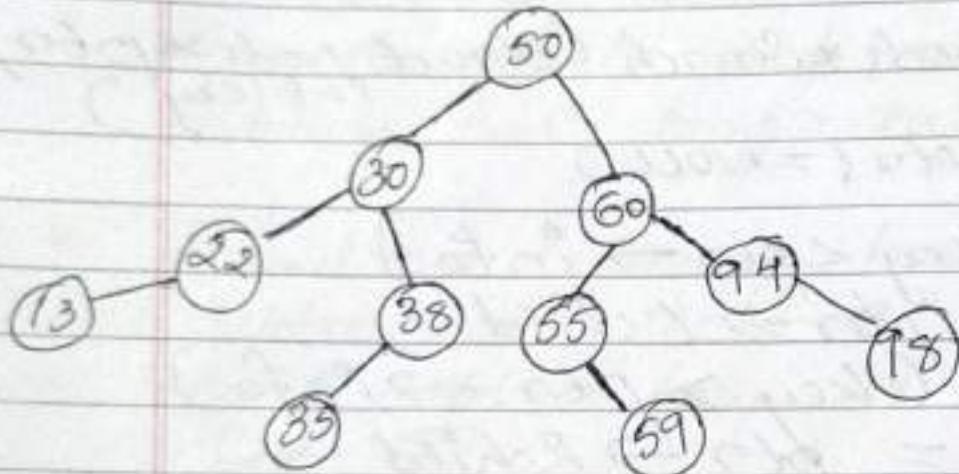
Operations On BST

- i Insertion
- ii Deletion

① Insertion

Creating a bst from keys

50, 30, 60, 38, 35, 55, 22, 59,
94, 13, 98



- * To insert in empty tree
root is made to point to
the new node. As we move
down the tree, we keep track
of the parent of the node

because this is required for insertion of new node

~~Code~~ struct node * insert(struct node * root,
int key)

```
{ struct node * temp, * par, * ptr;  
ptr = root;  
par = NULL;  
while (par != NULL)
```

```
    { par = ptr;  
if (key < ptr->info)  
    ptr = ptr->lchild;  
else if (key > ptr->info)  
    ptr = ptr->rchild;  
else
```

```
    { printf("Duplicate key");  
    return root;  
    }
```

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

temp->info = key;

```
tmp → Lchild = NULL;  
tmp → Rchild = NULL;  
if (par == NULL)  
    root = tmp;  
else if (key < par→info)  
    par→Lchild = tmp;
```

/* par is a pointer to the
parent node */

else

```
par → Rchild = tmp;  
return root;
```

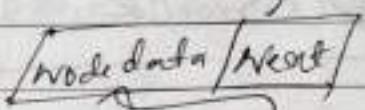
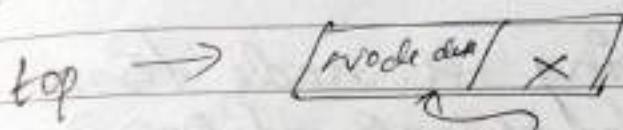
Linked List Implementation of Stack

- Instead of using array, we can also use linked list to implement stack.
- Linked list allocates the memory dynamically.
- Operations in linked stack

push()

pop()

peek()



- The top most node in the stack always contains null in its address field.

Adding a node to Linked Stack (push)

- Create a node first and allocate memory to it.

- 2 If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
- 3 If there are some nodes in the list already then we have to add the new element in the beginning of the list.

Deleting a node from Linked Stack (Pop)

Steps

- 1 Check for the underflow condition
- 2 Adjust the head pointer accordingly
3 elements are popped only from one end; therefore the value stored in head pointer must be deleted and the node must be freed.

Bit string

$U = \{1, 2, 3, 4, 5\} \rightarrow$ Universal set

Any Subset of universal Set in
0 & 1 form is called Bit
String.

e.g.: $A = \{1, 3, 5\}$

$A = \{10101\} \leftarrow$ bit string

$B = \{\}$

$= \{0, 0, 0, 0\} \leftarrow$ Bit string

$C = \{a\} = \{1, 1, 1, 1, 1\} \leftarrow$ Bit string

AND, (Intersection)

OR - Union

$A \cap B$

0	0	0
0	1	0
1	0	0
1	1	1

00	-0
01	-1
10	-1
11	-1

$$A \cup B = \{0, 1, -1\}$$

$$A = \{2, 4, 5\}$$

$$B = \{1, 3, 5\}$$

$A \cap B = A \text{ intersection } B = A \cap B$
 all common elements

$$C = \{5\}$$

In bit string \Rightarrow AND operation
 \Rightarrow Intersection (\cap)

OR \Rightarrow OR operation \Rightarrow Union (\cup)
 $\cap \Leftrightarrow \text{AND} ; \cup \Leftrightarrow \text{OR}$

$$\text{Ex. } A = \{0, 1, 0, 1, 1\}$$

$$B = \{1, 0, 1, 0, 1\}$$

$$C = \{0, 0, 0, 0, 0, 1\} = \{5\}$$

$$A \cap B = \{5\}$$

②

$$A \cup B$$

$$A = \{0, 1, 0, 1, 1\}$$

$$B = \{1, 0, 1, 0, 1\}$$

$$C = \{1, 1, 1, 1, 1\}$$

$$= \{1, 2, 3, 4, 5\}$$

$$A \cup B = \{1, 2, 3, 4, 5\} = \{1, 1, 1, 1, 1\}$$

3.

$$\bar{A}$$

$$A = \{0, 1, 0, 1, 1\}$$

$$\bar{A} = \{1, 3\}$$

$$= \{1, 0, 1, 0, 0\}$$

Difference

$$(A - B) = \{a / a \in A \text{ and } a \notin B\}$$

29/12/20

Hashing

Suppose we have to store record of 15 Students and each record consists of roll no. and name of the student. RNo are in range 0 to 14 and they are taken as key.

0	Anil
1	Ane
2	Ajay
3	Irat
4	Appu

Here we search any record with key K and

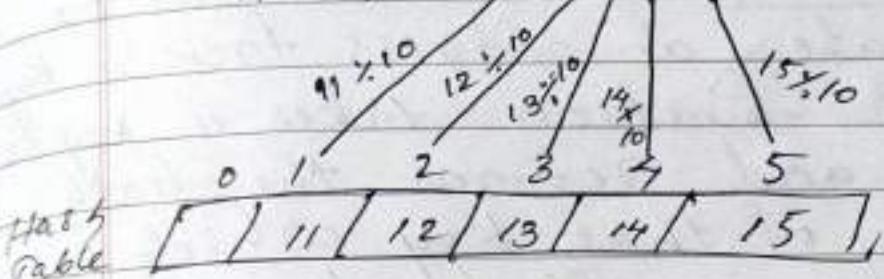
- * Hashing is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements.
- * The efficiency of mapping depends on the efficiency of the hash function used.

Let a hash function $H(x)$ maps the value x at the index $x \% 10$ in an array. For example if the list of values is $[11, 12, 13, 14, 15]$ it will be stored at positions $\{1, 2, 3, 4, 5\}$ in the array or hash table.

Hashing Data Structure

$$\text{List} = [11, 12, 13, 14, 15]$$

$$H(x) = \{x \% 10\}$$



In hashing we search any record with key $1/k$ and we access any record without comparison. This method is called direct addressing.

e.g.: There is 500 employee but their ID is 5 digit no: 00000 to 99999. These possible key is much more than number of employee, only

Note
Page

500 location of array must be needed. So we convert key into various address.

⇒ The process of converting key into address (index of array) is called hashing or key to address transformation and it is done through hash function.

Qn A hash function is used to generate an address from a key. A hash function takes a key as input and returns the hash value of that key which is used as the address for storing the key.

Key may be any type like integer, strings etc. But hash value will always be an integer.

Key → Hash fn → Address

Collision

The Situation collision occurs when the hash f^h generates same address for different keys.

⇒ Algorithm for hashing is focused mainly on :-

1. Selecting a hash f^h that converts keys to addresses
2. Resolving the Collision

Hash Function

A good hash f^h have two criteria

1. It should be easy to compute
2. generate address with minimum collision.

Method

Truncation

Easiest method for computing address from a key. Here we take a part of key as address i.e., take rightmost or leftmost digit.

8 2 3 9 4 5 6

8 8 5 6 9 3 7 1

Take address as 56, 71

Easy to compute but collision chance is high because last 2 digit can be same in many key.

MidSquare Method

Ex key = 1337

Square of key = 1787569

Address 875

In midSquare ; the key is

Squared and some digit or bits from the middle of the square are taken as address.

3 Folding method

ex. 738239456527

- * Break the key into parts

i.e; 738 239 456 527

- * After this, parts are shifted and added.

$$\begin{array}{r} 738 \\ 239 \\ 456 \\ 527 \\ \hline 1960 \end{array}$$

- * Now ignore the final carry i.e. the hash address for the key is 960. This technique is shift folding.

Another folding is boundary folding, here even parts are reversed before addition.

$$\begin{array}{r}
 738 \\
 932 \text{ (Rev)} \\
 456 \\
 725 \text{ (Rev)} \\
 \hline
 2851
 \end{array}$$

Ignore carry '2', address is 851

4. Division method / modulo-division

Here key is divided by the table 8120 and the remainder is taken as the address for hash table.

$$\text{e.g.; } H(k) = k \bmod m$$

$$\text{eg: } 82394561 \% 97 = 45$$

$$87139465 \% 97 = 0$$

So address of above keys are 45 & 0.

Collision Resolution

- * There are two collision resolution techniques : -
 1. Open addressing (closed hashing)
 2. Separate chaining (open hashing)
 3. Bucket hashing
- (b) In the process of searching given key is compared with many keys and each key comparison is known as probe.
- (c) The efficiency of a collision resolution technique is defined in the terms of the number of probes required to find a record with a given key.

2

Open addressing

In open addressing the key which caused the collision

is placed inside the hash-table itself but at location other than its hash address.

Initially a key value is mapped to a particular address in the hash table. If that address is already occupied then we will try to insert the key at some other empty location inside the table.

3 Methods in Open addressing

1. Linear probing
2. Quadratic probing
3. Double hashing.

1. Linear Probing

If address given by hash f^n is 'a' and it is not empty, then we will try to insert the key at next position or location.

ie; address at 1. If at 1 is occupied then try to insert next address at 2

ex 29, 46, 18, 36, 43, 21, 24, 34

Total table size = 11

$$h(\text{key}) = \text{key} \% 11$$

$$h(29) = 29 \% 11 = 7$$

$$h(46) = 46 \% 11 = 2$$

$$h(18) = 18 \% 11 = 7$$

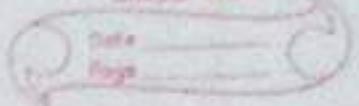
$$h(36) = 36 \% 11 = 3$$

$$h(43) = 43 \% 11 = 10$$

$$h(21) = 21 \% 11 = 10$$

0	1	2	3	4	5	6	7	8	9	10

insert 29, 46 → 29, 18, 36



formula

$$H(k, i) = (h(k) + i) \bmod T\text{size}$$

2

Quadratic Probing

In linear probing, the colliding keys are stored near the initial collision points, resulting in the formation of clusters.

In quadratic probing, this problem is solved by storing the colliding keys away from the initial collision point.

formula

$$H(k, i) = (h(k) + i^2) \bmod T\text{size}$$

i.e; $h(k)$; $h(k)+1$, $h(k)+4$,

$h(k)+9$ all mod $T\text{size}$

eg: 46, 28, 35, 57, 39, 50

$$h(k) = h(46) = 46 \div 11 = 2$$

$$h(k) = h(28) = 28 \div 11 = 6$$

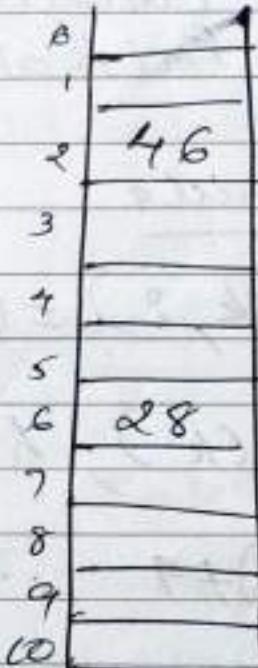
$$h(k) = h(35) = 35 \div 11 = 2$$

$$h(k) = h(57) = 57 \div 11 = 2$$

$$h(k) = h(39) = 39 \div 11 = 6$$

$$h(k) = h(50) = 50 \div 11 = 6$$

i Present 46, 28



6	
1	
2	46
3	35
4	.
5	
6	28

0	57
1	
2	
3	
4	
5	
6	

iii. Insert 57

ii. Insert 35

Workflow

- i. In (i) case, 46 and 28 are inserted without any collision.
- ii. Now 35 is inserted whose hash address is ②. To which ② is not empty so next location $(2+1) \mod 11 = 3$ is tried and it is empty, 35 is inserted in location 3.

- iii To insert 57 $\frac{1}{11}$ location
 \rightarrow 2nd tried ; not empty
 \rightarrow 2nd location $(2+1) \times 11 = 3$ not empty
 \rightarrow 3rd location $(2+4) \times 11 = 6$, not empty.
 \rightarrow 4th location $(2+9) \times 11 = 0$ empty
So 57 inserted at location 0

↳ Quadratic probing have another type of clustering problem called Secondary clustering ie; same location will be probed in each case are same.

3. Double Hashing

In double hashing, increment factor is not constant as in linear or quadratic probing. But depends on key.

The increment factor is another hash function and hence the name double hashing obtained.

Formula

$$H(k, i) = (h(k) + i h'(k)) \bmod T \text{ size}$$

- Value of i varies from 0 to $T \text{ size} - 1$
- h' is hash function. Search of empty location will be in the sequence: $- h(k), h(k) + h'(k), h(k) + 2h'(k), h(k) + 3h'(k) \dots$ all mod $T \text{ size}$.

Ex. $\{46, 28, 21, 35, 57, 39, 19, 50\} \quad \{8 \text{ elements}\}$

$$h(k) = \text{key \% 11}$$

$$h'(k) = 7 - (\text{key \% 7})$$

$$h(46) = 46 \% 11 = 2$$

$$h(28) = 28 \% 11 = 6$$

$$h(35) = 35 \% 11 = 2$$

$$h(21) = 21 \% 11 = 10$$

$$h(57) = 57 \% 11 = 2$$

$$h(39) = 39 \% 11 = 6$$

$$h(19) = 19 \times 11 = 8$$

$$h(50) = 50 \times 11 = 6$$

i Insert 46, 28, 21

0	
1	
2	46
3	
4	
5	
6	28
7	
8	
9	
10	21

0	
1	
2	46
3	
4	
5	
6	28
7	
8	
9	35
10	21

ii Insert 35

To insert 57, collision is at
2 occurs.

∴ Next probe

$$h(k) + h'(k)$$
$$= 2 + (7 - 57 \div 7) \times 11$$

$$= 2 + (7 - 1) \times 11$$

$$= \underline{8} \times 11$$

empty location is 8

∴ 57 is at location 8

in Insert, 39

Collision occurs at 6

Next probe: $h(k) + h'(k)$

$$\Rightarrow 6 + 1(7 - 39 \div 7) \times 11$$

$$\Rightarrow 6 + (7 - 4)$$

$$\Rightarrow 6 + 3 = 9 \times 11 = 9$$

But 9 is not empty.

Next probe:

$$h(k) + 2(h'(k))$$

$$\begin{aligned}
 & 6 + 2(7 - 39 \div 7) \times 11 \\
 & 6 + 2(3) \times 11 \\
 & 6 + 6 \times 11 \\
 & = 12 \times 11 \\
 & = \underline{\underline{1}} , \text{ empty location}
 \end{aligned}$$

0	
1	39
2	46
3	
4	
5	
6	28
7	
8	57
9	35
10	21

- * Double hashing is complex because 2 types calculations of hash fⁿ:
 - * Load factor denoted by λ and is calculated as
- $$\lambda = \frac{n}{m}$$
- where,
 n = no. of records
 m = no. of positions
- * In open addressing load factor is ≤ 1 .

Disadvantage of O.A

1. key values far from their home addresses, which increases probes.
2. Overflow occurs.

Separate chaining

In this method, a linked list is maintained for elements that have some hash address. All elements having some hash address 'i' and will be stored in separate LL and starting address of that LL stored in index i of hash table.

These LL are referred to as chains and hence the method is named as Separate chaining.

Suppose hash table of size 7

$$H(\text{key}) = \text{key} \% 7$$

$$H(1895) = 1895 \% 7 = 2$$

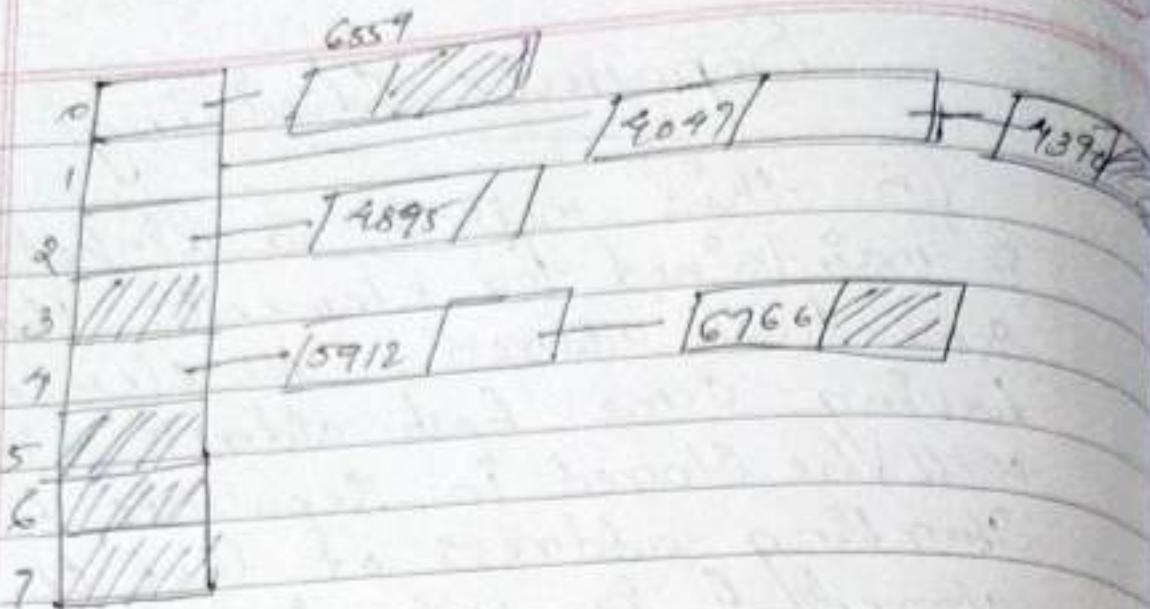
$$H(6559) = 6559 \% 7 = 0$$

$$H(5912) = 5912 \% 7 = 4$$

$$H(4047) = 4047 \% 7 = 1$$

$$H(6766) = 6766 \% 7 = 4$$

$$H(4390) = 4390 \% 7 = 1$$



- Here Separate chaining. There is no problem of overflow because linked list are dynamically allocated so there is no limitation on the number of records that can be inserted.

Disadvantage

- It needs extra space for pointers.
- If n records & m table size
 \rightarrow extra space $n + m$

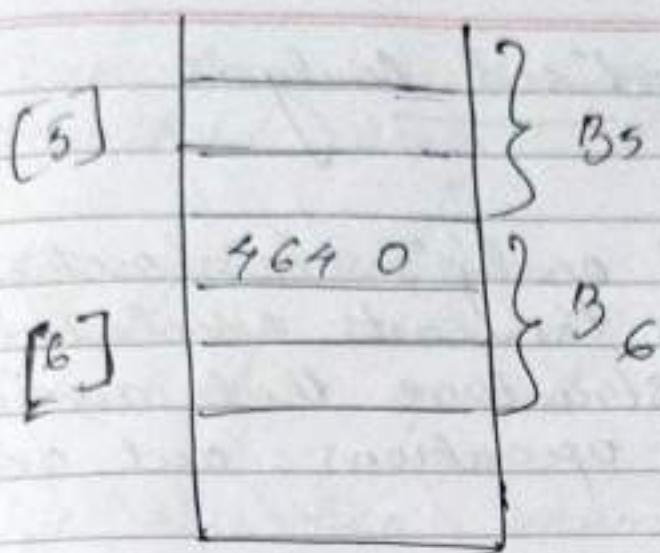
In separate chaining load-factor
 $(\lambda) > 1$

Bucket Hashing

- This technique postpones collision but does not resolve them completely. Here Hash table is made up of buckets, where each bucket hold numbers of records.
- There is one bucket at each hash address this means we can store multiple records at given hash address.
- Collision will occur only after a bucket is full
 - e.g. A bucket stores 3 records at same hash address - Collision occurs when the fourth record with the same hash address arrives.

$$\begin{aligned}
 H(4895) &= 4895 \div 7 = 2 \\
 H(6559) &= 6559 \div 7 = 0 \\
 H(5912) &= 5912 \div 7 = 4 \\
 H(4047) &= 4047 \div 7 = 1 \\
 H(6766) &= 6766 \div 7 = 4 \\
 H(4390) &= 4390 \div 7 = 1 \\
 H(4640) &= 4640 \div 7 = 6 \\
 H(4900) &= 4900 \div 7 = 0 \\
 H(4411) &= 4411 \div 7 = 1
 \end{aligned}$$

[0]	6559 4900	$\left. \begin{array}{c} 6559 \\ 4900 \end{array} \right\} B_0$
[1]	4047 4390	$\left. \begin{array}{c} 4047 \\ 4390 \end{array} \right\} B_1$
[2]	4411 4895 .	$\left. \begin{array}{c} 4411 \\ 4895 \\ . \end{array} \right\} B_2$
[3]	.	$\left. \begin{array}{c} . \\ . \end{array} \right\} B_3$
[4]	5912 6766 .	$\left. \begin{array}{c} 5912 \\ 6766 \\ . \end{array} \right\} B_4$

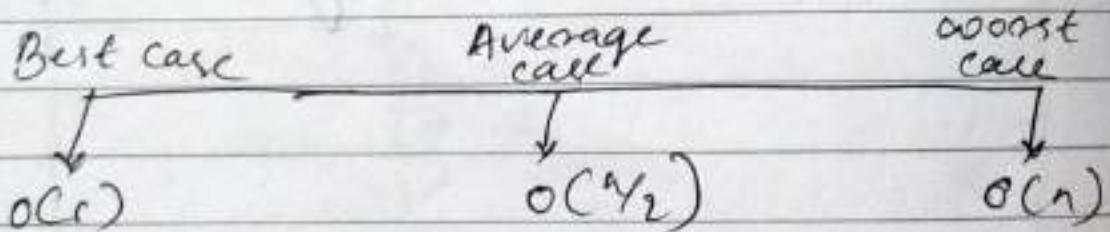


Disadvantage

1. Wastage of Space since many buckets will not be occupied or will be partially occupied.
2. Does not prevent collision but only defers them; when collision occurs, resolved using techniques in open addressing.

Amortized Analysis

- * Amortized analysis is a method of analyzing the costs associated with a data structure that averages the worst operations out over time.
- * In amortized analysis we average the time required to perform a sequence of data structure operations over all the operations performed.
- * Amortized analysis is not equal to average case analysis because amortized analysis guarantees the average performance of each operation in worst case.



Three common technique used in amortized analysis are :-

1. Aggregate analysis
2. Accounting method
3. Potential method

We use two examples to examine these three method :-

1. Stack with additional operation multipop ; which pop several object at once .
2. Binary Counter

Aggregate analysis

In aggregate analysis , there are two steps .

First , we must show that a sequence of n operations takes $T(n)$ time in the worst case .

Then , we show that each operation

takes $\frac{T(n)}{n}$ time , on average

- Therefore , in aggregate analysis ; each operation has the same cost .
- A common example of aggregate analysis is a modified stack .

eg: stack operations

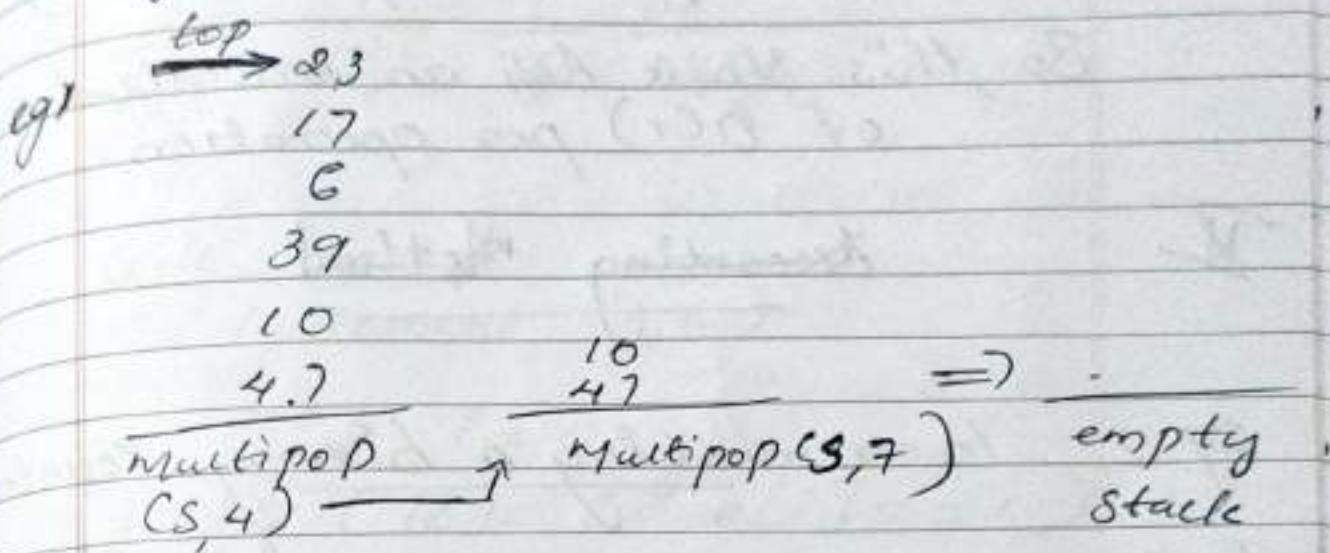
Two fundamental stack operations each of which takes $O(1)$ time.

i) $\text{push } (s, a) \rightarrow$ Pushes object a on stack s

ii) $\text{pop } (s) \rightarrow$ pops the top of stack s and returns the popped object . calling pop on empty stack generates an error .

in push & in pop total running time for n operation $\Theta(n)$

iii) Operation $\text{multipop}(s, k) \rightarrow$ which removes k top objects of stack s , popping the entire stack if the stack contains fewer than k objects.

Code

$\text{multipop}(s, k)$
 while stack not empty and $k > 0$:
 $k = k - 1$.

$\text{stack.pop}()$

↳ multipop can run for at most n times, where n is the size of the stack. So, the worst-case runtime for multipop is $O(n)$

- * for any value of n, any sequence of multiple pop, push and push takes $O(n)$ time. So, using aggregate analysis:-

$$\frac{T(n)}{n} = \frac{O(n)}{n} = O(1)$$

So, this stack has an amortized cost of $O(1)$ per operation.

X Accounting Method

eg 2 Incrementing a binary counter
Index (i)

$k=4$

Row No	3	2	1	$0 \leftarrow i$
→1	0	0	0	0
2	0	0	0	1
3	0	0	1	0
→4	0	0	1	1
5	0	1	0	0
6	0	1	0	1
7	0	1	1	0
8	0	1	1	1
9	1	0	0	0
10	1	0	0	1

11	1	0	1	0
12	1	0	1	1
13	1	1	0	0
14	1	1	0	1
15	1	1	1	0
16	1	1	1	1
	$\frac{1}{N/2}$	$\frac{1}{N/2^2}$	$\frac{1}{N/2^1}$	$\frac{1}{N/2^0}$

Codeincrement (A, k)

{

int $i = 0$ while ($i < k \text{ } \& \& A[i] == 1$)

{

 $A[i] = 0$ $i = i + 1$ if ($i < k$) // ok{ $A[i] = 1$ // 0 changes
to 1 in the
next array }{ push all 0 to the
1st array. }

Step 2

Pass 2nd array.

No. of bit = $k = 4$

Sequence 1, operation = 16

$$\text{ie; } n = 16 = 2^k$$
$$\Rightarrow 16 = \underline{\underline{2^4}}$$

$$n = 2^k$$
$$k = \underline{\underline{\log n}}$$

- * In a worst case situation, a single execution of increment takes $O(k)$ times, where k is size of array [@ 16th sequence 4 array changes]

- Total = $\frac{n}{2^0} + \frac{n}{2^1} + \frac{n}{2^2} + \dots + \frac{n}{2^{k-1}}$

$$\frac{n}{2^0} + \dots + \frac{n}{2^{k-1}}$$

$$n = 16$$

$$k=1$$

$$\therefore n \geq \frac{1}{2^k}$$

$$= 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots + \frac{1}{2^{k-1}}$$

= geometric progression

Sum to infinite terms

$$n \cdot \sum_{i=0}^{k-1} \frac{1}{2^i} = 1 + \frac{1}{2} + \frac{1}{4} + \dots$$

$\therefore \infty$

$$= \frac{a}{1-r}$$

$$= \frac{1}{1-\frac{1}{2}} = \frac{1}{\frac{1}{2}} = 2$$

$a = 1^{\text{st}}$ term ie: 1

$r = \text{common ratio}$ ie: $\frac{1}{2}$

$$= \underline{\underline{2}}$$

CLASSmate
Date _____
Page _____

$$\text{Aggregate} = \frac{\Theta(n^2)}{n} = O(n)$$

Constant

41.

Accounting Method

Amortized analysis is a method of analysing the costs associated with a data structure that averages the worst operations over time.

- In this method, borrows ideas & terms from accounting.

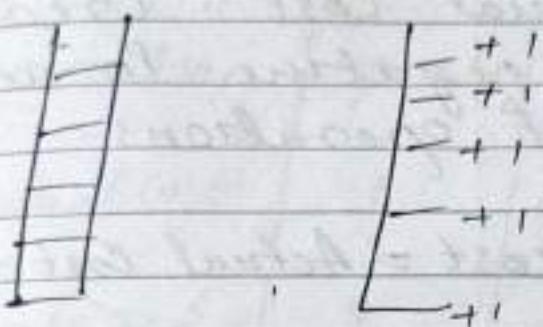
Save credit	Normal operation
Use saved credit	Expensive
Save credit	Normal operation

- Each operation is assigned a charge called the amortized

cost.

- Some operations can be charged more or less than they actually cost.

Actual cost = No. of steps taken /
time required to run



push (5 dep)

So actual cost of push = 5

Amortized cost of push = 10

[we are planned to give extra 1 to each step]

- If an operation's amortized cost exceeds its actual cost, we assign the difference, called a credit, to specific objects

In the data structure

$$\text{Credit} = \text{Amortized cost} - \text{Actual cost}$$
$$= 10 - 5 = 5$$

→ credit can be used later to help for other operation where amortized cost is less than their actual cost. Credit can never be negative in any sequence of operations.

$$\left\{ \begin{array}{l} \text{Amortized cost} = \text{Actual Cost} + \text{Credit} \end{array} \right.$$

eg: operation Some Cost
Amortized cost = 10
Actual cost = 5

eg: Stack

operation	Push	pop	Multipop
Actual cost	1	1	$\min(1, k)$
			(pop operate $\theta n/k$ times, $n=5$ = size of Stack; $k=No$)

of pop

Step 1 Actual cost

$$\text{push} = 1$$

$$\text{pop} = 1$$

$$\text{multipop} = \min(n, k)$$

Step 2	operation	push	pop	multipop
Actual cost		1	1	$\min(n, k)$
Amortized cost		2	0	0

add extra credit

↳ credit of push

→ Amortized cost -
Actual cost

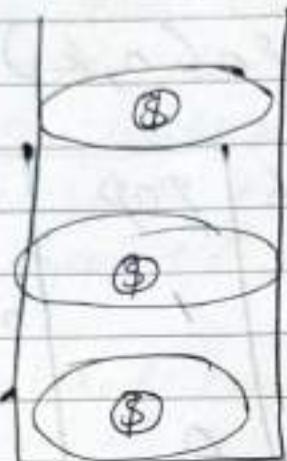
$$\Rightarrow \underline{\underline{2 - 1 = 1}}$$

- * Pop & multipop won't be done, if there is no push so amortized cost of pop & multipop is 0

Final Step: Show that it is possible to pay for any sequence of operations using the amortized costs.

e.g.: Push a plate require 2 coins

T



1 \$ Actual cost

1 \$ credit \rightarrow put credit at top of plate.

i.e; we pay \$1 for the actual cost of the operation & we are left with \$1 of credit kept left \$ on the plate.

T

Then apply pop operation
 \hookrightarrow need \$1 to pop the plate off
 \rightarrow Amortized cost of popping (\$0)

Actual cost of popping = \$ 1

* The \$ 1 on top of plate will act as the money needed to pop the plate off.

* Every plate has exactly \$ 1 on top of it which can be used to pop it off the stack.

So Actual cost of pop at st
time = \$ 1.

III Multipop \rightarrow Pop operation as subroutine

* Calling multipop on the stacks remove \$ 1 on top of every plate, credit is never negative and max credit used is n

* Cost of worst case operation is $O(n)$

* The amortized cost of an operation is average is $O(n) = O(1)$

re: * no plate
credit = 0

* 1. plate
credit = 1

* 2. plate
credit = 2

* n plate
credit = n

eg: 2 Incrementing a binary Counter

- ✓ Actual cost of i^{th} operation = c_i
- ✓ Amortized cost of i^{th} operation
 $= \hat{c}_i$

then $\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i$

$$\text{Credit} = \sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i$$

For a sequence of n operations

- In Increment operation, we charge amortized cost of \$'2 to set a bit to one.
- when a bit is set, we use 1 dollar (out of 2 dollars charged) to pay for actual setting of the bit and we place the other dollar on the bit as credit. At any point in time, every '1' in the counter has a dollar of count on it. So we need not charge anything to reset a bit to 0, we just pay \$1 for reset.

In n increment operations
 $\Rightarrow O(n)$

III

Potential Method

- This PEs Similar to accounting method.
- Instead of using cost & credit, method use "potential energy".
- Potential energy is associated with the data structure as a whole, not with individual operation (push, pop, multi-pop)
⇒ So potential energy for a stack as a whole.
- PE starts with an initial data structure D_0 .
- Then n operations are performed turning the initial data structures into $D_1, D_2, D_3 \dots D_n$
 $\uparrow \quad \uparrow \quad \uparrow$
 n^{th} stage 1^{st} stage 2^{nd} stage

PE changes in each step or stage

C_i is the cost associated with the i^{th} operation and D_i is the data structure resulting from the i^{th} operation.

- i) ϕ is the potential function
 \hookrightarrow which maps the data structure D to a number $\phi(D)$, the potential associated with the data structure
- ii) $D_0 \rightarrow$ initial D_0
- iii) 1^{st} operation (stack) then D_1 -potential
- iv) 2^{nd} operation (stack),
 \quad (push, pop . . .)
 then D_2 - potential
- v) n operations
 $\quad D_n$ potential
- vi) cost associated with each step

Amortized cost of the i^{th} operation
 is defined by

$$C'_i = C_i + \underbrace{\phi(D_i) - \phi(D_{i-1})}_{\text{Cost}}$$

change in potential due to operation

// for 'i' operation cost

$$\sum_{i=1}^n c_i = \sum_{i=1}^n c_i + \sum_{i=1}^n \phi(D_i) - \phi(D_{i-1})$$

↓
Current
Stage

[Charge in Potential = $\phi(D_i) - \phi(D_{i-1})$]

- * If change in potential is +ve value then data structure is over charged.
- * If change in potential is -ve value then data structure is undercharged.

eg: Stack

↷ Push operation [size = 7]

$$\begin{aligned} \text{Potential change} &= \phi(D_7) - \phi(D_6) \\ &= (\text{size of stack} + 1) - \text{size of stack} \end{aligned}$$

$$(D_i) \text{ Current Size} = \text{Previous Size} + 1$$

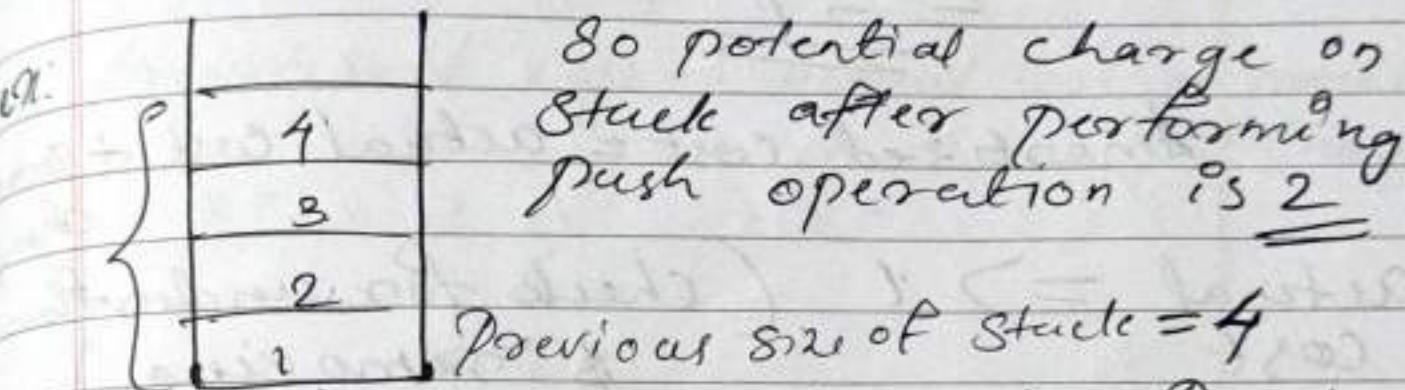
$$\phi(D_i) - \phi(D_{i-1}) = (n+1) - n$$

Potential change = 1

Amortized cost = actual cost + potential change

$$= C_p + 1$$

$$= 1 + 1 = \underline{\underline{2}}$$



(D_{i-1}) . size = 4 \rightarrow initial or previous

$$D_0 = \text{size} \pm 5 \rightarrow \text{current}$$

$$\Rightarrow n + 1 - n$$

$$\Rightarrow \cancel{0} 4 + 1 - 4$$

$$\Rightarrow 5 - 4$$

$$\underline{\underline{1}}$$

C_i = One step push
operation

II Pop operation

$$\begin{aligned}
 \text{Potential change} &= \phi(D_i) - \phi(D_{i-1}) \\
 &= (\text{size of } i-1 \text{ stack}) - \text{size of stack} \\
 &= (n-1) - n \\
 &= \underline{\underline{-1}}
 \end{aligned}$$

Amortized cost = actual cost + potential change
 actual $\Rightarrow 1$ (check for underflow
 cost & removing)

$$\begin{aligned}
 \text{Amortized} &= 1 + (-1) = 0 \\
 * \text{ Amortized cost of ordinary pop} \\
 \text{operation} &= \underline{\underline{0}}
 \end{aligned}$$

III Multipop

$$\text{Potential change} = \phi(D_i) - \phi(D_{i-1})$$

$$\Rightarrow (t-1) - t \\ = \underline{\underline{-t}}$$

where $t = \min(s, k)$

\downarrow \downarrow \downarrow
it becomes Stack size no. of popped
underflow

Amortized cost = Actual cost + potential
change

$$= t - t \\ = \underline{\underline{0}}$$

i.e., Push operation = 2
pop operation = 0
Multipop = 0 } all constant.

$$\text{Amortized Cost} = \hat{c} = O(1)$$

Amortized cost of n operations
 $\Rightarrow n * O(1) = \underline{\underline{O(n)}}$

Incrementing a binary counter

Example

2: we define the potential of the counter after the i^{th} increment operation to be b_i , the number of 1's in the counter after the i^{th} operation.

- The number of 1's in the counter after the i^{th} operation is therefore $b_i \leq b_{i-1} - t_i + 1$ and the potential difference is $\phi(D_i) - \phi(D_{i-1}) \leq (b_i - 1 - t_i + 1) - b_{i-1} = 1 - t_i$.

- The amortized cost is therefore

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

$$\leq (t_i + 1) + (1 - t_i)$$

$$= \underline{\underline{2}}$$

If counter starts at 0, then $\phi(D_0) = 0$. Since $\phi(D_i) \geq 0$ for all i , the total amortized cost of a sequence of n increment operations is an upper bound on the total actual cost, and so the worst-case cost of n increment operations is $O(n)$.

The potential method gives us an easy way to analyse the counter even when it does not start at 0.

There are initially b_0 1's, and after n INCREMENT operations there are b_n 1's, where $0 \leq b_0, b_n \leq K$. We can rewrite eqn 10 as

$$\sum_{i=1}^n c_i = \sum_{i=1}^n \tilde{c}_i - \phi(D_n) + \phi(D_0)$$

We have $\tilde{c}_i \leq 2$ for all i $1 \leq i \leq n$. Since $\phi(D_0) = b_0$ and $\phi(D_n) = b_n$, the total actual cost of n increment

operations is $\sum_{i=1}^n c_i \leq \sum_{i=1}^n 2 - b_{n+i}$
 $= 2n - b_n + b_0$

R_0	3	2	1	0	\emptyset	C_i	Amortized cost $C_i + \emptyset_i - \emptyset(C_{i-1})$
1	0	0	0	0	0	0	$0 + (0 - 0) = 0$
2	0	0	0	1	1	1	$1 + (1 - 0) = 2$
3	0	0	1	0	1	2	$2 + (1 - 1) = 2$
4	0	0	1	1	2	1	$1 + (2 - 1) = 2$
5	0	1	0	0	1	3	$3 + (1 - 2) = 2$
6	0	1	0	1	2	1	$1 + (2 - 1) = 2$
7	0	1	1	0	2	2	$2 + (2 - 2) = 2$
$C_i = \emptyset_i$	0	1	1	1	3	1	$1 + (3 - 2) = 2$
9	1	0	0	0	1	4	$4 + (1 - 3) = 2$
10	1	0	0	1	2	1	$1 + (2 - 1) = 2$
11	1	0	1	0	2	2	$2 + (2 - 2) = 2$
12	1	0	1	1	3	1	$1 + (3 - 2) = 2$
13	1	1	0	0	2	3	$3 + (2 - 3) = 2$
14	1	1	0	1	3	1	$1 + (3 - 2) = 2$
15	1	1	1	0	3	2	$2 + (3 - 3) = 2$
c_i	1	1	1	1	4	1	$1 + (4 - 3) = 2$

$b_0 \leq K$ [$K = \text{no. of bits}$]

$(K = O(n)) = \text{total actual cost } O(n)$

so increment change only 1 bit from 0 to 1
 and amortized cost remains $O(1)$

Disjoint Sets

Representation :- UNION, FIND algorithms.

A disjoint set data structure maintains a collection $S = \{S_1, S_2, \dots, S_k\}$ of disjoint dynamic sets.

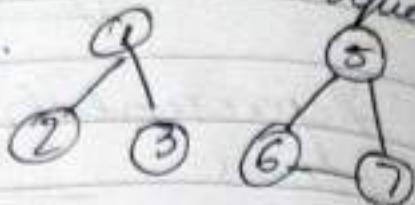
$$S_1 = \{3, 4, 5, 6, 7\}$$

$S_2 = \{1, 2\}$ \Rightarrow These two sets are disjoint bcz there is no common element in both set.

Disjoint set operation

1. **MAKE-SET(x)** \rightarrow Creates a new set whose only member (representative) is pointed to by x .
2. **UNION(x, y)** \rightarrow Unites dynamic sets that contain x and y i.e., $S_x \cup S_y$ into new set $S_x \cup S_y$.
3. $S_x = \{1, 2, 3\}$ $S_y = \{5, 6, 7\}$
new $S_z = \{1, 2, 3, 5, 6, 7\}$
4. **FIND-SET(x)** - returns a pointer to

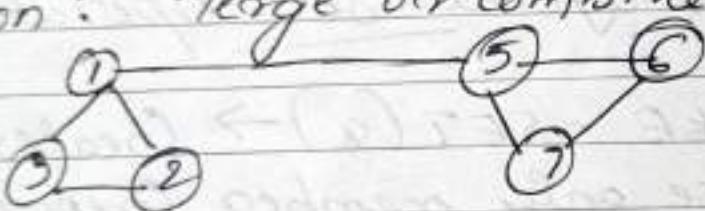
the representation of the (unique) set containing x .



UNION - FIND Algorithm

A union - find algorithm performs two operations

- a. **find**: Determine which set a particular element is in.
- b. **Union**: Merge or combine two sets



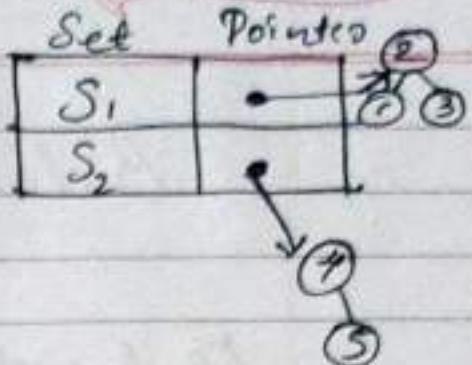
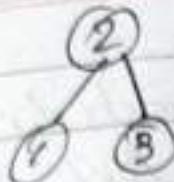
Ex

Connected Component $S_1 \cup S_2$

Disjoint Set Representation

- i. Linked list

Let $S_1 = \{1, 2, 3\}$ and $S_2 = \{4, 5\}$



i) Array

i	1	2	3	4	5	
$P[i]$	2	-1	2	-1	4	

Two or more set with nothing is common are called disjoint sets.

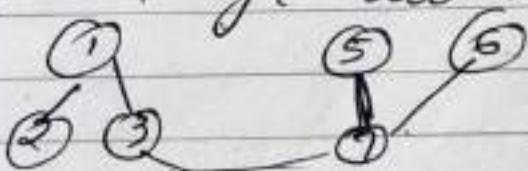
ii) Find Operation:

keep track of the set that an element belongs to i.e. it is easier to check, given 2 elements, whether they belong to the same subset.

belongs to $S_1 \leftarrow$ ie. find (3)

belongs to $S_2 \leftarrow$ ie. find (5)

Union : Merge two set



Union - Find implementation

1. Make set (i) \rightarrow from 0 to 9

$\{0\} \{1\} \{2\} \{3\} \{4\} \{5\} \{6\} \{7\} \{8\} \{9\}$

Union-set ($0, 1$) $\rightarrow \{0\} \{2, 1\} \{3\} \{4\} \{5\}$

Union-set ($4, 3$) $\rightarrow \{0\} \{2, 1\} \{4\} \{3\}$
 $\{5\} \{6\} \dots$

Union-set ($8, 4$) $\rightarrow \{0\} \{2, 1\} \{4, 3, 8\}$
 $\{5\} \{6\} \dots$

Union-set ($8, 9$) $\rightarrow \{0\} \{2, 1\} \{4, 3, 8, 9\}$
 $\{6\} \{5\} \{7\} \dots$

Q. Are 8 & 7 connected?

Find (8) = = Find (4) \Rightarrow Both are
one same set. \Rightarrow
then connected
else disconnected.