20/11/20

Module - 1

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

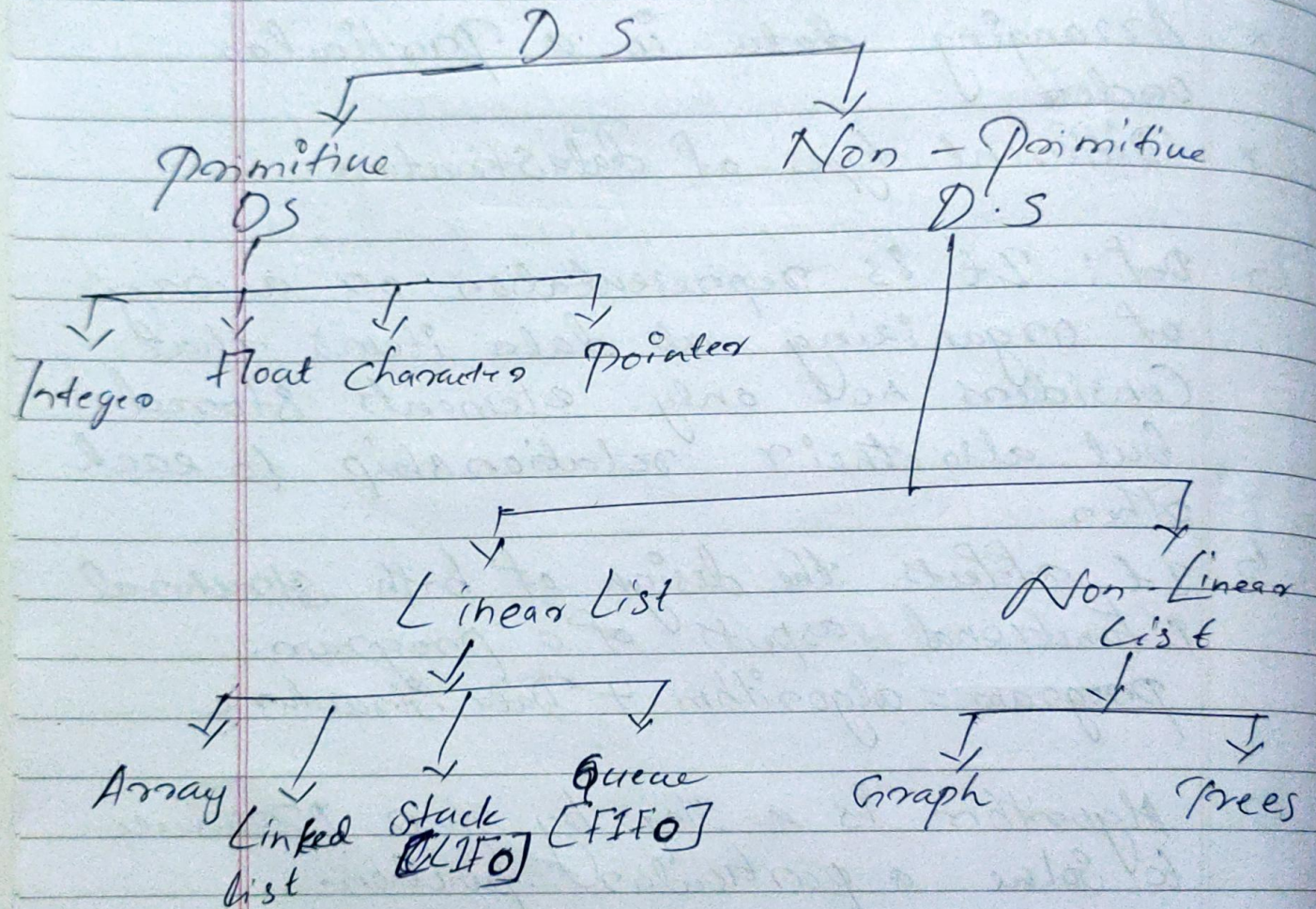
↳ Algorithm is a step by step procedure to solve a particular function.

↳ To develop a program of an algorithm we should select an appropriate data st

for that algorithm.

Classification of Data Structure

- Primitive D.S
- Non-primitive D.S



Primitive DS

- Basic Structures that directly operate upon the machine instructions.
- Built in or predefined data types and can be used directly by user to declare variables.

Non Primitive

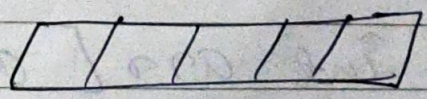
- More Sophisticated Data Structure.
- Derived from primitive D.S
- Emphasize on structuring of a group of homogeneous or heterogeneous (diff type) data items.

Arrays and Pointers

Arrays

- Array is a data structure containing a no. of data values (all of which are of same type.

Visualize Array

A = 

* Collection of Smaller block of m/y 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]

1. int arr[5]

ie; arr [| | | |]

* It means that compiler allocate 5 location.

Size = 5 x Size of (int)

ie; integer takes 4 bytes of location.

Variant of declaring.

eg int arr[a, b]

* The length of array cannot be negative.

Note: Specifying length

```
#define N 10 ← Macro [for fixing large limit]
int arr[N];
```

Accessing

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

Initializing

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

* for loop

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

Designated Initializers

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

Size calculation

* Sizeof (name of array[0])
  ↳ for finding the size of array

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

Multidimensional array

* Array of array

datatype name_of_arr [size1][size2]...[sizeN]

Size calculation: Multiplying the size of all dimensions.

ex: size of a[10][20]
= 10 * 10
= 20
= 20 * 4 (size)
= 80 byte

Two dimensional array

int a[4][5]
↳ 4 rows, 5 columns

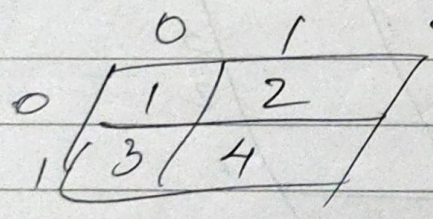
Size of a[4][5]
= 20 × 4 = 80 bytes
size of element

Initalize

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

Access 2D array

* Using row Index and Column index
ie: a[0][1] = 2



Looping

2 loops
1 - row
1 - column

Three dimensional Column

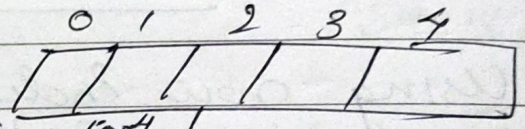
eg int a[1][2][3]

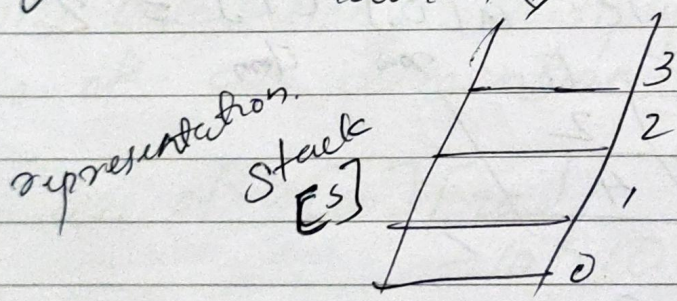
25/4 **Stack** LIFO

* Type of data structure.
* operation is ~~[illegible]~~ up and up

Implementation of Stack using array

Array a[5] =



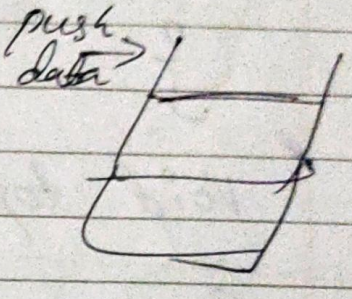


4 operations

```
Void main ()
 {
  push ()
  pop ()
  top ()
  display ()
 }
```

// using switch (ch)
case 1 : push
case 2 : pop
3 . . .

ex # define N 5 ← Macro
int stack [N]
int top = -1



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

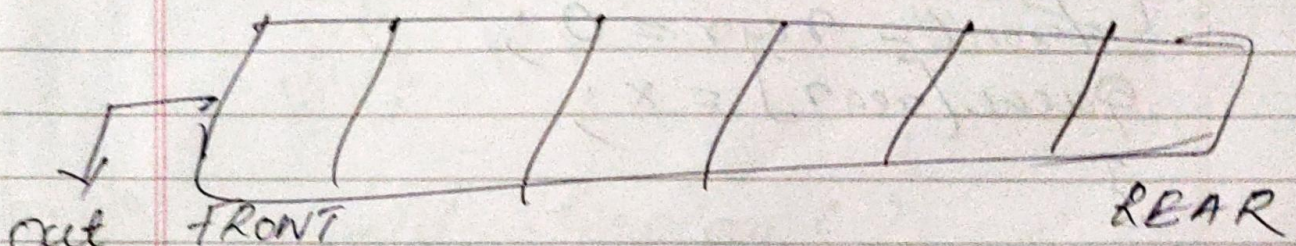
2.
```
Void pop ( )
{ int item;
  if (top == -1) // No data in the list
     {
       underflow
     }
else
{  item = stack[top]
     top --;
printf ("%d", poped item, item)
   }
```

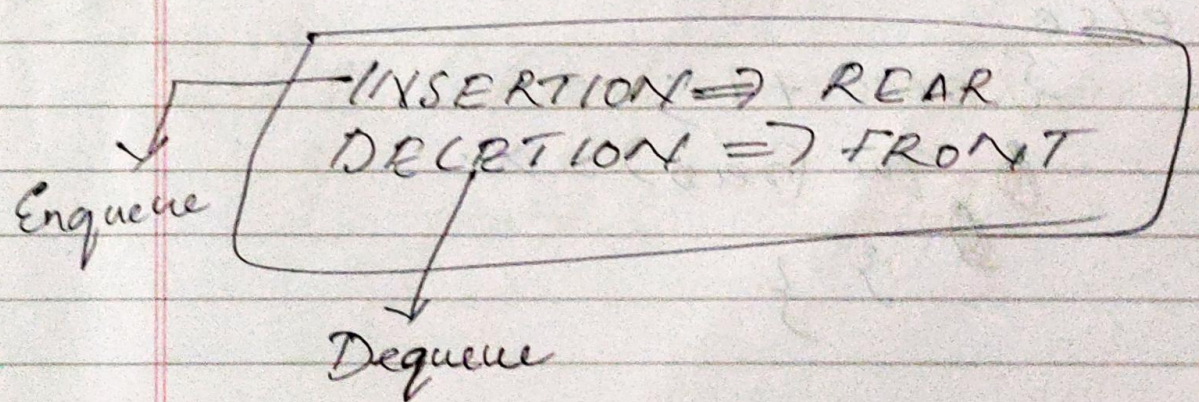
3.
```
Void top ( ) // getting the top most element.
{ if (top == -1 ) // empty )
    {
      stack is empty
    }
else { printf ("%d", stack[top]);
    }
```

4. Void display ()

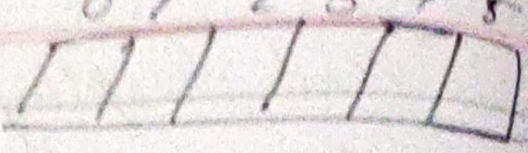
```
{ int i
for (i=top ; i≥0 ; i--)
{ printf ("%d", stack[i]);
}
}
```

// Queue [FIFO]





int a[6];



#define N 5
int queue[N]
int front = -1, rear = -1; // queue empty

Enqueue()

```
void enqueue (int x)
{
  if (rear == N-1)
  {
    print ("Queue is full");
  }
else if (front==-1 && rear==-1)
  {
   front= rear = 0;
   queue[rear] = x;
  }
else
     { rear ++;
       que(rear) = x.
       } }
```

Dequeue()

```
void dequeue()
{ if (front == -1 && rear == -1)
  { printf("underflow")
else if (front == rear)
  { front = rear = -1;
  }
else
 { printf("element deque" %d
                queue[front]);
   front++;
 } }
```

Display()

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

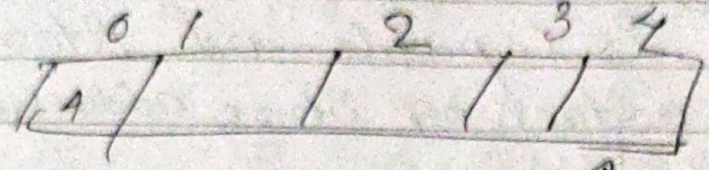
```
else
{ for(int i=front ; i<rear+1 ;
  i++)
   { printf("%d", queue[i]);
   }
```

top()

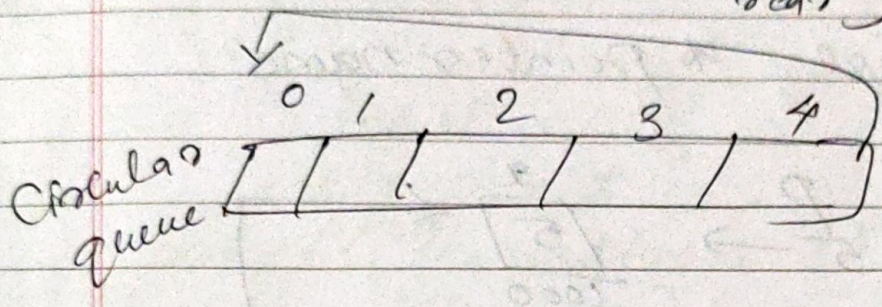
```
Void top()
{ if (front==-1 && rear==-1)
    { printf("empty")
    }
else
   { printf("%d", queue[front]);
   }
```

Drawback

* we can also represent it using linked list.



front } enque call
rear } → (N-1) no space



30/11/20

Array & pointers

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

Syntax of declaration

* Datatype * pointer name

eg: int * p
int x = 5 → 5 (1000) ← q

int * ptr
ptr = &x → 1000

* Value of operation (*) / indirection 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 pointer

```
int a = 5
int *ptr
ptr = &a
printf("%d", *ptr);
```

ie; print the value of ptr dereference operator.

eg:

```
int main()
{
    int i = 10;
    int *p = &i;
printf("The address of variable i is
        %p", p);
    return 0;
}
```

eg:

```
int b=10
int *p;
p = &b;
```

① printf("%d", b)

o/p = 10

② Printf("%d", *p)

& = address

* = dereferencing operator / value of pointer

3. printf("%p", &b) // o/p = 200

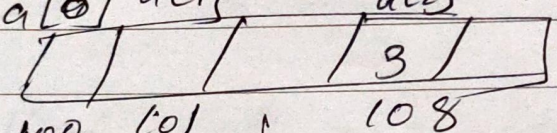
4. printf("%p", p) o/p 200

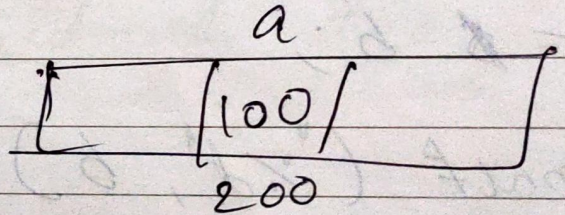
⇒ If we want to print the address of an array, it points the base address.

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

int * q

q = a // q contains base address of a i.e; q = &a[2] // q contains address 108





8/12/20

Relationship between Arrays and Pointers

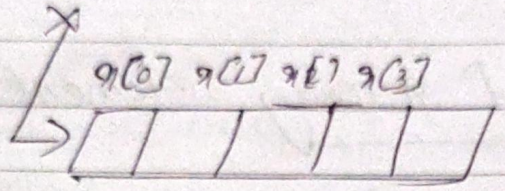
* An array is a block of sequential data.

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. It is bez the variable name[x] points to the first element of the array



* x[0] is equivalent to *x
&x[i] " " " *(x+1)
&x[i] " " " *(x+i)

ex:1 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 += *(a+i);
  }
  printf("Sum = %d", sum);
  return 0;
}
```

o/p

```
Enter 6 numbers: 2
3
4
4
12
12
Sum = 29
```

CN-2 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); //3
printf("*(ptr+1) = %d \n", *(ptr+1)); //4
printf("*(ptr-1) = %d", *(ptr-1)); //2
return 0;
```

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 beez 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

C Dynamic Memory Allocation

* Dynamically allocate memory in your C program using standard library functions:

- malloc()
- calloc()
- free()
- realloc()

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

i malloc()

* The name "malloc" stands for m/y allocation.
* It reserves a block of m/y 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 400 bytes of m/y. Bcz the size of float is 4 bytes. And the pointer ptr holds the address of first byte in the allocated m/y.

2 calloc()

* The name "calloc" stands for Contiguous allocation.

* The malloc() function allocates m/y and leaves the m/y uninitialized. Whereas the calloc() function allocates m/y and initializes all bits to zero.

realloc()

* If the dynamically allocated m/y is insufficient or more than required, you can change the size of previously allocated m/y using the realloc() function.

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

ex:
```
int main() { // ptr is reallocated with new size x
int *ptr, i, n1, n2; printf("Enter Size");
Scanf("%d", &n1);
ptr = (int*) malloc(n1 * sizeof(int));
printf("Addresses of previously allocated m/y:");
for(i=0; i<n1; i++)
printf("%u\n", ptr+i); printf("Enter new Size");
Scanf("%d", &n2); //reallocating the m/y.
ptr = realloc(ptr, n2 * sizeof(int));
printf("Addresses of newly allocated m/y:");
for(i=0; i<n2; i++)
printf("%u\n", ptr+i);
free(ptr);
return 0;
}
```

Linked List

memory

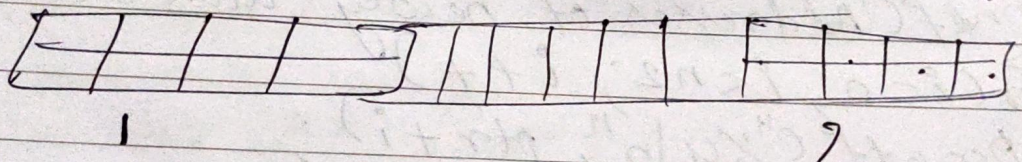
int x; // Compiler allocates [| | | | | | | |]
0 1 2 3 4 5 6 7 8 9

int a[3] → [| | |] continous location
0 1 2

int a[100] → Compiler allocates 100 × 4
400 bytes of m/y.

* If the programmer use only 10 location only, then remaining locations are wasted.
Another program cannot allocate these m/y in future. This is the drawback of array.
To resolve this linked list introduced.

* L. 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

eg: [1 | 100] → [2 | 108] → [3 | ,]
Null

[,] [,]

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;
  Struct node *next;
}
```

[Data | address]

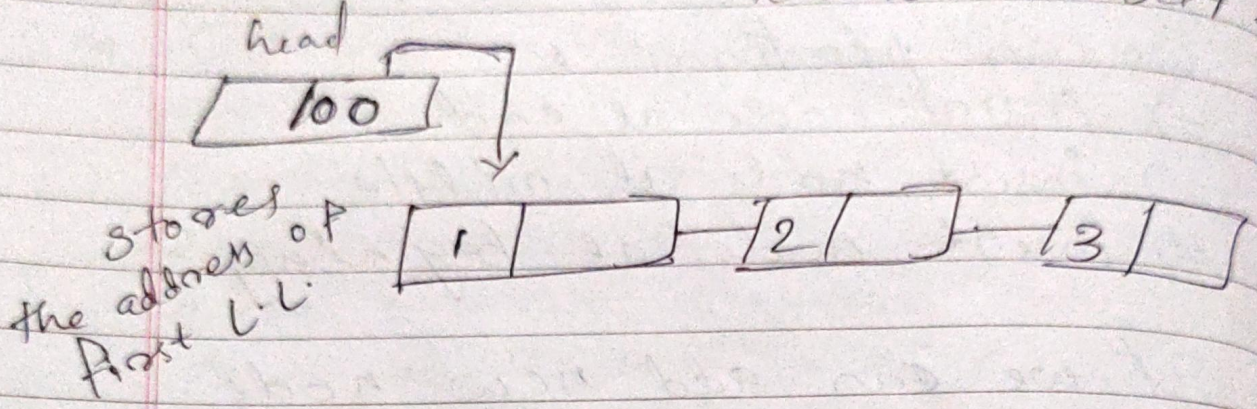
→ Self referential Structure.

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

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

* float *a ; → Stores the address of a float value.

↳ Struct node *next → address of the next node → self.

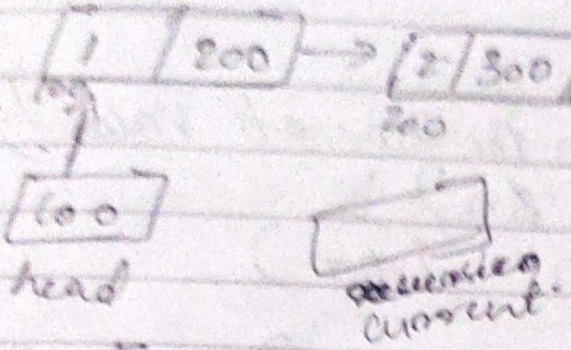


Types of Linked list or Variants

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

2/12/20

Singly Linked list



① // first step create node of the list

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

[data/next]

× Our aim is to create 3 such nodes so 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 value");
 scanf ("%d", &n);
for (i=1; i<n; i++)
{ newnode = (struct node *) malloc (sizeof
                (struct node));

scanf ("%d", &newnode->data);
newnode -> next = NULL
 if (head == NULL)
  { head = newnode;
  current = newnode;
  }
else
  { current -> next = newnode
    current = newnode;
  }
```

Program

```
printf ("Linked list")
for (current = head; current != NULL;
     current = current -> next)
printf ("%d -> ", current -> data);
printf ("NULL");
```

Traversal

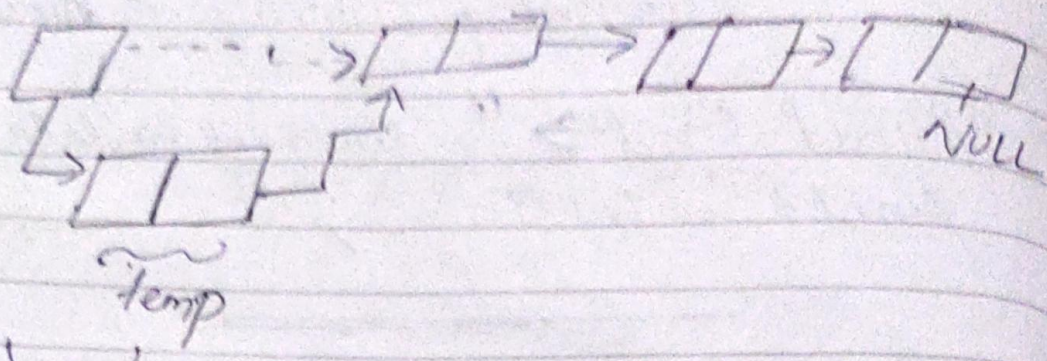
* Moving through the list & visiting each node exactly once.

Traverse (head)

1. Set Current = head
2. Repeat steps 3 to 4 while Current ≠ NULL
3. Apply process (printing) 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

```
temp -> info = data;
```

* 'start' points to the first element of linked list. for insertion of beginning, we assign the value of start to link of temp node.

```
tmp -> link = start;
```

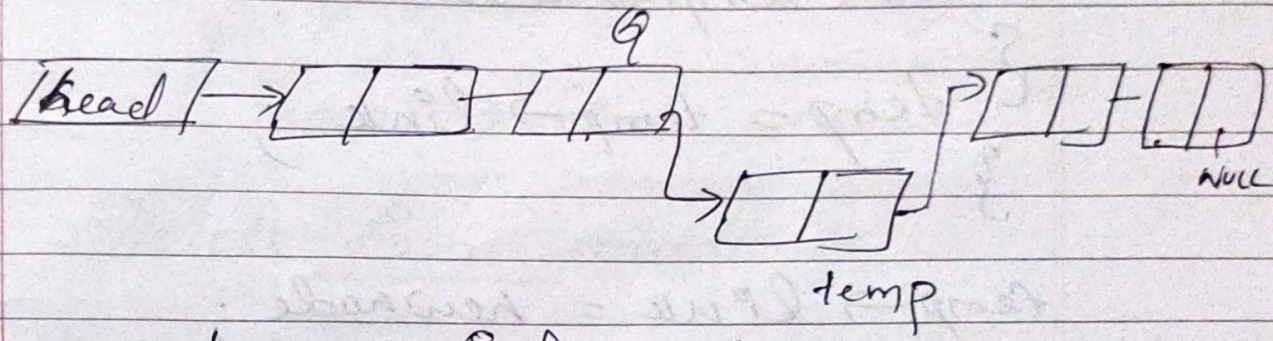
* Now inserted node is first node of L.L so start reassigned as

```
start = temp
```

Case II

Insertion 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.



temp → info = data
temp → link = q → link
q → link = temp

Case III

Insertion at End

* Allocate m/y for new node
* Store data
* Traverse to last node
* Change nent of last node to recently

created node.

```
eg: Struct node * newnode;
    newnode = malloc (sizeof (Struct node))
    newnode -> data = 4;
    newnode -> link = NULL;

    Struct node *temp = head;
    while (temp -> link != NULL)
    {
      temp = temp -> link;
    }

    temp -> link = newnode;
```

Deletion from a Linked List

1. Delete from beginning

* point head to the second node

```
head = head -> link;
```

II Delete from end

* Traverse to second last element
* Change its next pointer to null

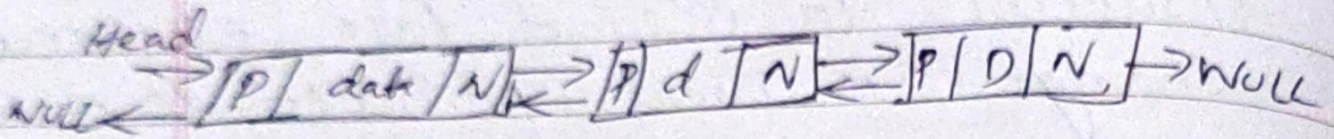
ex.
```
Struct node *temp = head;
while (temp -> link -> link != NULL)
    {
    temp = temp -> link;
    }
temp -> link = NULL;
```

III Delete from middle

* Traverse to element before the element to be deleted.
* 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 -> li
```

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.

↳ 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.

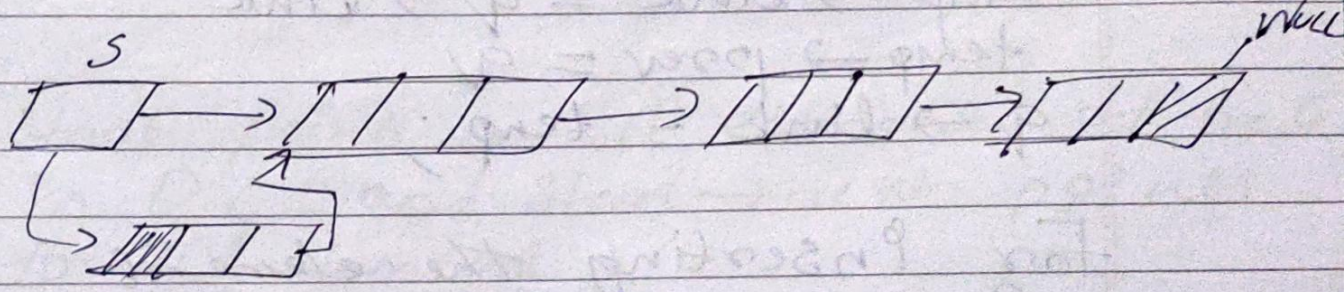
↳ for processing the next element, we assign the address of next node to ptr as

ptr = ptr → next.

* we can traverse until ptr has NULL value.

while (ptr != NULL)
ptr = ptr → next.

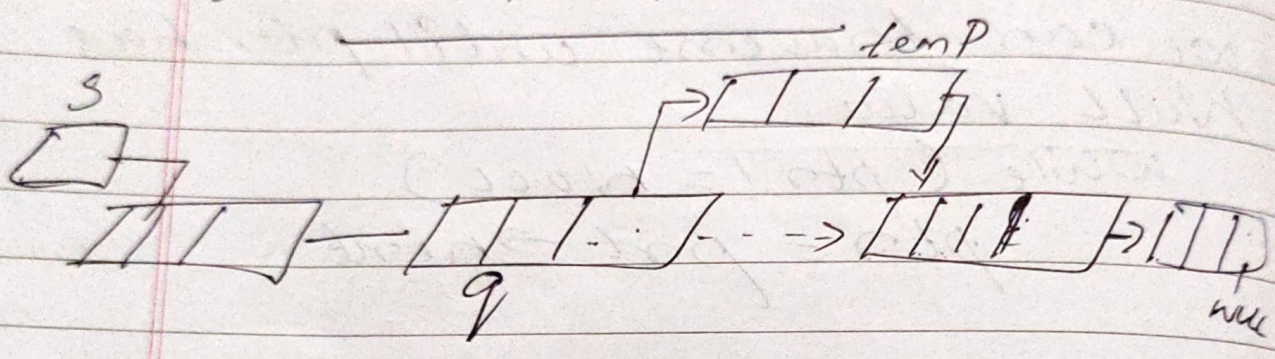
2. Insertion into D.L.L



or
```
Struct node * temp
temp = malloc (sizeof(Struct node))
temp -> info = data
temp -> prev = NULL
temp -> next = start
start -> prev = tmp
start = temp
```

Case 2

Insertion in Middle




```
q -> link -> prev = temp
temp -> link = q -> link
temp -> prev = q
q -> link = temp;
```

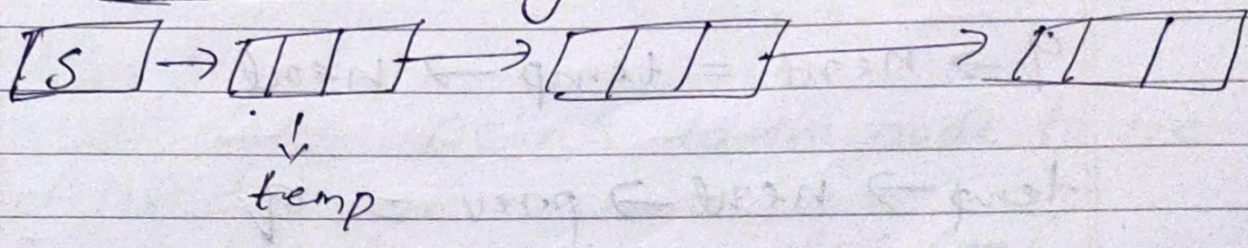
For inserting the element after the node we assign the address of inserted node to

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.

(8) Deletion

Case:1 Deletion at beginning

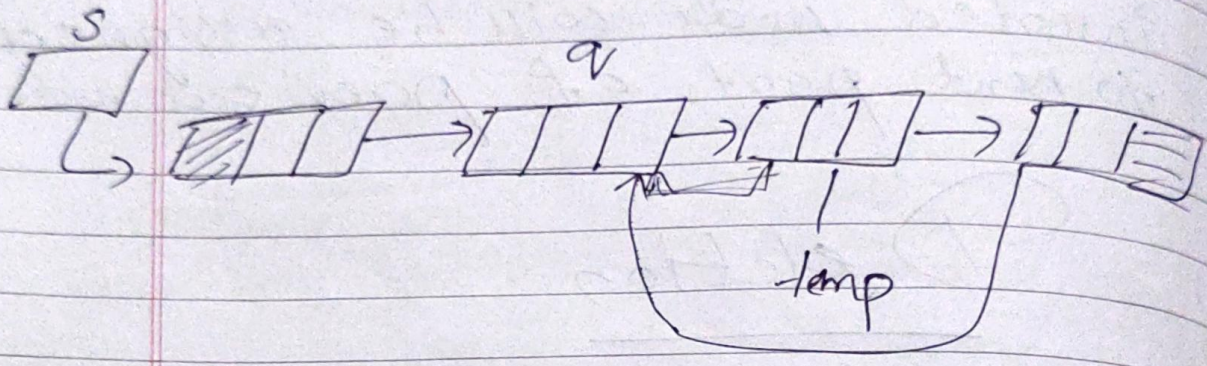


* Start points to 1st element of node in D.L.L and Start → next points to IInd element

temp = start (1st element)

Start = Start → next (IInd element)
Start → prev = NULL
free (temp);

Case 2
Deletion from Middle



temp = q → next

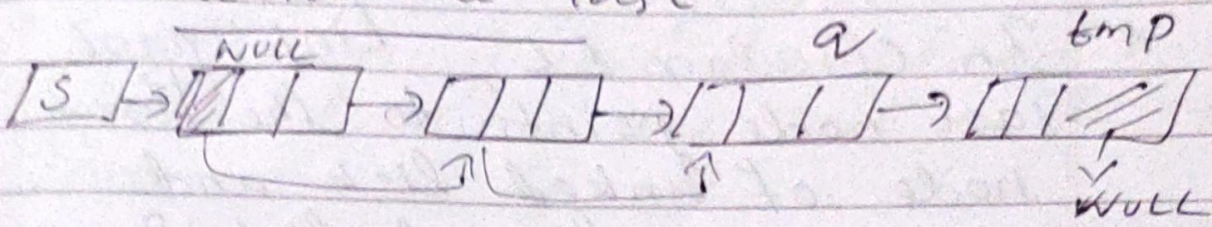
q → next = temp → next

temp → next → prev = q

free (temp)

Case 3

Deletion at last



① temp = q → next

free (temp)

q → next = NULL

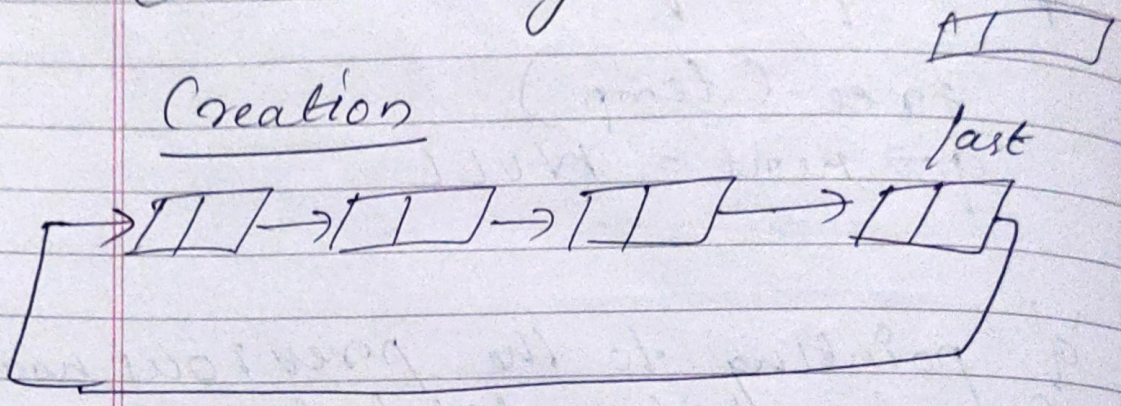
② 'q' pointing to the previous node of node to be deleted.

③ 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.



*1 Newo element can be added as.

```
temp → link = last → link
last → link = temp
last = temp
```

*2 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 ②, pointer variable "last" will point to added node which is now the last node of list.

Operations

1. Traversal

Traversal need a pointer variable which ~~poor~~ 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.

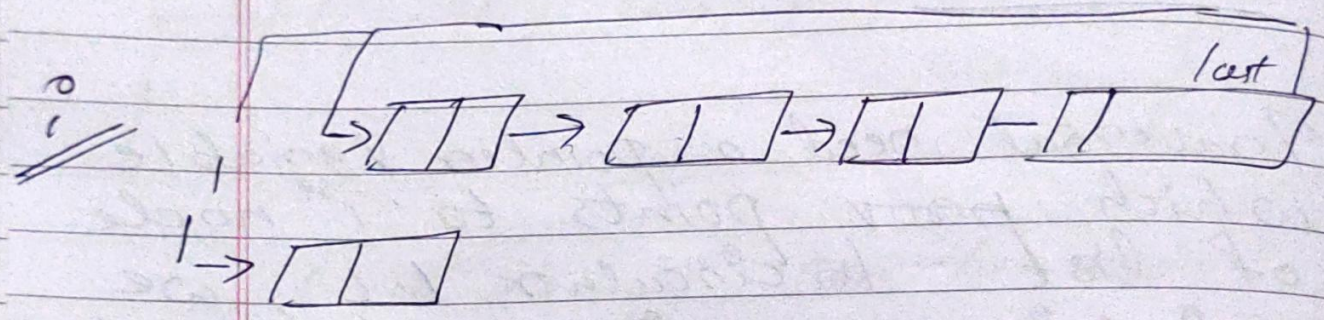
q → last

q - last → link

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

(2) Insertion

i Insertion at beginning
ii Insertion in between.



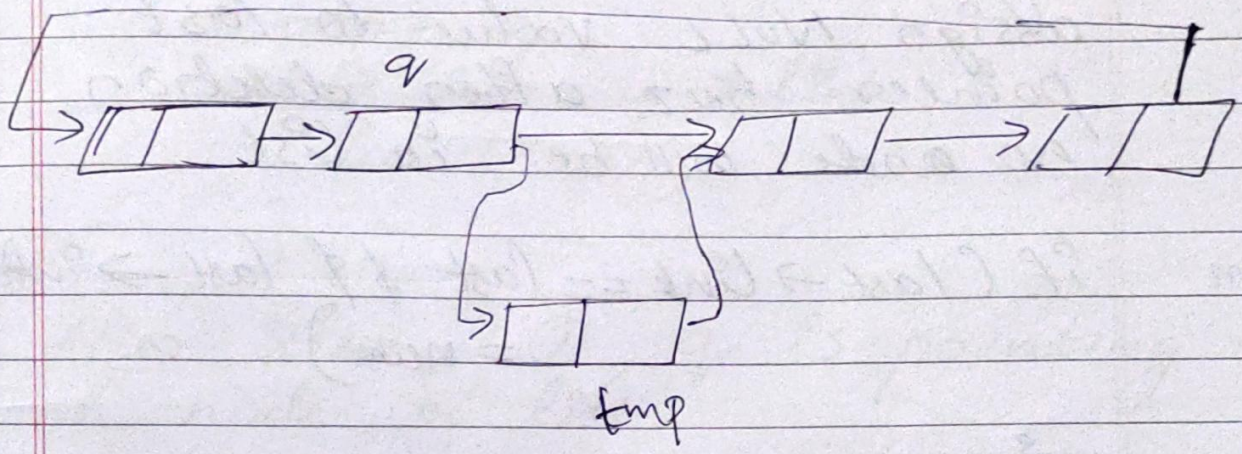

```
tmp -> data = data
tmp -> link = last -> link
last -> link = tmp
```

* After Statement 2 inserted node points to previous 1st 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




```
tmp -> link = q -> link
tmp -> info = num
q -> link = tmp;
```

3 Deletion – 4 cases

i If list has only one element
ii Note 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);
     }
```

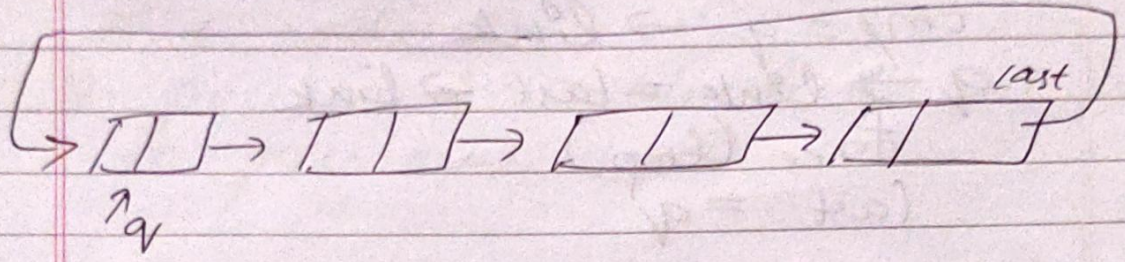
Case 2: Delete at first

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.



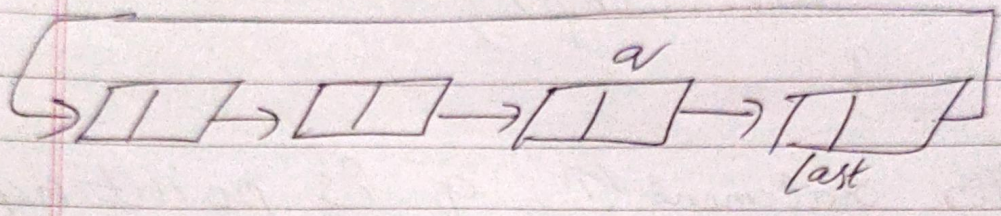
Case 3 Deletion in between

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

```
if (q -> link -> info == num)
{
  tmp = q -> link
  q -> link = tmp -> link
  free (tmp)
```

Case 4

Deletion at last




```
tmp = q -> link
q -> link = last -> link
free (tmp)
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' here after deletion of last node pointer variable last should point to the previous node.

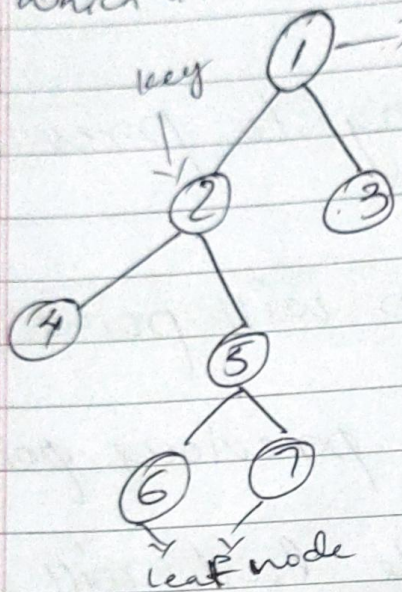
* Here q is pointing to previous node.

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.

14/12/20

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), (5) are non-terminal nodes

(4), (3), (6), (7) are leaf nodes.

Node

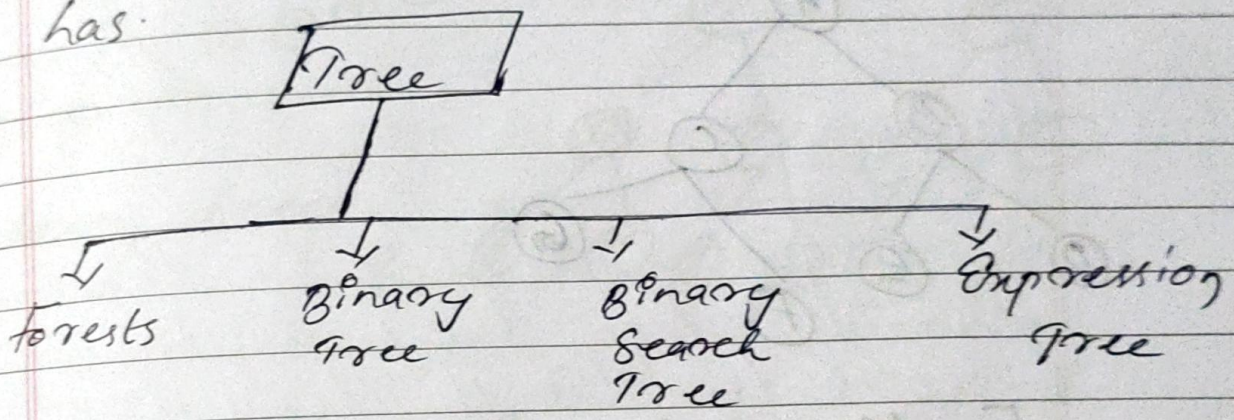
i Root node:- Top most node in the tree hierarchy.

ii Leaf node: It is bottom most node in a tree hierarchy.

iii Parent node: Any node except the root

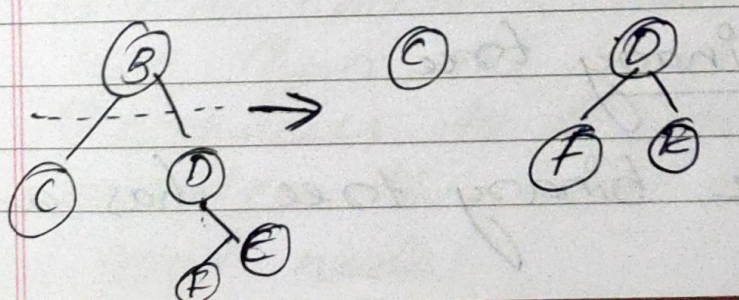
node that has a child node and an edge upward towards the parent.

- **key**: It represents the value of a node.
- path: The path is a sequence of consecutive edges.
- 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 trees.

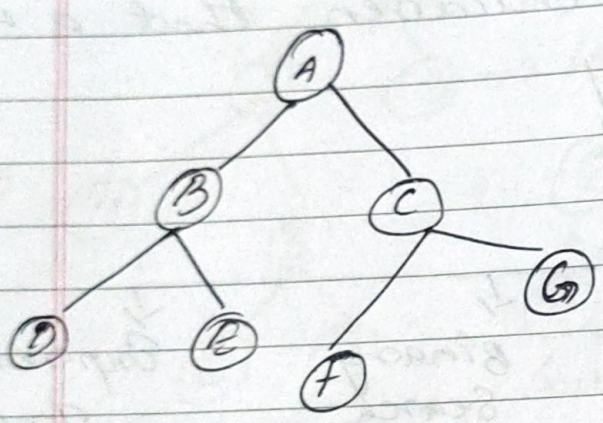
eg.



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.



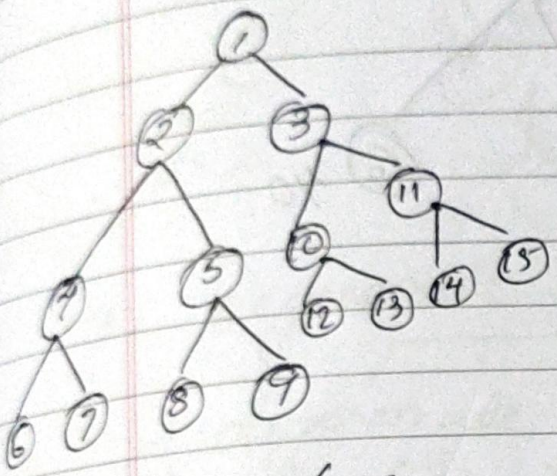
3. Full binary tree

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

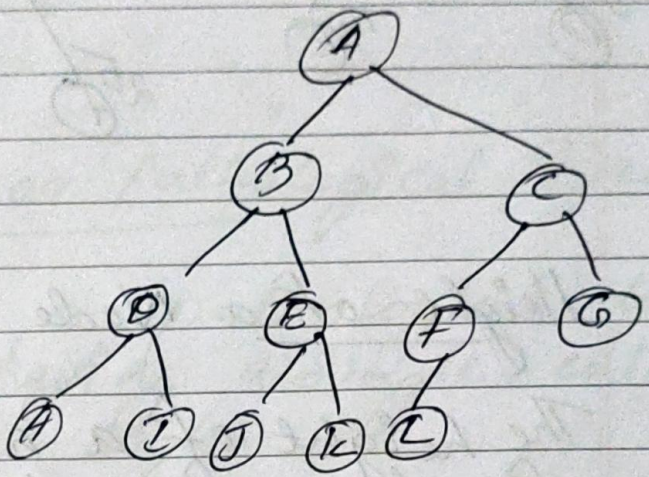
Complete binary tree

A complete binary tree has a

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



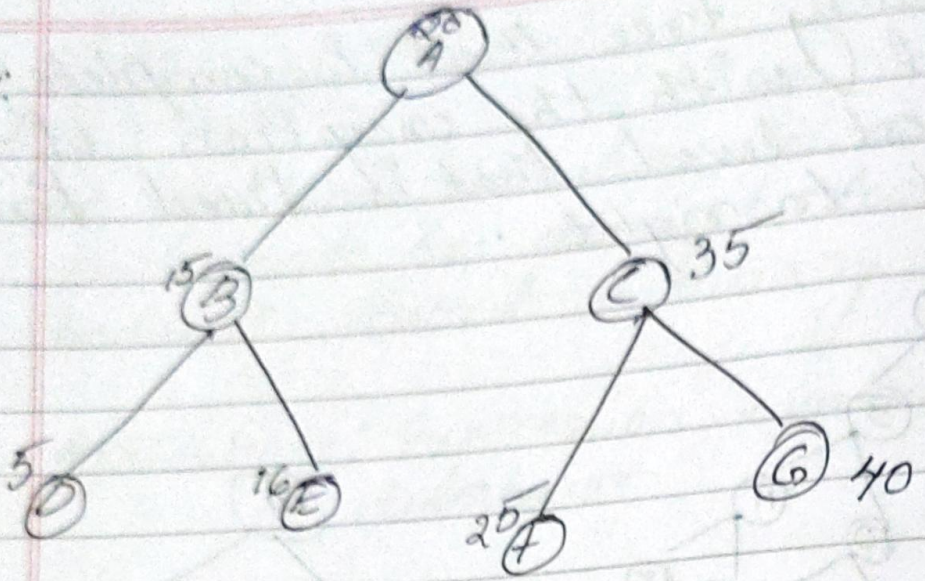
Full binary tree



3. Binary Search Tree

The binary Search tree is the binary tree that is ordered ~~is~~ 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.

Ex:



Height of a Node

The height of a node is the number of edges from the node to the deepest leaf (ie 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.

15/12/20

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.

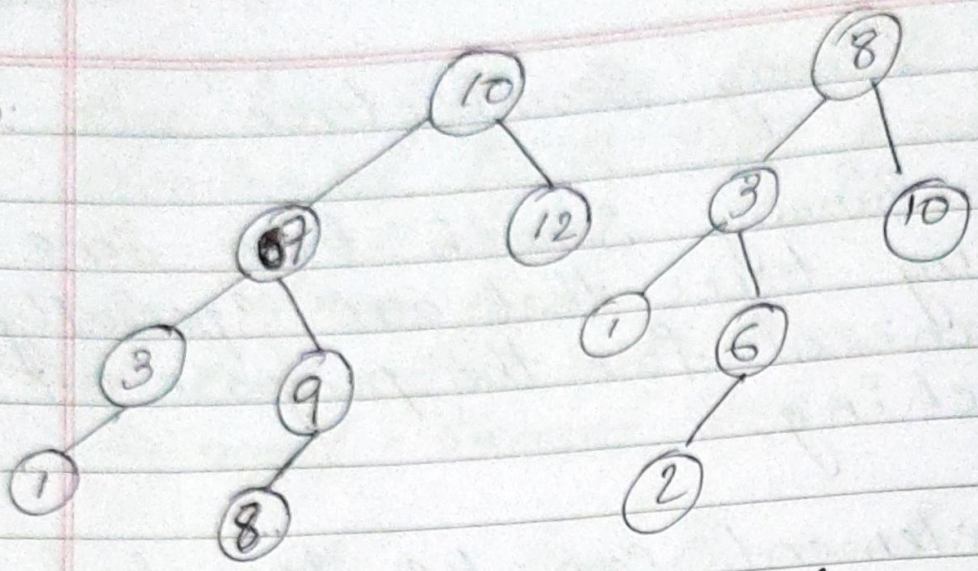
A binary Search tree is a binary tree that may be empty and if it is not empty then it Satisfies 2 properties

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.

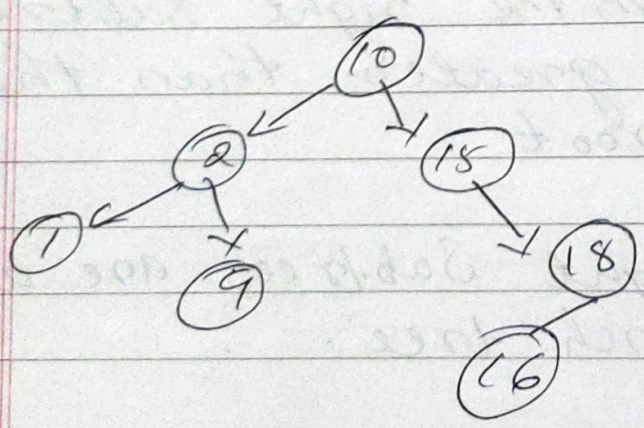
Q9.



✓

✗

- 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

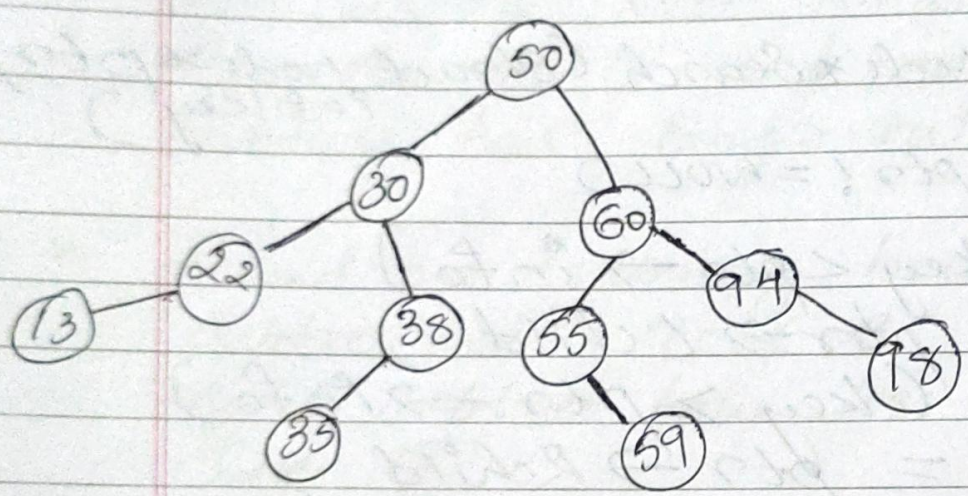
```
struct node *Search (struct node *ptr, int key)
while (ptr != NULL)
{ if (key < ptr -> info)
    ptr = ptr -> Lchild;
  else if (key > ptr -> info)
    ptr = ptr -> Rchild;

  else
    return ptr;
  }
  return NULL
```

Operations On BST

i. Insertion
ii. Deletion

(i) 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 *tmp, *par, *ptr;
    ptr = root;
    par = NULL
  while (ptr != NULL)
    {
      par = ptr;
  if (key < ptr->info)
      ptr = ptr->lchild;
  else if (key > ptr->info)
      ptr = ptr->rchild;
  else
    {
      printf("Duplicate key");
        return root;
      }

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

tmp->info = key;
```

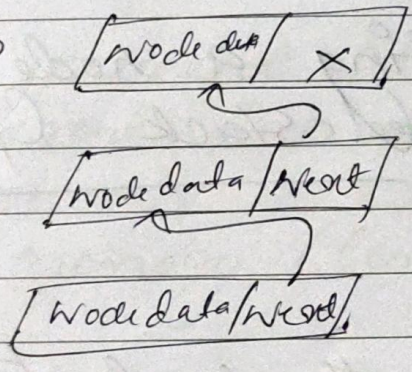
```
tmp -> Lchild = NULL;
tmp -> Rchild = NULL;
if (par == NULL)
    root = tmp;
else if (key < par -> info)
  par -> Lchild = tmp;

/* 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 m/y 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)

1. Create a node first and allocate m/y 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
   Ⓢ elements are popped only from one end, therefore the value stored in head pointer must be deleted and the node must be freed.

28/12/20

Bit string

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

Any subset of universal set in 0 & 1 form is called bit string.

eg: $A = \{1, 3, 5\}$

$A = \{1\ 0\ 1\ 0\ 1\}$ ← bit string

$B = \{\}$

$= \{0, 0, 0, 0, 0\}$ ← Bit string

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

AND (intersection) OR - Union

A∩B

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

A∪B

0 0 - 0
0 1 - 1
1 0 - 1
1 1 - 1

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

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

$A \cap B$ = A intersection B = A AND B

all common elements

$C = \{5\}$

In bit string ⇒ AND operation

⇒ Intersection. ($\cap$)

OR ⇒ OR operation ⇒ Union ($\vee$)

$\cap \Leftrightarrow$ AND ; $\vee \Leftrightarrow$ OR

Ans (a) $A = \{0, 1, 0, 1, 1\}$

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

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

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

(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. $\overline{A}$

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

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

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

Difference

$(A - B) = \{x \mid x \in A \text{ and } x \notin B\}$

29/12/20

Hashing

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

0	0 Ani
1	1 Anu
2	2 Ajay
3	3 Arul
4	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.

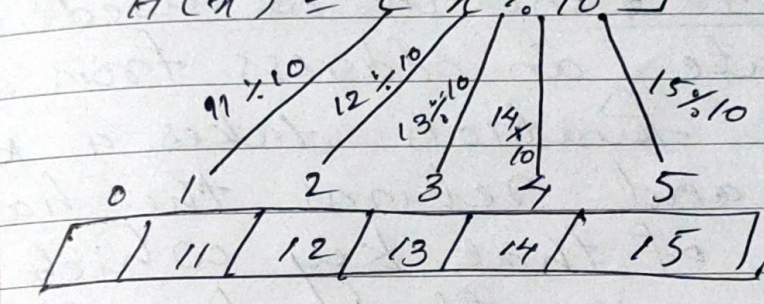
* The efficiency of mapping depends of the efficiency of the hash function used.

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

Hashing data Structure

List = [11, 12, 13, 14, 15]

H(x) = [x %. 10]



* In hashing we search any record with key Ik and we access any record without comparison. This method is called direct addressing.

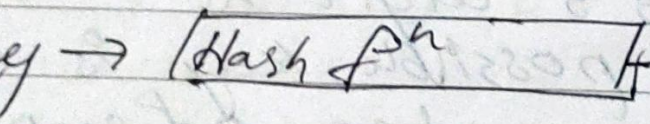
Eg: There is 500 employee but their ID is 5 digit no: 00000 to 99999. Here possible key is much more than number of employee, only

Soo 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.

OR 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.



Collision

The situation collision occurs when the hash f^n generates same address for different keys.

⇒ Algorithm for hashing is focused mainly on :-

1. Selecting a hash f^n that converts keys to addresses
2. Resolving the collision

Hash function

A good hash f^n have two criteria

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

Method

1. Truncation

Easiest method for computing address from a key. Here we take a part of key as address ie; take right most or left most digit.

8239456 8356937

* Take address as 56, 71.

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

2 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 addres

3 Folding method

ex: 738239456527

× Break the key into parts

ie; 738 239 456 527

× After this, parts are shifted and added.

```
  738
  239
  456
  527
 ----
 1960
```

× Now ignore the final carry 1 the hash address for the key's is 960. This technique is shift folding.

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

```
  738
  932  (Rev)
  456
  725  (Rev)
 ------
 2851
```

4. Ignore carry '2', address is 851

4. Division method / Module-Division

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

ie; $H(K) = k \bmod m$

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

(L) In the process of searching given key is compared with many keys and each key comparison is known as probe

(L) 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.

Open addressing

In open addressing the key which caused the collision

is placed inside the hash-table itself but a 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 fn is 'a' and it is not empty, then we will try to insert the key at next position. or location.

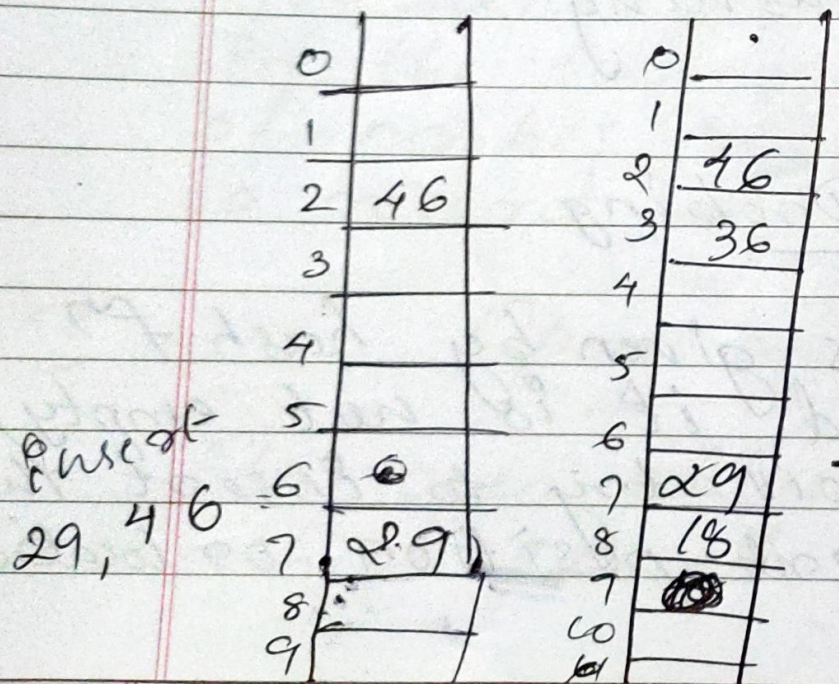
ie; address a+1. If a+1 is occupied then try to insert at next address a+2

eg 29, 46, 18, 36, 43, 21, 24, 54

Total table size = 11

h(key) = 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



formula

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

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

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

ie; $h(k)$, $h(k)+1$, $h(k)+4$, $h(k)+9$ ..... all mod TSIZE

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

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

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

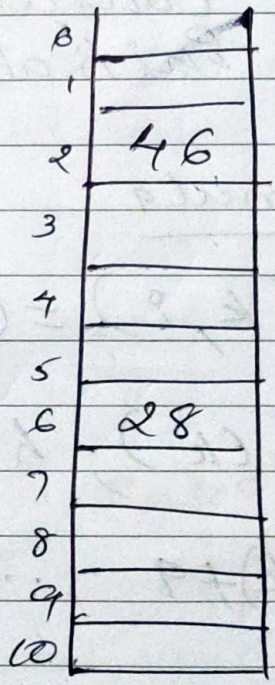
$h(k) = h(35) = 35 \% 11 = 2$

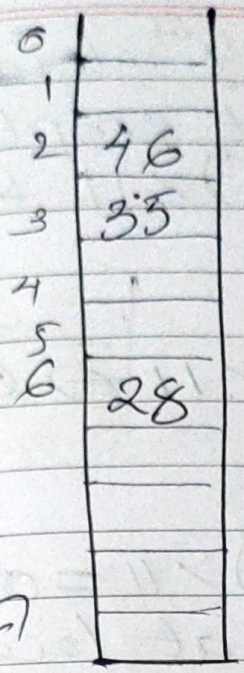
$h(k) = h(57) = 57 \% 11 = 2$

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

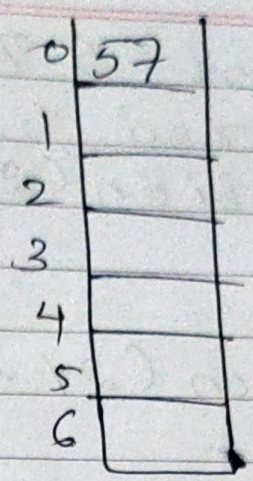
$h(k) = h(50) = 50 \% 11 = 6$

i) insert 46, 28





ii Insert 35



iii. Insert 57

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)%11 = 3 is tried and it is empty, 35 is inserted in location 3.

iii To insert 57, 1st location
→ 2 is tried ; not empty
→ 2nd location (2+1)%11 → not empty
→ 3rd location (2+4)%11 = 6, not empty
→ 4th location (2+9)%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

$$H(k,i) = (h(k) + i\,h'(k)) \bmod TSize$$

* Value of i varies from 0 to TSize-1
h' is hash function. Search of empty location will be in the sequence :- $h(k)$, $h(k)+h'(k)$, $h(k)+2h'(k)$, $h(k)+3h'(k)$ ... all mod TSize.

eg. {46, 28, 21, 35, 57, 39, 19, 50} 8 elements

$h(k) = key \% 11$

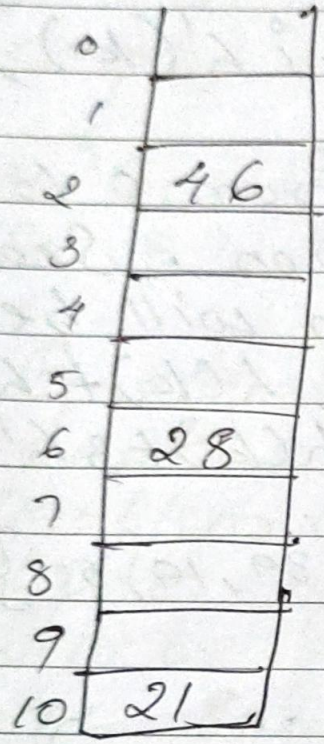
$h'(k) = 7 - (key \% 7)$

$h(46) = 46 \% 11 = 2$
$h(28) = 28 \% 11 = 6$
$h(35) = 35 \% 11 = 2$
$h(21) = 21 \% 11 = 10$
$h(57) = 57 \% 11 = 2$
$h(39) = 39 \% 11 = 6$

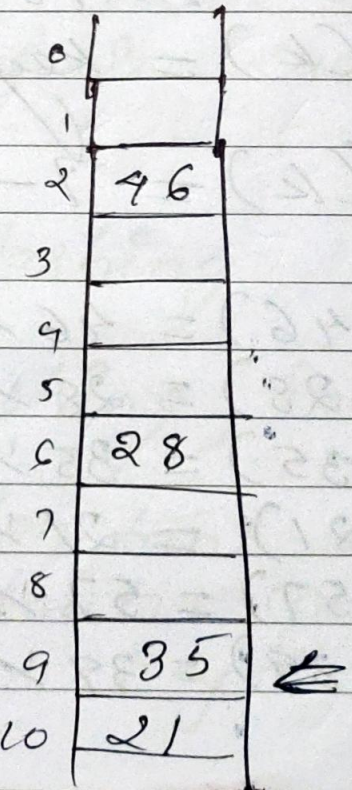
$h(19) = 19 \% 11 = 8$

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

i Insert 46, 28, 21



ii Insert 35



To Insert 57, collision is at 2 occurs.

∴ Next probe

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

$= 2 + (7 - 57 \% 7) \% 11$

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

$= 8 \% 11$

empty location is 8

∴ 57 is at location 8

iv insert 39

Collision occurs at 6

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

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

$\Rightarrow 6 + (7 - 4)$

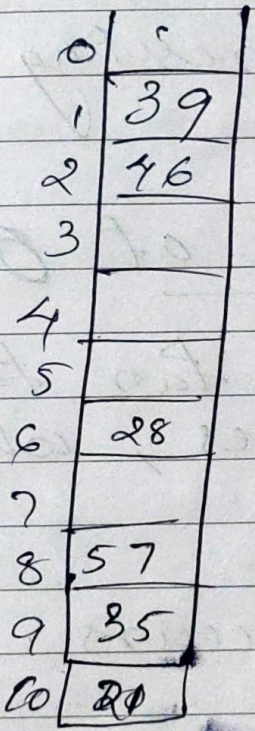
$\Rightarrow 6 + 3 = 9 \% 11 = 9$

But 9 is not empty.

Next probe.

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

$6 + 2(7 - 39 \% 7) \% 11$
$6 + 2(3) \% 11$
$6 + 6 \% 11$
$= 12 \% 11$
$= 1$, empty location



* Double hashing is complex because 2 times calculation of hash fn.

* Load factor denoted by λ and is calculated as

$$\lambda = \frac{n}{m}$$

where,
n = no. of records
m = no. of position

* In open addressing load factor is ≤ 1.

Disadvantage of O.A

1. Key values far from their home addresses, which increase probes.

2. Overflow occurs.

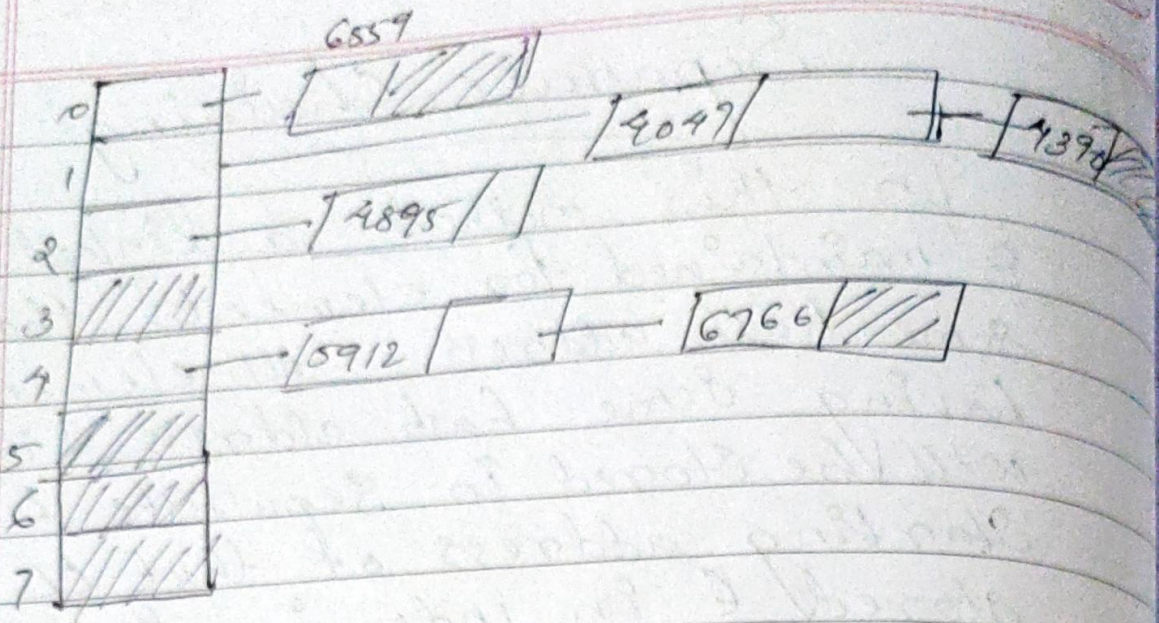
Separate Chaining

In this method, a linked list is maintained for elements that have same 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 reffered to as chains and hence the method is named as separate chaining.

Suppose hash table of size 7

H(key) = key % 7
H(4895) = 4895 % 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
  → extra space n + m

In Separate chaining load factor $(\lambda) \geq 1$

Bucket Hashing

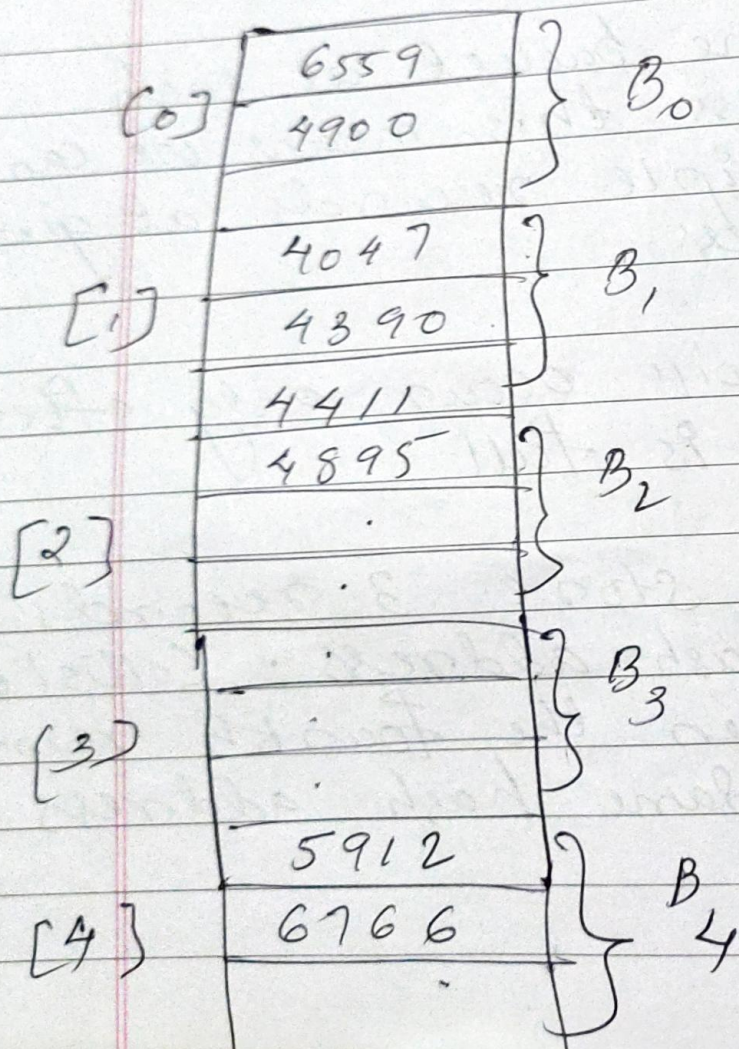
- This technique postpones collision but doesnot resolve them completely. Here hashtable is made up of buckets, where each bucket hold number of records.

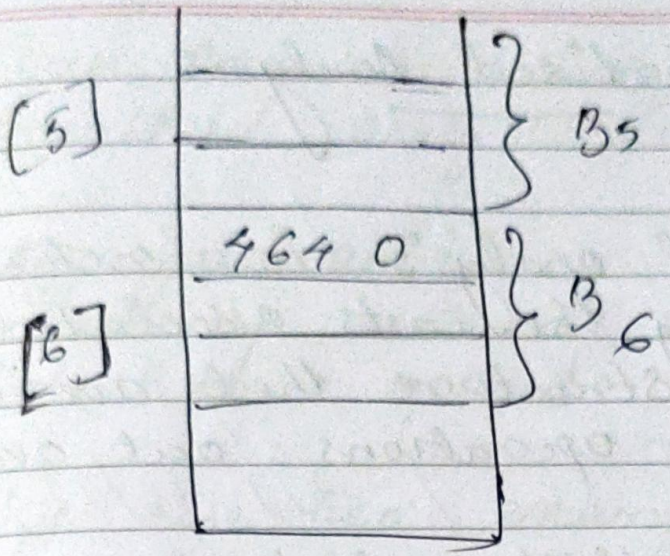
- There is one bucket at each hash address this means we can store multiple records at given hash address.

- Collision will occur only after a bucket is full

eg. A bucket store 3 records at same hash address. Collision occurs when the fourth record with the same hash address arrives.

H(4895) = 4895 % 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
H(4640) = 4640 % 7 = 6
H(4900) = 4900 % 7 = 0
H(4411) = 4411 % 7 = 1





Disadvantage

1. Wastage of Space since many buckets will not be occupied. or will be partially occupied.

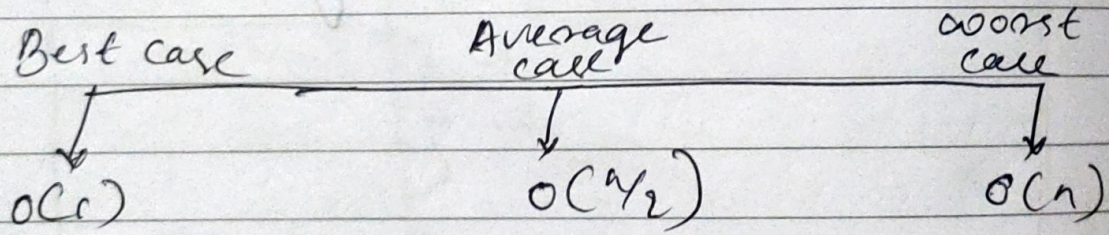
2. Does not prevent collision but only defers them, when collision occurs. resolved using techniques in open addressing.

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 operation 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

1. 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 operation

Two fundamental stack operation each of which takes O(1) time:

i Push (S, x) → Pushes object x on stack S

ii Pop (S) → pops the top of stack S and returns the popped object. Calling pop on empty stack generates an error.

n push & n pop total running time for n operation ~~O(n)~~ Θ(n).

iii Operation multipop (s, k) → which removes k top objects of stack S, popping the entire stack if the stack contains fewer than k objects.

eg: top → 23, 17, 6, 39, 10, 47

multipop (S, 4) → 10, 47 → multipop (S, 7) ⇒ empty stack

Code

```
Multipop (s, k)
while stack not empty and k > 0:
    k = k - 1
    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 multipop, pop 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.

~~Accounting Method~~

eg 2 Incrementing a binary counter

index (i)

$k = 4$

Row No	3	2	1	0 ← 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
	$n/2^3$	$n/2^2$	$n/2^1$	$n/2^0$

Code

```
Increment (A, k)
{
  int i = 0
while (i<k && A[i] == 1)
  {
    A[i] = 0
    i = i + 1
  }
if (i<k)  // 0<4
  {
    A[i] = 1   // 0th changes to 1 in the next array
  }
}
```

push all 0 to the 1st array.

Step 2

Pass 2^{nd} array.

No. of bit = $k = 4$

Sequence 1, operation = 16

ie; $n = 16 = 2^k$

$\Rightarrow 16 = 2^4$

$\therefore n = 2^k$

$k = \log n$

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

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

$\frac{n}{2^3} + \cdots \frac{n}{2^i}$

$n = 16$

$\therefore n \sum_{k=1}^{k-1} \frac{1}{2^i}$

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

= Geometric progression

Sum to infinite terms

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

$\cdots \infty$

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

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

$a =$ 1st term ie: 1

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

$= \langle 2 n$

$$\text{Aggregate} = \frac{O(2n)}{n} = O(1)$$

Constant

II

Accounting Method

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

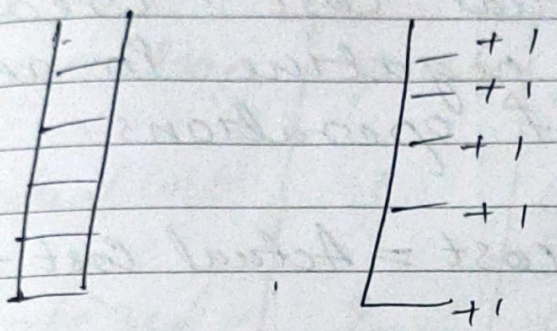
* In this method, borrows ideas & terms from accounting.

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 step)

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.

Credit = Amortized cost − Actual cost
= 10 − 5 = 5 //

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

{ Amortized cost = Actual Cost + Credit }

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

eg: Stack

operation	Push	pop	Multipop
Actual cost	1	1	min(n, k) (pop operation on k items, n = S = size of Stack; k = No

of pop.

Step 1 Actual cost

push = 1

pop = 1

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

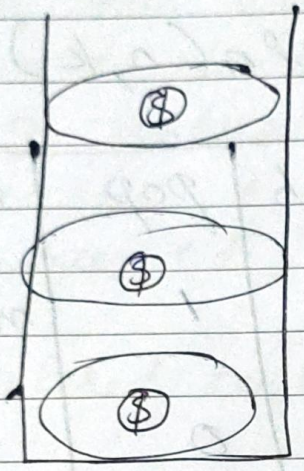
⇒ 2 − 1 = 1

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

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

eg: Push a plate require 2 coins

I



1 $ Actual cost

1 $ credit → put credit at top of plate.

ie; we pay $1 for the actual cost of the operation $ we are left with $1 of credit keep left $ on the plate.

II) Then apply pop operation
↳ need $1 to pop the plate off
→ 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 1st time = $1.

III) Multipop → 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 n operation is O(n)

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

ie: * No plate
credit = 0

* 1 plate
credit = 2

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 operation

* 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 operation
i.e. O(n)

III

Potential Method

- This is 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, multipop)

⇒ 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 $D_1, D_2, D_3 \cdots D_n$

D_1 ↑ 1st stage, D_2 ↑ 2nd stage, D_n ↑ nstage.

PE changes on 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.

ϕ is the potential function
↳ which maps the data structure D to a number $\phi(D)$, the potential associated with the data structu

$D_0 \rightarrow$ initial DS
- 1st operation (stack) then D_1 - potential
- 2nd operation (stack)
(push, pop ...)

then D_2 - potential
- n operation
D_n potential
- cost associated with each step.

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

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

C_i : cost

$\phi(D_i) - \phi(D_{i-1})$: change in potential due to operation

// for 'n' operation cost

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

$\phi(D_n)$ ↓ Current stage

$\phi(D_0)$ ↓ initial stage

[Change 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 datastructure is undercharged.

eg: Stack

1) Push operation [size = n]

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

$=$ (size of stack + 1) − size of stack

(D_i) Current Size = Previous Size + 1

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

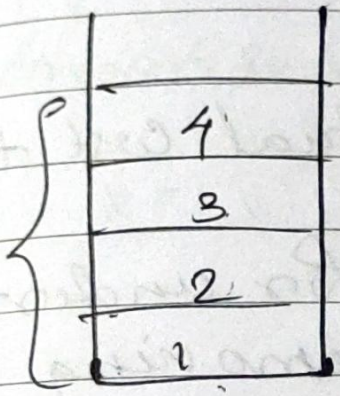
Potential Change $= 1$

Amortized cost = actual cost + potential change.

$= C_i + 1$

$= 1 + 1 = 2$

(a.)



So potential change on stack after performing push operation is 2

Previous size of stack = 4

(D_{i-1}) size = 4 → Initial or Previous

D_i = size = 5 → Current.

$\Rightarrow n+1-n$

$\Rightarrow 4+1-4$

$\Rightarrow 5-4$

$= 1$

C_i = One step push operation

II Pop operation

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

$= \left(\begin{matrix}\text{Size of} \\ \text{stack}\end{matrix} - 1\right) - \text{size of stack}$

$= (n-1) - n$

$= -1$

Amortized cost = actual cost + potential change

actual cost $\Rightarrow$ 1 (check for underflow & removing)

Amortized $= 1 + (-1) \Rightarrow 0$

∴ Amortized cost of ordinary pop operation $= 0$

III Multipop

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

$\Rightarrow (t-1) - t$

$= -t$

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

- t: it becomes underflow
- s: Stack size
- k: No. of popped

Amortized Cost = Actual Cost + potential Change

$= t - t$

$= 0$

ie, Push operation = 2
pop operation = 0
Multipop = 0

} All constant.

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

Amortized cost of n operation

$\Rightarrow n * O(1) = O(n)$

Incrementing a binary Counter

Example 2: We define the potential of the Counter after the ith increment operation to be b_i, the number of 1's in the counter after the ith 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 (17.2) as

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

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

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

$= 2n - b_n + b_0$

							Amortized cost
R0	3	2	1	0	ϕ	c_i	$c_i + \phi_i - \phi(i-1)$
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	$2 + (2 - 2) = 2$
8	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	$2 + (3 - 3) = 2$
16	1	1	1	1	4	1	$1 + (4 - 3) = 2$

c_i = no. of bits

$b_0 \leq k$ [k = no. of bits]

$k = O(n)$ = total actual cost $O(n)$

∴ So increment change only 1 bit from 0 to 1, and amortized cost increase is 2//

classmate
Date 7/1/2021
Page

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 set.

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

$S_y = \{1, 2\}$ ⇒ These two sets are disjoint becz there is no common element in both set.

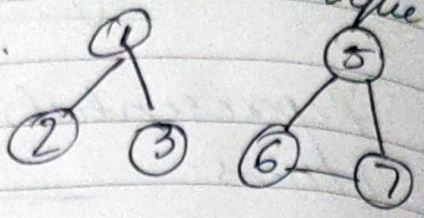
Disjoint set operation

1. MAKE-SET(x) → Creates a new set whose only member (representative) is pointed to by by x.
2. UNION(x, y) → Unites dynamic sets that contain x and y i.e., S_x & S_y into new set $S_x \cup S_y$.

ex. $S_x = \{1, 2, 3\}$ $S_y = \{5, 6, 7\}$
new $S = \{1, 2, 3, 5, 6, 7\}$

3. FindSet(x) - returns a pointer to

the representative of the (unique) set containing x.

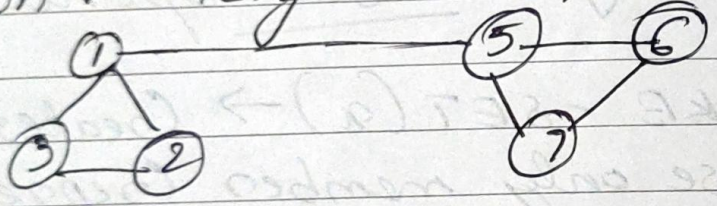


UNION - FIND Algorithm

A union - Find algorithm performs two operation

a. Find: Determine which set a particular element is in.

b. Union: Merge or combine two sets



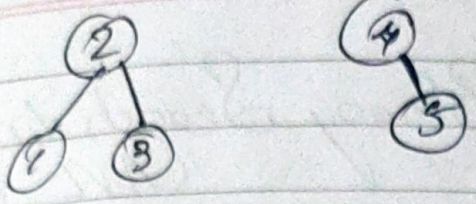
$S_1 \cup S_2$

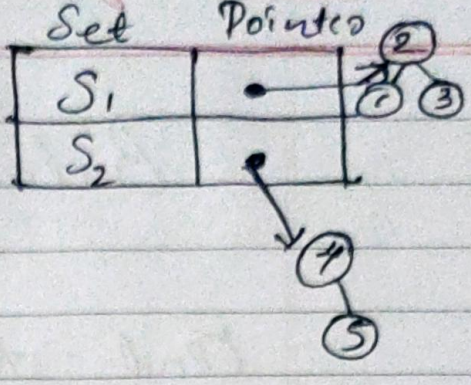
Connected Component $S_1 \cup S_2$

(1) Disjoint Set Representation

i. Linked list

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





ii Array

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

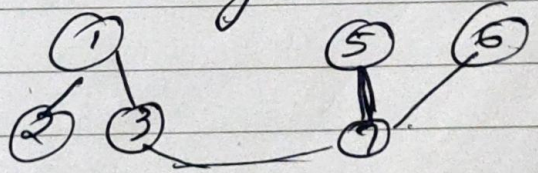
Two or more set with nothing is common are called disjoint sets.

* Find Operation:

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

belongs to S_1 ← ie: Find (3)
belongs to S_2 ← ie: Find (5)

Union: Merge two Set



Union - Find implementation

1. MakeSet (i) → from 0 to 9

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

Union-Set (2,1) → {0}{2,1}{3}{4}{5}...

Union-Set (4,3) → {0}{2,1}{4,3}
{5}{6}...

Union-Set (8,4) → {0}{2,1}{4,3,8}
{5}{6}...

Union-Set (8,9) → {0}{2,1}{4,3,8,9}
{6}{5}{7}...

2. x Are 8 & 4 Connected?

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