Gangaa Karthikeyan. G1
ROLL NO : 19

## Advanced Data Structures

2)

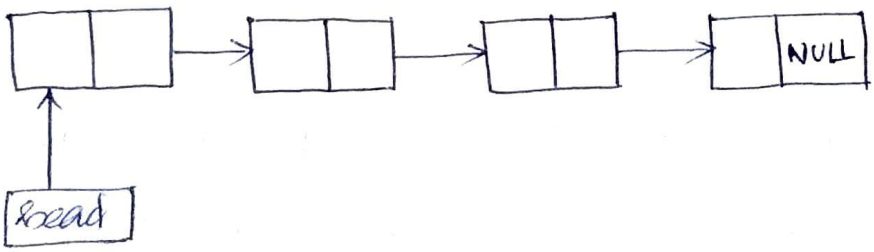
The LinkedLists were introduced to overcome the drawback of array. The LinkedList is a collection of data items in which elements are not allocated in a continuous memory cell.

The LinkedList is a collection of nodes in which each node consists of 2 parts.

→ Data

→ Link — Link is a pointer to the next node ie, it holds the address of the next node

The structure of a node of singly LinkedList is


The head is a pointer which points to the first node of the LinkedList and the last node of the LinkedList points to NULL

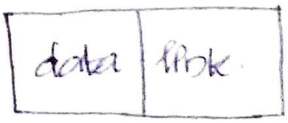
The operations that can be performed on a singly linked list are :

→ creating a node
→ Insert node at beginning
→ Insert node at end
→ insert node in between
→ deletion at front
→ deletion at end
→ deletion in between

head
100

creating the structure of a node

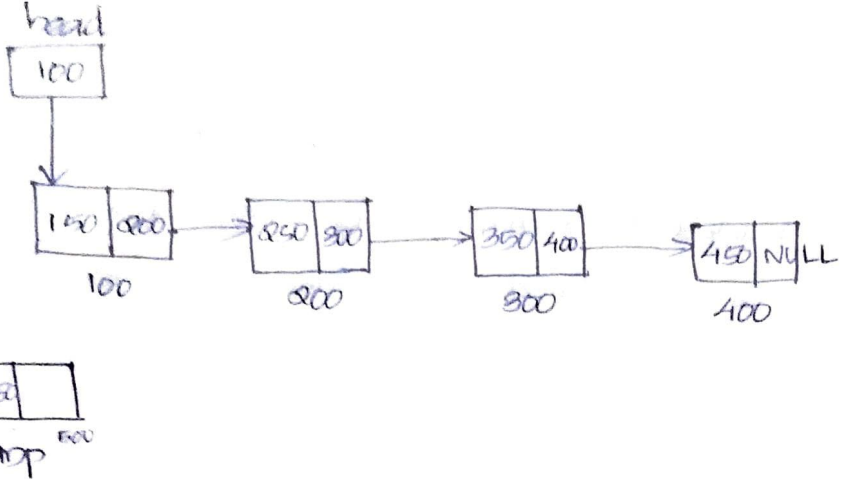
```
struct NODE
{
int data;
struct node *link;
}
```




This is called self referential structure Because the pointer points to a structure of the same type.

Insertion

(1) Insertion at beginning



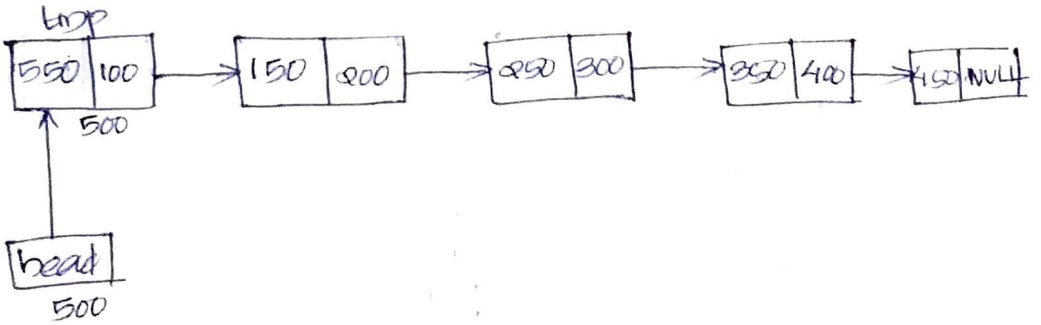




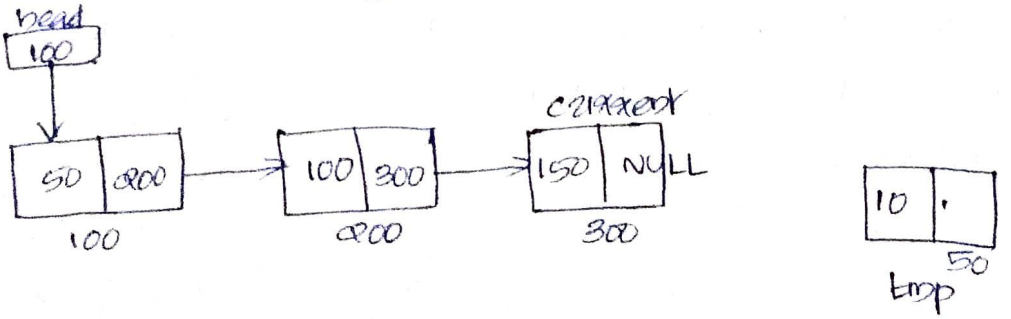
Here temp is the node to be inserted. head is the pointer that points to the 1st node in the linked list. To insert a new node in the beginning do the following steps

```
temp -> data = x;
temp -> link = head;
head = temp;
```

Now the new node is inserted at beginning



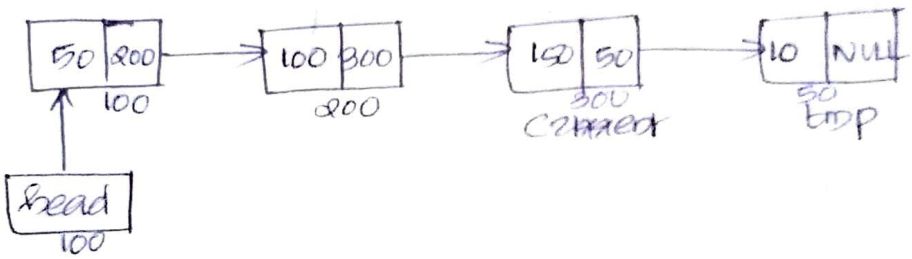

(ii) Insertion at end



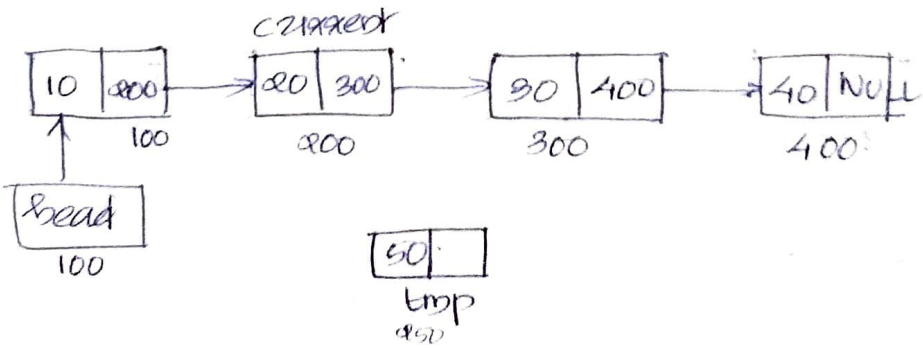
the last node of the linked list points to NULL instead to make the newnode 'tmp' as the last node of the linked list Make the following changes

```
tmp -> data = x;
current -> link = tmp;
tmp -> link = NULL;
```

Now the newnode is inserted at the end



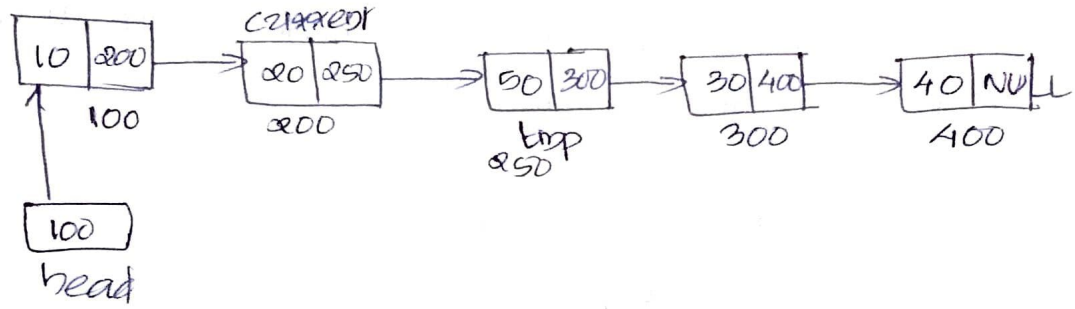

(iii) Insertion in between




'tmp' points to the newnode to be inserted after the current node to insert the newnode at a specified position Make the following changes.

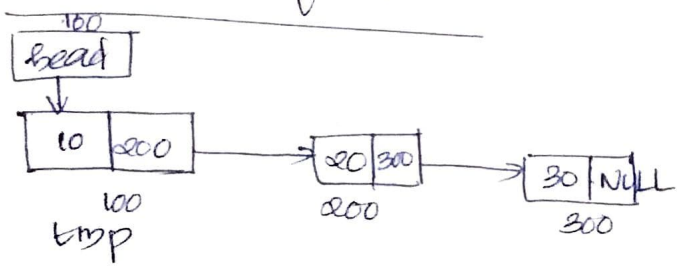
```
tmp -> data = x;
tmp -> next = current -> next;
current -> next = tmp;
```

Now the new node is inserted at specified position



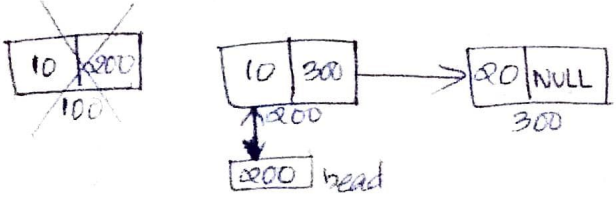

## Deletion

(i) Deletion at front

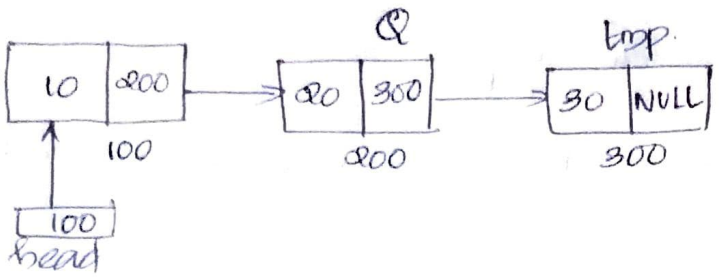



Inorder to delete the 1st node of a singly linked list, make the following changes

```
tmp = head;
head = head -> next;
free(tmp);
```



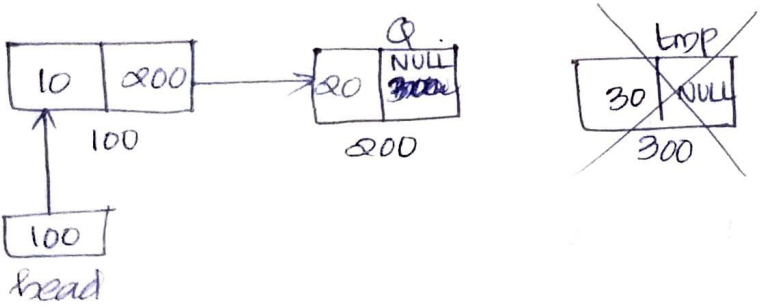
(ii) deletion at end



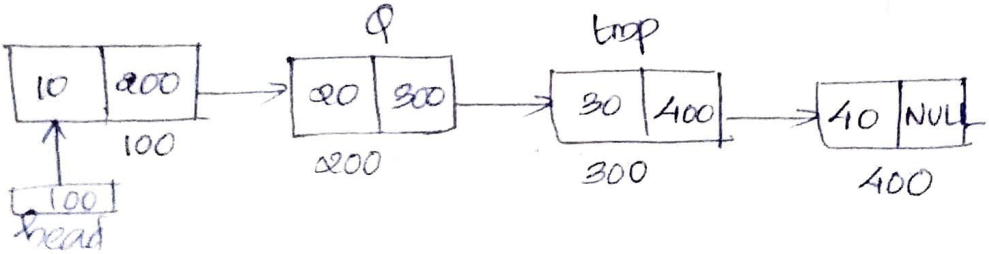

To delete the last node of the linked list, make the following changes

```
Q → link = NULL;
free(tmp)
```

Now the linkedlist will be



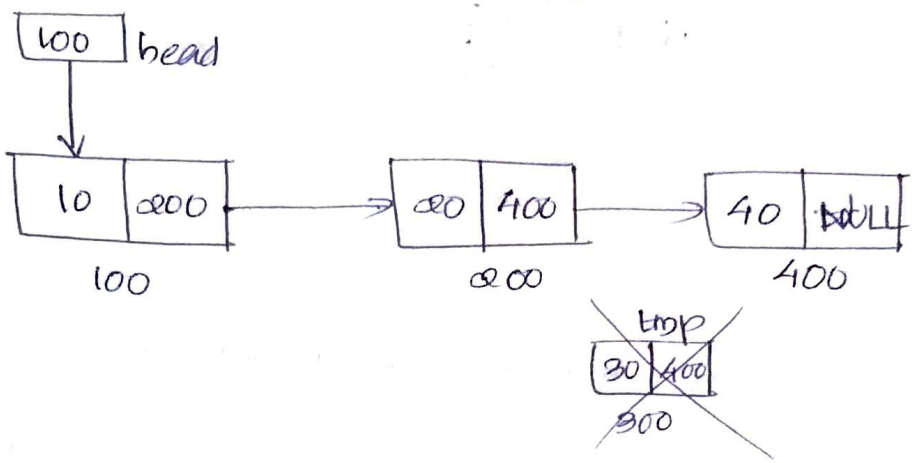

(iii) Deletion in between




To delete a spec node at a specified position (here delete tmp). Do the following steps.

```
tmp = q -> next;
q -> next = tmp -> next;
free(tmp);
```

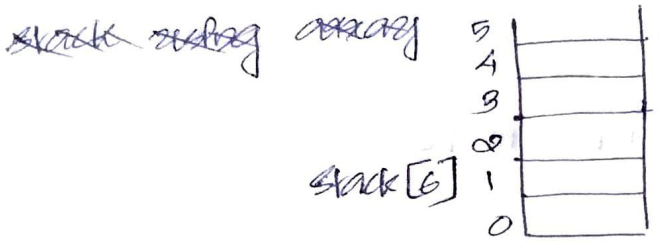
Now the linked list will be




1) (a) Stack

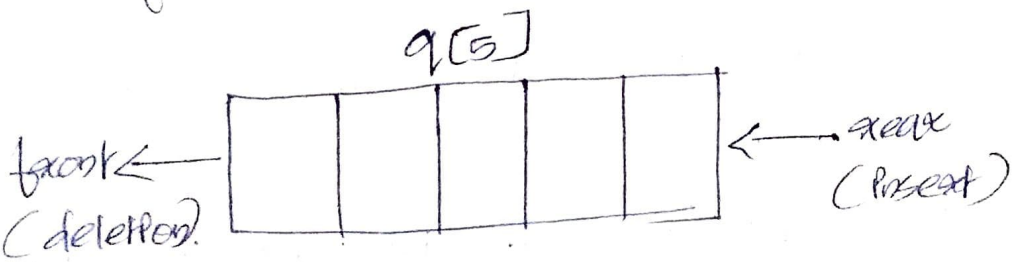
* Stack is a linear data structure.
* It works in a LIFO fashion [Last In First Out]
* Elements are inserted & deleted from the same end.
* Therefore, the last inserted element is deleted first.
* The process of ~~insertion~~ inserting elements into a stack is known as PUSH operation.
* The process of deleting elements from the stack is known as pop operation.

* top is the pointer that points to the top most or last inserted element of the stack.




## Queue

* Queue is a linear data structure.
* It works in a FIFO fashion [First In First Out]
* The first inserted element is deleted first.
* The elements are inserted into the queue through Rear end & are deleted from the queue through ~~the~~ front end.
* The process of inserting elements into the queue is called enqueue & process of deleting elements from the queue is known as dequeue.




## (c) bitstring

Any subset of the universal set that

is expressed in terms of 0's and 1's is called a bitstring.

Eg:- $U = \{1, 2, 3, 4, 5, 6, 7\}$.

$A = \{1, 3, 5\}$ and $B = \{2, 4, 6, 7\}$

Bitstring of A & B are

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

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

## Operations

### Union

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

$A = \{1, 3\}$

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

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

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

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

Bitstring $(A \cup B) = \{1, 1, 1, 1, 1\}$.

i.e.,

| A | B | A ∨ B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

## Intersection

~~From above example~~

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

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

Bitstring(A) = {1, 1, 1, 0, 0}

Bitstring(B) = {0, 0, 1, 1, 1}

$A \cap B = \{3\}$

Bitstring($A \cap B$) = {0, 0, 1, 0, 0}

i.e,

| A | B | $A \cap B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

## Complement

| A | $\bar{A}$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

$U = \{1, 2, 3\}$

$A = \{1, 2\}$ Bitstring(A) = {1, 1, 0}

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

Difference

U = {1, 2, 3, 4}

A = {1, 2, 3}, Bitstring(A) = {1, 1, 1, 0}

B = {2, 4}, Bitstring(B) = {0, 1, 0, 1}

A − B = A − A∩B

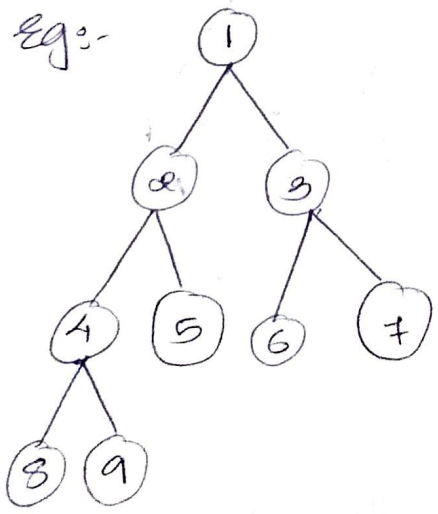
= {1, 3}

= {1, 0, 1, 0}

(b) Binary tree.

A binary tree is a tree data structure. It is a non-linear data structure in which each parent node can have almost 2 children.

Eg:-




Binary search tree

The binary search trees are also binary trees. They are designed for searching purpose.

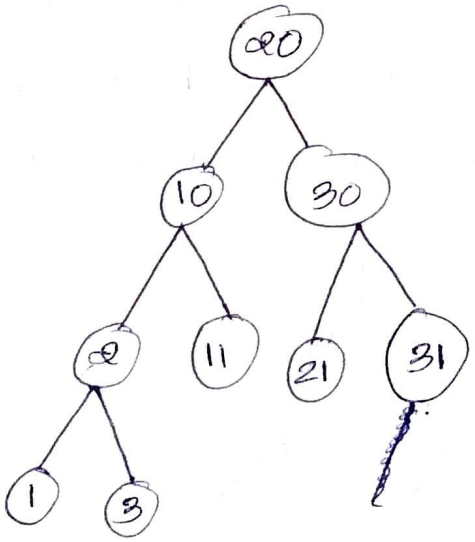
In addition to binary trees, binary search trees have a extra properties.

① All the keys in the left subtree of the root are always lesser than the keys in the ~~right subtree~~ root

② All the keys in the right subtree of the root are always greater than the key in the root.

③ The left subtree & right subtree also satisfy the above properties.

Eg:-




(d) Disjoint set means there is no common elements in both the set. The disjoint data structure maintains a collection of such disjoint sets.

Eg:- $S_x = \{1, 2, 3, 4, 5\}$
$S_y = \{10, 11\}$.

} These are disjoint sets

## operations

### (i) MAKE_SET(x)

This creates a new set whose only member is pointed to by x.

Eg: Make_set(x) → from 0 to 7

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

(ii) UNION(x, y) → This operation unites / merges 2 dynamic sets that contain x & y in new set

Eg:- $S_x = \{3, 4, 5\}$

$S_y = \{1, 4, 2\}$.

$S_{xy} = S_x \cup S_y = \{1, 2, 3, 4, 5\}$.

(iii) FindSet(x) → This operation returns a pointer to the unique set that containing the element x.

$S_x = \{1, 2, 3\}$

$S_y = \{5, 6, 4\}$

find(1) → belongs to $S_1$

find(4) → belongs to $S_2$

find(2) → belongs to $S_1$

Eg: Makeset(5) → from 1 to 5

{1} {2} {3} {4} {5}

Union-set(2, 3) → {1} {2, 3} {4} {5}

2 & 3 are connected

because find(2) == find(3)

both are in the same set

4) The process of converting keys into address is known as hashing. The hash function generates address from a given ~~key~~ key. There are several methods to ~~convert~~ generate address from keys.

methods

→ Truncation method

→ Folding method

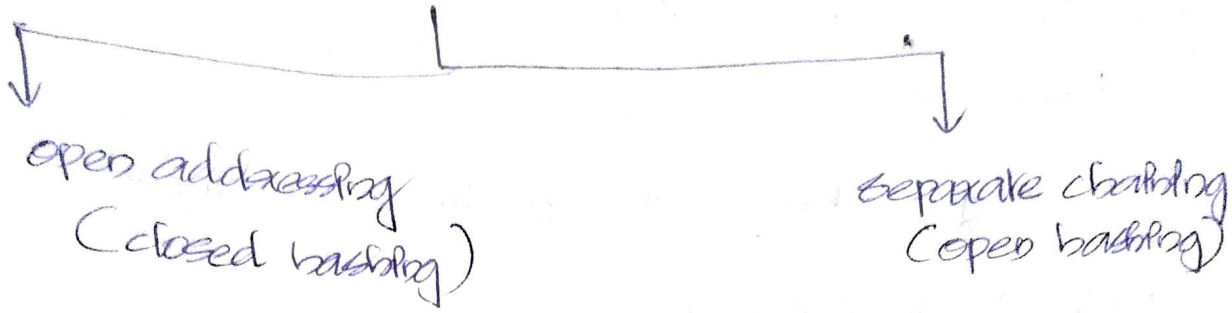
→ division method

→ mid square method

key → hash function → address.

If the hash function generates same address for different keys a situation called collision occurs. To resolve collision, there are several collision resolution techniques

collision resolution techniques




In this, a given key is compared with many different keys & each comparison is known as a probe.

## (i) open addressing

In this method, if a key causes collision, that key is placed in other location within the hash table.

3 methods

→ Linear probing
→ quadratic probing
→ Double hashing.

## → Linear probing

If the address generated for a key is 'm' and that location is not empty. then that key is inserted in next location 'm+1' & if it is also not empty, then insert that key in 'm+2' location. & so on.

Eg:- 10, 16, 13, 40, 35, 38, 28, 39, 60, 11

Table size = 11

h(key) = key % 11 // division method

- h(10) = 10 % 11 = 10
- h(16) = 16 % 11 = 5
- h(13) = 13 % 11 = 2
- h(40) = 40 % 11 = 7
- h(35) = 35 % 11 = 2

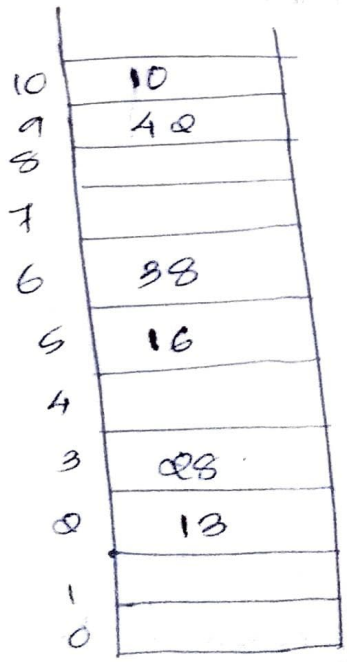
~~h(13) = 13 % 11~~

- h(28) = 28 % 11 = 6

~~h(39) = 39 % 11 = 6~~

h(60) = 60 % 11 = 10

h(7) = 7 % 11 = 7




If we insert 38, collision occurs. location 5 is not empty so next location is 5+1=6. 6 is empty so 38 is inserted there.

28 also causes collision. location 6 is not empty. So next location is 6+1=7. 7 is empty so 28 is inserted there.

## Quadratic probing

In quadratic probing, the colliding keys are placed away from the original collision point

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

Eg:- 56, 49, 32, 40, 16, 39, table size = 11

$h(56) = 56 \% 11 = 1$

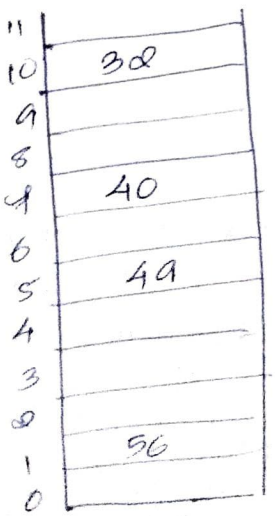
$h(49) = 49 \% 11 = 5$

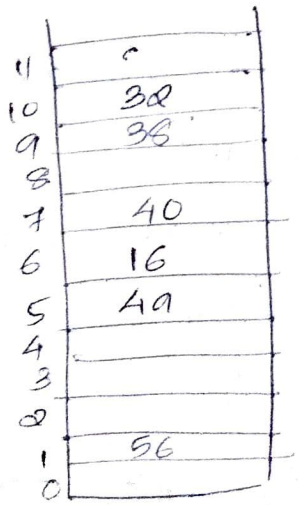
$h(32) = 32 \% 11 = 10$

$h(40) = 40 \% 11 = 7$

$h(16) = 16 \% 11 = 5$

$h(39) = 39 \% 11 = 5$




Insert 56, 49, 32, 40

When we want to insert 16. But location 5 is not empty so next location is $5+1^2=6$. 6 is empty so 16 is inserted there. next we

2 and so next 35. But location 5 is not empty. So next location is $5+1^2=6$. That location is also not empty. So next location is $5+2^2 = 5+4 = 9$, location 9 is empty. So 35 is inserted there.

→ double hashing

In double hashing, increment factor is not constant. The increment factor is another hash function. So named double hashing.

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

Eg: 56 49 32 ~~16~~ 35 10 34 ~~25~~

$h(56) = 56 \% 11 = 1$

$h(49) = 49 \% 11 = 5$

$h(32) = 32 \% 11 = 10$

~~$h(16) = 16 \% 11 = 5$ // collision~~

$h(35) = 35 \% 11 = 6$

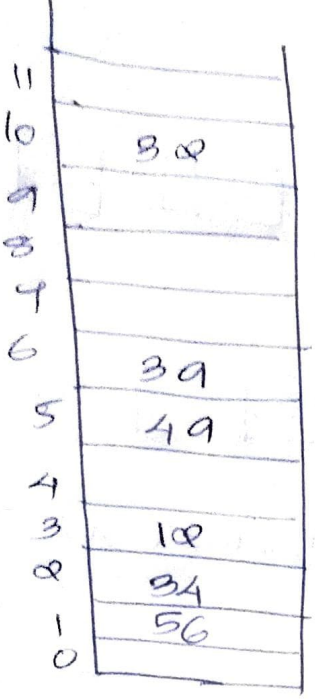
$h(10) = 10 \% 11 = 1$ // collision next location $= 1+(1-12\%7)$
$= 3 \% 11 = 3$.

$h(34) = 34 \% 11 = 1$ // collision ~~next location = 1+1=~~

~~$h(25) = 25 \% 11 = 3$.~~

~~16 → next location $= (5+7-(16\%7)) \% 11$~~
~~$= 5+(7-2) \% 11 = 5+5 = 10\%11$~~

next location
$34 = 1+7-(34\%7)$
$= 1+(7-6)$
$= (1+1) - 2 \% 11 = 2$




(ii) Separate chaining

In this method a linked list is maintained for elements with same hash address. These linked lists are referred to as chains & hence this method is known as separate chaining.

Eg:- 1035 0148 3249 1368 0348
1945 4063

table size = 5

h(1035) = 1035 % 5 = 0

h(0145) = 0148 % 5 = 3

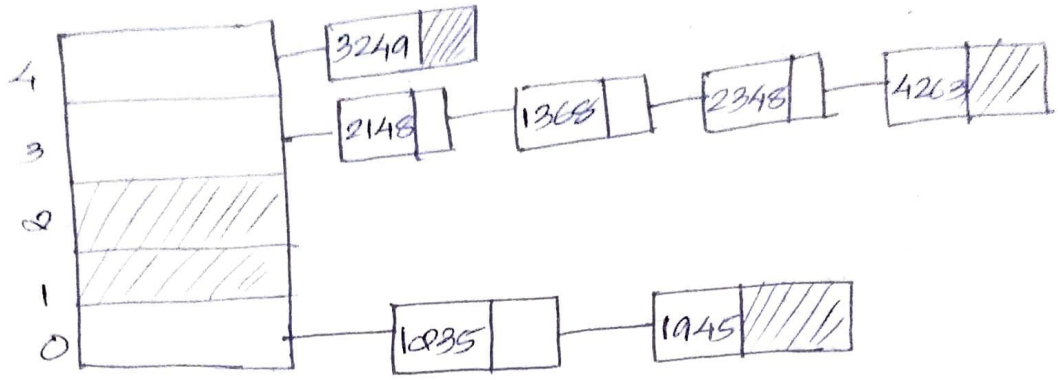
h(3249) = 3249 % 5 = 4

h(1365) = 1368 % 5 = 3

h(0345) = 2345 % 5 = 3

h(1945) = 1945 % 5 = 0

h(4063) = 4063 % 5 = 3.



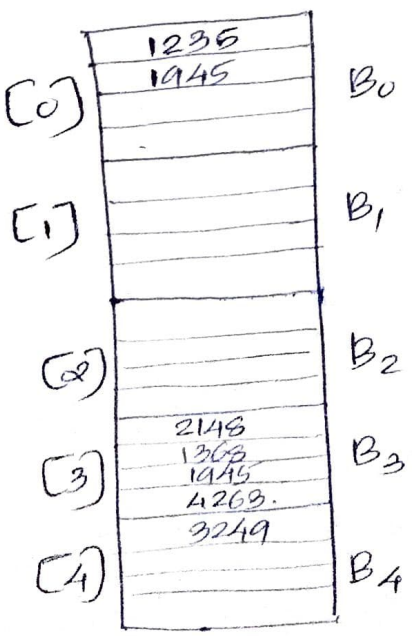

In separate chaining, load factor ($\lambda$) > 1.

## Bucket hashing

In this method, the hash table is made of buckets each bucket holds a number of records. This method doesn't resolve the collision completely.

Eg:- 1035, 0148, 3049, 1365, 0345, 1945, 4063.

[from above Eg:-]



3) Amortized analysis

Amortized analysis is the method of finding the amortized cost. It differs from average case analysis. It find out the amortized cost of each operation in the worst case.

Amortized cost = Average cost

3 techniques to find out the amortized cost
→ aggregate analysis
→ accounting method
→ potential method.

Potential method

This method is similar to accounting method. But instead of credit, this method uses potential energy. A potential energy is associated with each data structure operation.

The Amortized cost,

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

$C_i$ ↓ cost. $\phi(D_i) - \phi(D_{i-1})$ change in PE.

∴ change in potential = $\phi(D_i) - \phi(D_{i-1})$

current DS, previous DS

Eg:-

Stack (push operation)

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

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

The current size, $D_i$ = $n+1 - n$

= 1

Amortized cost = $C_i$ + change in potential

= 1 + 1 = 2

Pop operation

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

= $n - 1 - n$ = −1

Amortized cost = 1 + (−1) = 0

multipop.

Amortized cost of 1 operation $\hat{C}$ = O(1)

Amortized cost of n operation

= n * O(1) = O(n)

Eg (2) Incrementing a binary counter

| Row No | 8 | 4 | 2 | 1 | $\phi$ | $c_i$ | Amortized cost $c_i + \phi_i - \phi_{i-1}$ |
|---|---|---|---|---|---|---|---|
| 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 | $3 + (2-1) = 2$ |
| 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 | $1 + (4-3) = 2$ |

The amortized cost $\hat{c}_i = 2$

Amortized cost = $c_i + \phi(D_i) - \phi(D_{i-1})$

= bit changed from 0 to 1 +
bit changed from 1 to 0
+ bit changed from 0 to 1 −
bit changed from 1 to 0

= 2 # bit changed from 0 to 1

## Accounting method

In this method, we save some amount of rupees over some period of time and after a long time, this saved rupees are used to buy some other expensive things. These saved rupees are called credit.

Actual cost = No: of steps taken.

Amortized cost = Actual cost + credit

credit = Amortized cost - Actual cost.

Eg:- STACK

| operation | push | pop | multipop |
|---|---|---|---|
| Actual cost | 1 | 1 | min(n, k) |
| Amortized cost | 2 | 0 | 0 |

### push operation

credit = 2 - 1 = 1

No pop & multipop operation can be performed without any push operation

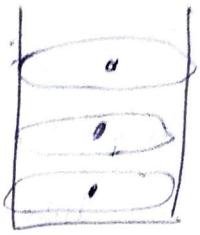
so amortized cost of pop & multipop are 0.

Final step

Push

2 coins are required to push a plate

$1 is the actual cost & $1 is the credit. The credit is placed on the top of the plate & $1 is used for push

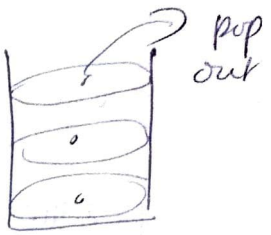


Pop

$1 on the top of the plate is used for pop operation.

So actual cost = $1

& amortized cost = $0




multipop.

This will remove $1 on the top of every plate

worst case cost of n operation = O(n).

(Eg 2) Implementing a binary counter

Actual cost of $i^{th}$ operation = $c_i$

amortized cost = $\hat{c}_i$

$$\text{Credit} = \sum_{i=1}^{n} \hat{C}_i - \sum_{i=1}^{n} C_i$$

When a bit is set, use $1 for actual setting of the bit & we place the other bit as credit & that is used for reset.