

## COLLISION RESOLUTION TECHNIQUES

If collision occurs then it should be handled by applying some techniques. Such a technique is called collision handling technique.

### OPEN HASHING

#### 1. Chaining

### CLOSE HASHING

1. Open addressing (linear probing)
2. Quadratic probing
3. Double hashing
4. Rehashing

## CHAINING

In collision handling method chaining is a concept which introduces an additional field with data i.e. chain. A separate chain table is maintained for colliding data. When collision occurs then a linked list(chain) is maintained at the home bucket.

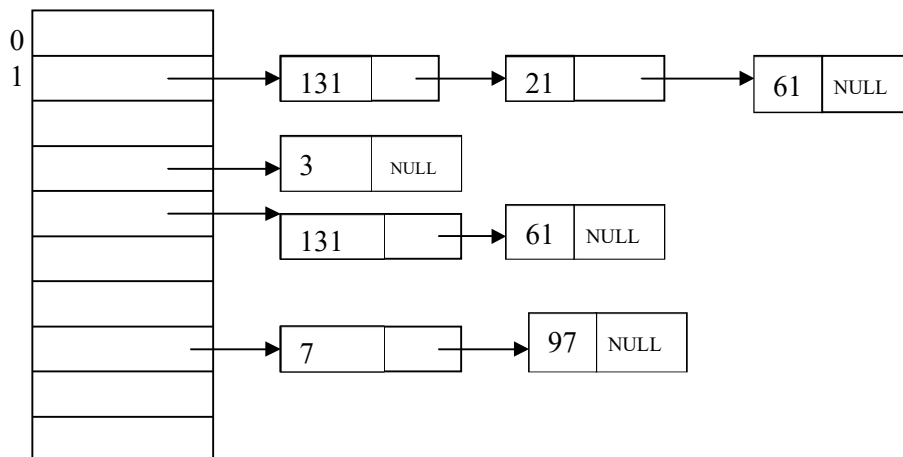
For eg;

Consider the keys to be placed in their home buckets are  
131, 3, 4, 21, 61, 7, 97, 8, 9

then we will apply a hash function as  $H(\text{key}) = \text{key} \% D$

Where D is the size of table. The hash table will be-

Here  $D = 10$



A chain is maintained for colliding elements. for instance 131 has a home bucket (key) 1. similarly key 21 and 61 demand for home bucket 1. Hence a chain is maintained at index 1.

## OPEN ADDRESSING – LINEAR PROBING

This is the easiest method of handling collision. When collision occurs i.e. when two records demand for the same home bucket in the hash table then collision can be solved by placing the second record linearly down whenever the empty bucket is found. When use linear probing (open addressing), the hash table is represented as a one-dimensional array with indices that range from 0 to the desired table size-1. Before inserting any elements into this table, we must initialize the table to represent the situation where all slots are empty. This allows us to detect overflows and collisions when we inset elements into the table. Then using some suitable hash function the element can be inserted into the hash table.

For example:

Consider that following keys are to be inserted in the hash table

131, 4, 8, 7, 21, 5, 31, 61, 9, 29

Initially, we will put the following keys in the hash table.

We will use Division hash function. That means the keys are placed using the formula

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

$$H(\text{key}) = \text{key} \% 10$$

For instance the element 131 can be placed at

$$\begin{aligned} H(\text{key}) &= 131 \% 10 \\ &= 1 \end{aligned}$$

Index 1 will be the home bucket for 131. Continuing in this fashion we will place 4, 8, 7.

Now the next key to be inserted is 21. According to the hash function


$$H(\text{key}) = 21 \% 10$$

$$H(\text{key}) = 1$$

But the index 1 location is already occupied by 131 i.e. collision occurs. To resolve this collision we will linearly move down and at the next empty location we will prob the element. Therefore 21 will be placed at the index 2. If the next element is 5 then we get the home bucket for 5 as index 5 and this bucket is empty so we will put the element 5 at index 5.

Index	Key	Key	Key
0	NULL	NULL	NULL
1	131	131	131
2	NULL	21	21
3	NULL	NULL	31
4	4	4	4
5	NULL	5	5
6	NULL	NULL	61
7	7	7	7
8	8	8	8
9	NULL	NULL	NULL

after placing keys 31, 61



The next record key is 9. According to decision hash function it demands for the home bucket 9. Hence we will place 9 at index 9. Now the next final record key 29 and it hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs. To handle it we move back to bucket 0 and the location over there is empty 29 will be placed at 0<sup>th</sup> index.

**Problem with linear probing:**

One major problem with linear probing is primary clustering. Primary clustering is a process in which a block of data is formed in the hash table when collision is resolved.

19%10 = 9  
18%10 = 8  
39%10 = 9  
29%10 = 9  
8%10 = 8

cluster is formed

rest of the table is empty

this cluster problem can be solved by quadratic probing.

Key
39
29
8
18
19

**QUADRATIC PROBING:**

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula.

$$H(\text{key}) = (\text{Hash}(\text{key}) + i^2) \% m$$

where m can be table size or any prime number.

for eg; If we have to insert following elements in the hash table with table size 10:

37, 90, 55, 22, 17, 49, 87

37 % 10 = 7  
90 % 10 = 0  
55 % 10 = 5  
22 % 10 = 2  
11 % 10 = 1

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	
9	

Now if we want to place 17 a collision will occur as 17%10 = 7 and bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

$$H_i(\text{key}) = (\text{Hash}(\text{key}) + i^2) \% m$$

Consider i = 0 then  
(17 + 0<sup>2</sup>) % 10 = 7

$$(17 + 1^2) \% 10 = 8, \text{ when } i=1$$

The bucket 8 is empty hence we will place the element at index 8.  
Then comes 49 which will be placed at index 9.

$$49 \% 10 = 9$$

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	49
9	

Now to place 87 we will use quadratic probing.

$$(87 + 0) \% 10 = 7$$

$$(87 + 1) \% 10 = 8... \text{ but already occupied}$$

$$(87 + 2^2) \% 10 = 1.. \text{ already occupied}$$

$$(87 + 3^2) \% 10 = 6$$

It is observed that if we want place all the necessary elements in the hash table the size of divisor (m) should be twice as large as total number of elements.

0	90
1	11
2	22
3	
4	
5	
6	55
7	87
8	37
9	49

## DOUBLE HASHING

Double hashing is technique in which a second hash function is applied to the key when a collision occurs. By applying the second hash function we will get the number of positions from the point of collision to insert.

There are two important rules to be followed for the second function:

- it must never evaluate to zero.
- must make sure that all cells can be probed.

The formula to be used for double hashing is

$$H_1(\text{key}) = \text{key mod tablesize}$$

$$H_2(\text{key}) = M - (\text{key mod } M)$$

where M is a prime number smaller than the size of the table.

Consider the following elements to be placed in the hash table of size 10

37, 90, 45, 22, 17, 49, 55

Initially insert the elements using the formula for  $H_1(\text{key})$ .

Insert 37, 90, 45, 22

$$H_1(37) = 37 \% 10 = 7$$

$$H_1(90) = 90 \% 10 = 0$$

$$H_1(45) = 45 \% 10 = 5$$

$$H_1(22) = 22 \% 10 = 2$$

$$H_1(49) = 49 \% 10 = 9$$

Key
90
22
45
37
49

Now if 17 to be inserted then

$$H_1(17) = 17 \% 10 = 7$$

$$H_2(\text{key}) = M - (\text{key} \% M)$$

Here M is prime number smaller than the size of the table. Prime number smaller than table size 10 is 7

Hence  $M = 7$

$$\begin{aligned} H_2(17) &= 7 - (17 \% 7) \\ &= 7 - 3 = 4 \end{aligned}$$

That means we have to insert the element 17 at 4 places from 37. In short we have 4 jumps. Therefore the 17 will be placed at index 1.

Now to insert number 55

$$H_1(55) = 55 \% 10 = 5 \rightarrow \text{Collision}$$

$$\begin{aligned} H_2(55) &= 7 - (55 \% 7) \\ &= 7 - 6 = 1 \end{aligned}$$

That means we have to take one jump from index 5 to place 55. Finally the hash table will be -

Key
90
17
22
45
37
49

Key
90
17
22
45
55
37
49

### Comparison of Quadratic Probing & Double Hashing

The double hashing requires another hash function whose probing efficiency is same as some another hash function required when handling random collision.

The double hashing is more complex to implement than quadratic probing. The quadratic probing is fast technique than double hashing.

### REHASHING

Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating a new table. It is preferable if the total size of table is a prime number. There are situations in which the rehashing is required.

- When table is completely full
- With quadratic probing when the table is filled half.
- When insertions fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by re computing their positions using hash functions.

Consider we have to insert the elements 37, 90, 55, 22, 17, 49, and 87. the table size is 10 and will use hash function.,

$$H(\text{key}) = \text{key mod tablesize}$$

$$37 \% 10 = 7$$

$$90 \% 10 = 0$$

$$55 \% 10 = 5$$

$$22 \% 10 = 2$$

$$17 \% 10 = 7 \text{ Collision solved by linear probing}$$

$$49 \% 10 = 9$$

Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size. The old table size is 10 then we should double this size for new table, that becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23. And new hash function will be

$$H(\text{key}) = \text{key mod } 23$$

$$37 \% 23 = 14$$

$$90 \% 23 = 21$$

$$55 \% 23 = 9$$

$$22 \% 23 = 22$$

$$17 \% 23 = 17$$

$$49 \% 23 = 3$$

$$87 \% 23 = 18$$

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	49
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	

Now the hash table is sufficiently large to accommodate new insertions.

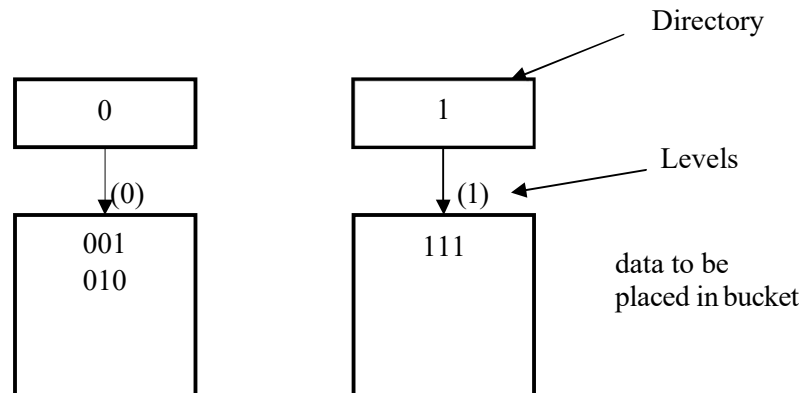
**Advantages:**

1. This technique provides the programmer a flexibility to enlarge the table size if required.
2. Only the space gets doubled with simple hash function which avoids occurrence of collisions.

**EXTENSIBLE HASHING**

- Extensible hashing is a technique which handles a large amount of data. The data to be placed in the hash table is by extracting certain number of bits.
- Extensible hashing grow and shrink similar to B-trees.
- In extensible hashing referring the size of directory the elements are to be placed in buckets. The levels are indicated in parenthesis.

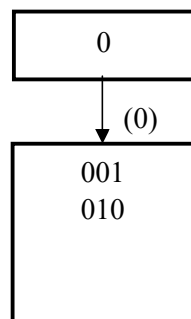
For eg:



- The bucket can hold the data of its global depth. If data in bucket is more than global depth then, split the bucket and double the directory.

Consider we have to insert 1, 4, 5, 7, 8, 10. Assume each page can hold 2 data entries (2 is the depth).

**Step 1:** Insert 1, 4



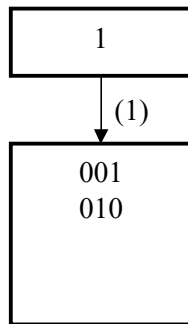
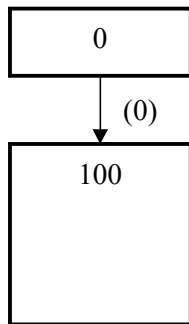
1 = 001

4 = 100

We will examine last bit of data and insert the data in bucket.

Insert 5. The bucket is full. Hence double the directory.





1 = 001

4 = 100

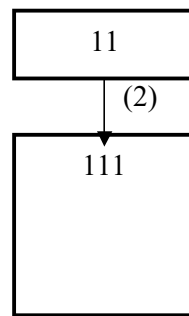
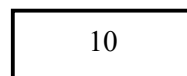
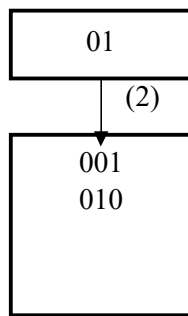
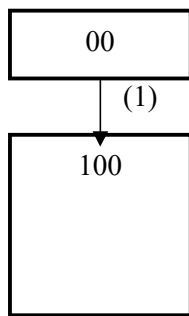
5 = 101

Based on last bit the data is inserted.

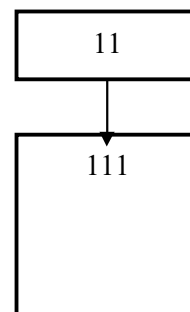
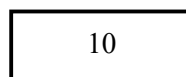
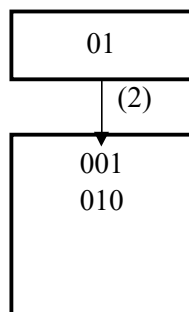
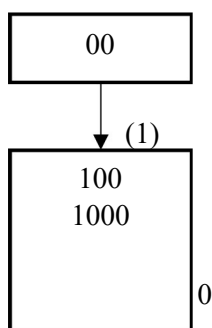
**Step 2: Insert 7**

7 = 111

But as depth is full we can not insert 7 here. Then double the directory and split the bucket. After insertion of 7. Now consider last two bits.



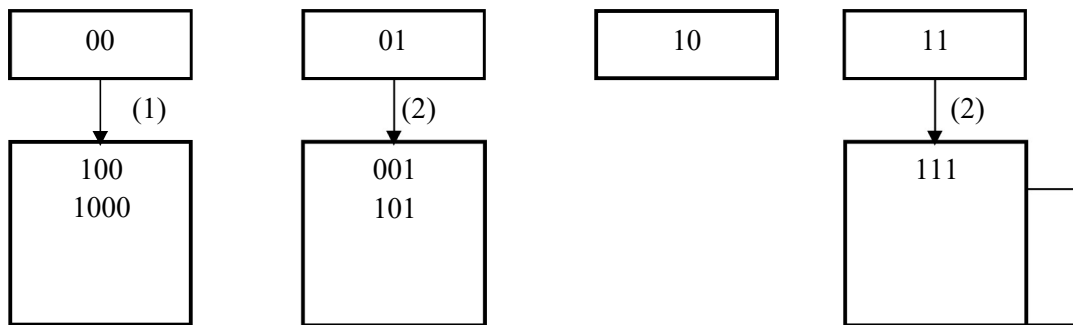
**Step 3: Insert 8 i.e. 1000**



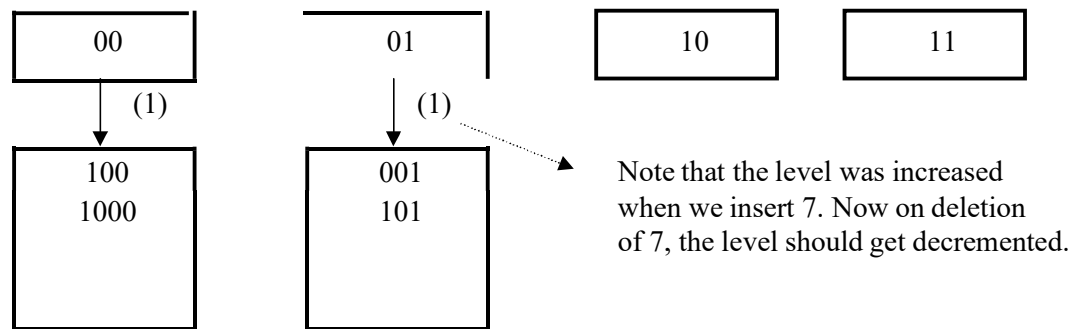
Thus the data is inserted using extensible hashing.

### Deletion Operation:

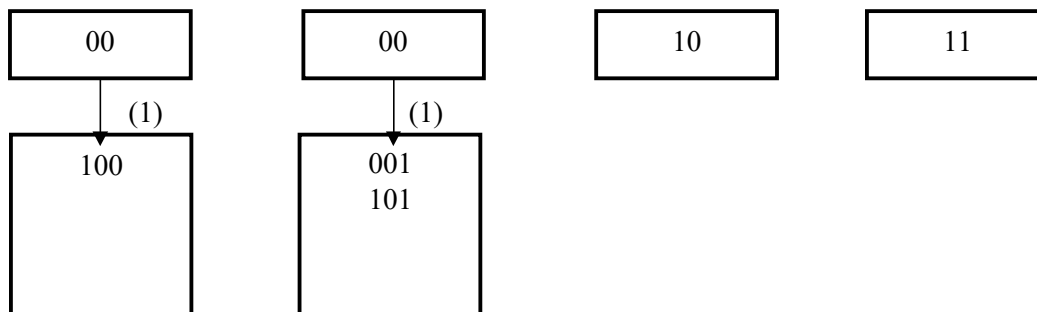
If we want to delete 10 then, simply make the bucket of 10 empty.



Delete 7.



Delete 8. Remove entry from directory 00.



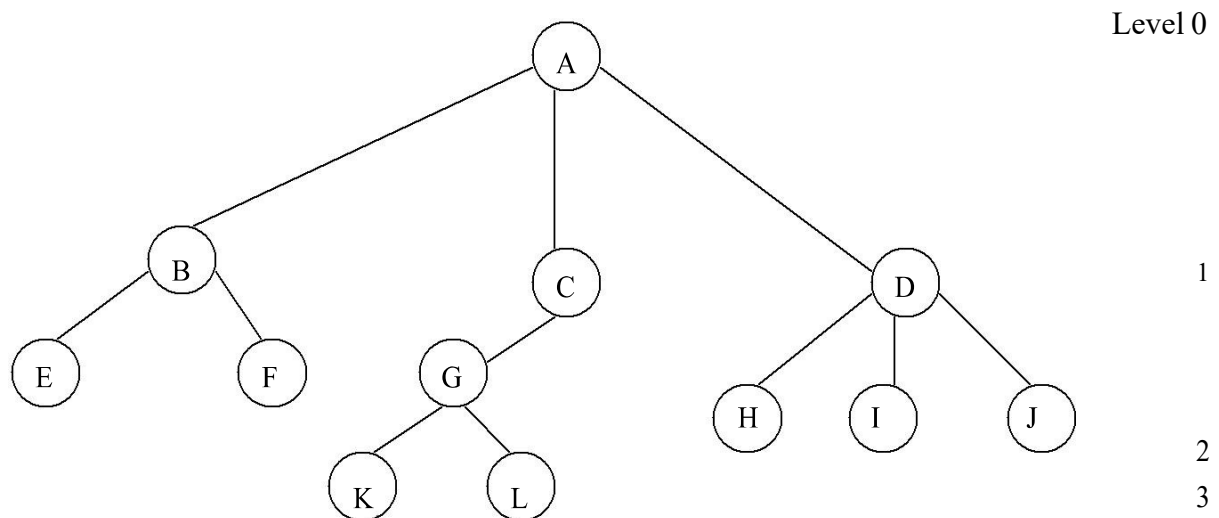
**Applications of hashing:**

1. In compilers to keep track of declared variables.
2. For online spelling checking the hashing functions are used.
3. Hashing helps in Game playing programs to store the moves made.
4. For browser program while caching the web pages, hashing is used.
5. Construct a *message authentication code* (MAC)
6. Digital signature.
7. Time stamping
8. Key updating: key is hashed at specific intervals resulting in new key

**Binary Search Trees:** Various Binary tree representation, definition, BST ADT, Implementation, Operations- Searching, Insertion and Deletion, Binary tree traversals, threaded binary trees,  
**AVL Trees :** Definition, Height of an AVL Tree, Operations – Insertion, Deletion and Searching  
**B-Trees:** B-Tree of order m, height of a B-Tree, insertion, deletion and searching, B+ Tree.

## TREES

A Tree is a data structure in which each element is attached to one or more elements directly beneath it.



## Terminology

- The connections between elements are called **branches**.
- A tree has a single root, called **root** node, which is shown at the top of the tree. i.e. root is always at the highest level 0.
- Each node has exactly one node above it, called **parent**. Eg: A is the parent of B,C and D.
- The nodes just below a node are called its **children**. ie. child nodes are one level lower than the parent node.
- A node which does not have any child called **leaf or terminal node**. Eg: E, F, K, L, H, I and M are leaf
- Nodes with at least one child are called **non terminal or internal nodes**.
- The child nodes of same parent are said to be **siblings**.
- A **path** in a tree is a list of distinct nodes in which successive nodes are connected by branches in the tree.
- The **length** of a particular path is the number of branches in that path. The **degree** of a node of a tree is the number of children of that node.
- The maximum number of children a node can have is often referred to as the **order** of a tree. The **height or depth** of a tree is the length of the longest path from root to any leaf.

1. **Root:** This is the unique node in the tree to which further sub trees are attached. Eg: A

**Degree of the node:** The total number of sub-trees attached to the node is called the degree of the node. Eg: For node A degree is 3. For node K degree is 0

3. **Leaves:** These are the terminal nodes of the tree. The nodes with degree 0 are always the leaf nodes.

Eg: E, F, K, L, H, I, J

4. Internal nodes: The nodes other than the root node and the leaves are called the internal nodes. Eg: B, C, D, G
5. Parent nodes: The node which is having further sub-trees(branches) is called the parent node of those sub-trees. Eg: B is the parent node of E and F.
6. Predecessor: While displaying the tree, if some particular node occurs previous to some other node then that node is called the predecessor of the other node. Eg: E is the predecessor of the node B.
7. Successor: The node which occurs next to some other node is a successor node. Eg: B is the successor of E and F.
8. Level of the tree: The root node is always considered at level 0, then its adjacent children are supposed to be at level 1 and so on. Eg: A is at level 0, B,C,D are at level 1, E,F,G,H,I,J are at level 2, K,L are at level 3.
9. Height of the tree: The maximum level is the height of the tree. Here height of the tree is 3. The height if the tree is also called depth of the tree.
10. Degree of tree: The maximum degree of the node is called the degree of the tree.

## BINARY TREES

Binary tree is a tree in which each node has at most two children, a left child and a right child. Thus the order of binary tree is 2.

A binary tree is either empty or consists of  
a) a node called the root  
b) left and right sub trees are themselves binary trees.

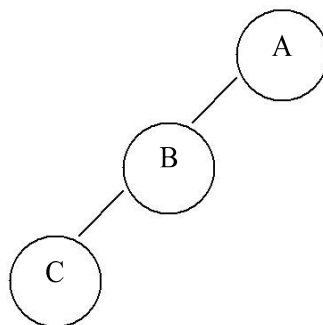
**A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint trees called left sub-tree and right sub-tree.**

In binary tree each node will have one data field and two pointer fields for representing the sub-branches. The degree of each node in the binary tree will be at the most two.

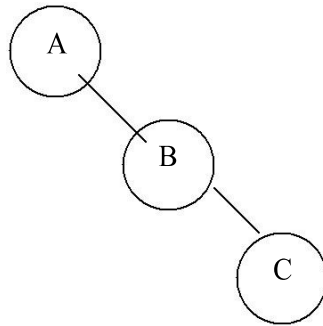
### Types Of Binary Trees:

There are 3 types of binary trees:

1. **Left skewed binary tree:** If the right sub-tree is missing in every node of a tree we call it as left skewed tree.

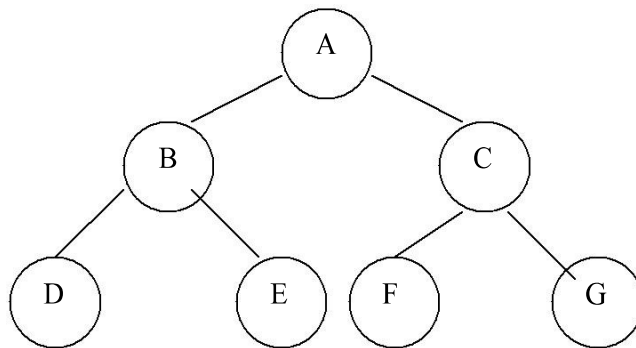


2. **Right skewed binary tree:** If the left sub-tree is missing in every node of a tree we call it is right sub-tree.



3. **Complete binary tree:**

The tree in which degree of each node is at the most two is called a complete binary tree. In a complete binary tree there is exactly one node at level 0, two nodes at level 1 and four nodes at level 2 and so on. So we can say that a complete binary tree depth  $d$  will contain exactly  $2^l$  nodes at each level  $l$ , where  $l$  is from 0 to  $d$ .



**Note:**

1. A binary tree of depth  $n$  will have maximum  $2^n - 1$  nodes.
2. A complete binary tree of level  $l$  will have maximum  $2^l$  nodes at each level, where  $l$  starts from 0.
3. Any binary tree with  $n$  nodes will have at the most  $n+1$  null branches.
4. The total number of edges in a complete binary tree with  $n$  terminal nodes are  $2(n-1)$ .

## **Binary Tree Representation**

A binary tree can be represented mainly in 2 ways:

- a) Sequential Representation
- b) Linked Representation

### **a) Sequential Representation**

The simplest way to represent binary trees in memory is the sequential representation that uses one-dimensional array.

- 1) The root of binary tree is stored in the 1<sup>st</sup> location of array
- 2) If a node is in the  $j^{\text{th}}$  location of array, then its left child is in the location  $2J+1$  and its right child in the location  $2J+2$

The maximum size that is required for an array to store a tree is  $2^{d+1} - 1$ , where  $d$  is the depth of the tree.