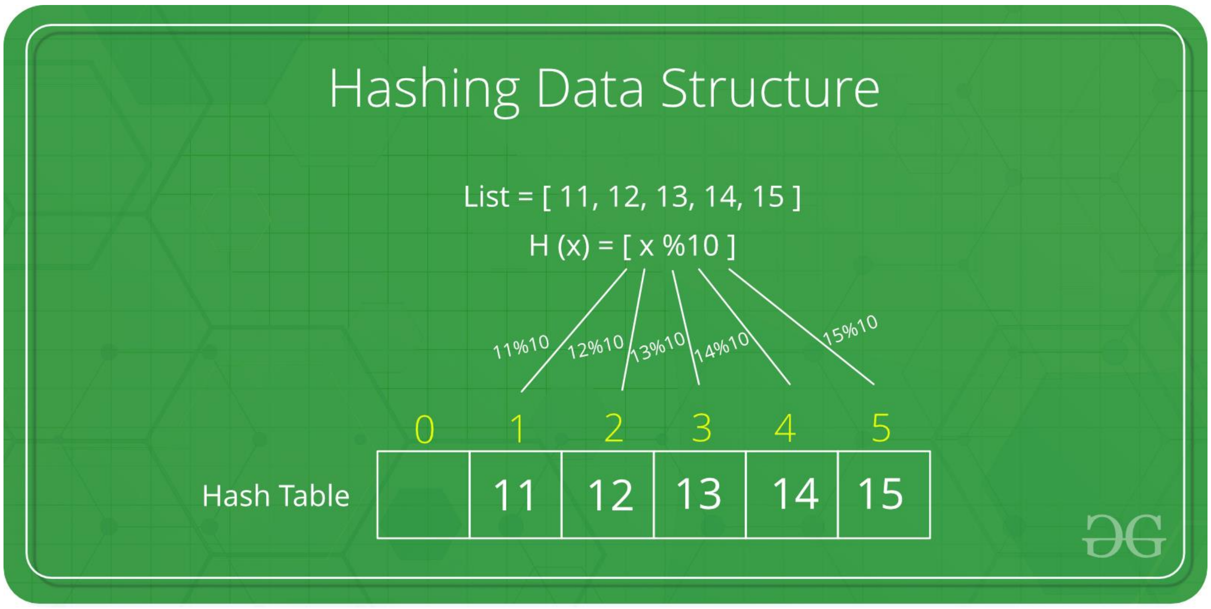
Hashing

Hashing is a technique or process of mapping keys, and values into the hash table by using a **hash function**. **It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.**

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



Hashing Data Structure

A **Hash Function** is a function that converts a given numeric or alphanumeric key to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function **maps** a significant number or string to a small integer that can be used as the **index** in the hash table.

The pair is of the form **(key, value)**, where for a given key, one can find a value using some kind of a "function" that maps keys to values. **The key for a given object can be calculated using a function called a hash function.** For example, given an array A, if i is the key, then we can find the value by simply looking up A[i].

Types of Hash Function:

There are many hash functions that use numeric or alphanumeric keys.

1.Division Method.
2.Mid Square Method.
3.Folding Method.
4.Multiplication Method.

1. Division Method:

This is the most simple and easiest method to generate a hash value. The hash function divides the value k by M and then uses the remainder obtained.

Formula:
h(K) = k mod M
Here,
***k** is the key value, and*
***M** is the size of the hash table.*

It is best suited that **M** is a prime number as that can make sure the keys are more uniformly distributed. The hash function is dependent upon the remainder of a division.

Example:
k = 12345
M = 95
h(12345) = 12345 mod 95
= 90

k = 1276
M = 11
h(1276) = 1276 mod 11
= 0

Pros:

1. This method is quite good for any value of M.
2. The division method is very fast since it requires only a single division operation.

Cons:

1. This method leads to poor performance since consecutive keys map to consecutive hash values in the hash table.
2. Sometimes extra care should be taken to choose the value of M.

2. Mid Square Method:

The mid-square method is a very good hashing method. It involves two steps to compute the hash value-

1. Square the value of the key k i.e. k^2
2. Extract the middle **r** digits as the hash value.

Formula:

h(K) = h(k x k)

Here,

***k** is the key value.*

The value of **r** can be decided based on the size of the table.

Example:

Suppose the hash table has 100 memory locations. So r = 2 because two digits are required to map the key to the memory location.

k = 60
k x k = 60 x 60
= 3600
h(60) = 60

The hash value obtained is 60

Pros:

1. The performance of this method is good as most or all digits of the key value contribute to the result. This is because all digits in the key contribute to generating the middle digits of the squared result.
2. The result is not dominated by the distribution of the top digit or bottom digit of the original key value.

Cons:

1. The size of the key is one of the limitations of this method, as the key is of big size then its square will double the number of digits.
2. Another disadvantage is that there will be collisions but we can try to reduce collisions.

3. Digit Folding Method:

This method involves two steps:

1. Divide the key-value **k** into a number of parts i.e. **k1, k2, k3,....,kn**, where each part has the same number of digits except for the last part that can have lesser digits than the other parts.

2. Add the individual parts. The hash value is obtained by ignoring the last carry if any.

Formula:
k = k1, k2, k3, k4, ….., kn
s = k1+ k2 + k3 + k4 +….+ kn
h(K)= s
Here,
s** is obtained by adding the parts of the key **k

Example:
k = 12345
k1 = 12, k2 = 34, k3 = 5
s = k1 + k2 + k3
= 12 + 34 + 5
= 51
h(K) = 51

Note:
The number of digits in each part varies depending upon the size of the hash table. Suppose for example the size of the hash table is 100, then each part must have two digits except for the last part which can have a lesser number of digits.

<u>**4. Multiplication Method**</u>
This method involves the following steps:

1. Choose a constant value A such that 0 < A < 1.
2. Multiply the key value with A.
3. Extract the fractional part of kA.
4. Multiply the result of the above step by the size of the hash table i.e. M.
5. The resulting hash value is obtained by taking the floor of the result obtained in step 4.

Formula:
h(K) = floor (M (kA mod 1))
Here,
***M** is the size of the hash table.*
***k** is the key value.*
***A** is a constant value.*

Example:

k = 12345
A = 0.357840
M = 100

*h(12345) = floor[100 (12345*0.357840 mod 1)]*
= floor[100 (4417.5348 mod 1)]
= floor[100 (0.5348)]
= floor[53.48]
= 53

Pros:
The advantage of the multiplication method is that it can work with any value between 0 and 1, although there are some values that tend to give better results than the rest.

Cons:
The multiplication method is generally suitable when the table size is the power of two, then the whole process of computing the index by the key using multiplication hashing is very fast.

Types of Hashing:

1.Open Hashing or closed Addressing-

i.Separate Chaining

2.Closed Hashing or Open Addresssing-

i.Linear Probing

ii.Quadratic Probing

iii.Double Hasing

Separate Chaining:

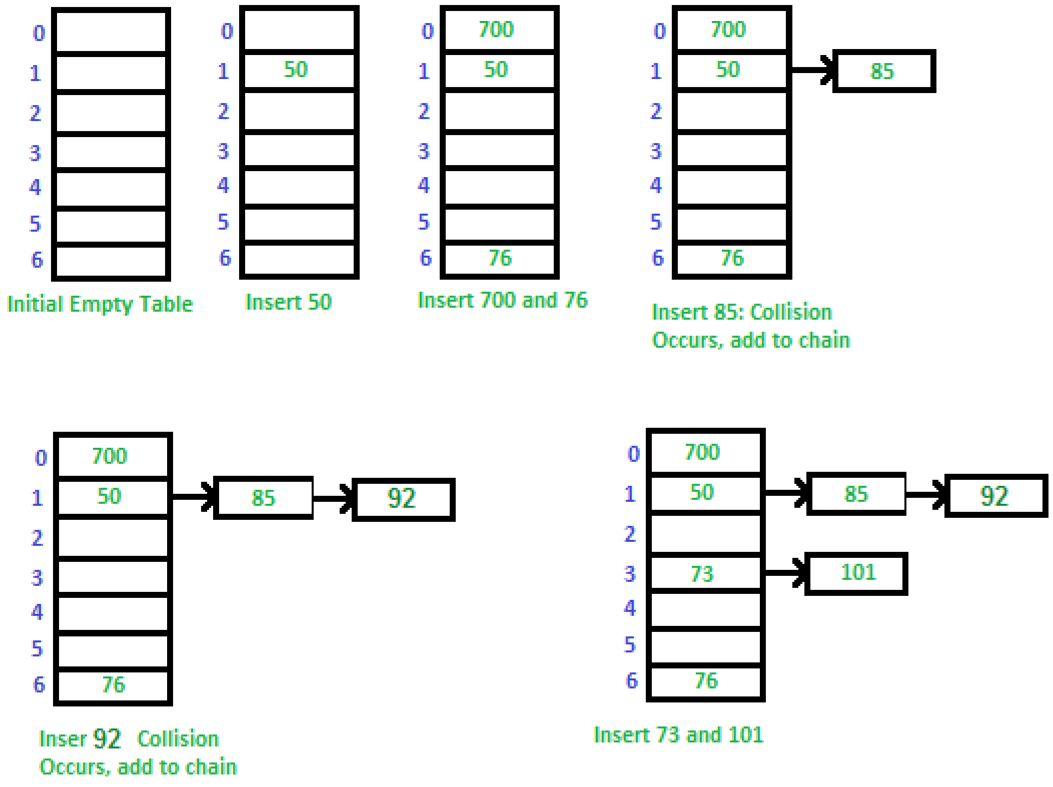
The idea behind separate chaining is to implement the array as a linked list called a chain. Separate chaining is one of the most popular and commonly used techniques in order to handle collisions.

*The **linked list** data structure is used to implement this technique.when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.*

Here, all those elements that hash into the same slot index are inserted into a linked list. Now, we can use a key K to search in the linked list by just linearly traversing. If the intrinsic key for any entry is equal to K then it means that we have found our entry. If we have reached the end of the linked list and yet we haven't found our entry then it means that the entry does not exist.

Hence,in separate chaining, if two different elements have the same hash value then we store both the elements in the same linked list one after the other.

Example: Let us consider a simple hash function as "**key mod 7**" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101



Initial Empty Table
Insert 50
Insert 700 and 76
Insert 85: Collision Occurs, add to chain
Inser 92 Collision Occurs, add to chain
Insert 73 and 101
700
50
85
92
73
101
76

Advantages:

- Simple to implement.
- Hash table never fills up, we can always add more elements to the chain.
- Less sensitive to the hash function or load factors.
- It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- The cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
- Wastage of Space (Some Parts of the hash table are never used)
- If the chain becomes long, then search time can become O(n) in the worst case
- Uses extra space for links

Performance of Chaining:

Performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing).

m = Number of slots in hash table
n = Number of keys to be inserted in hash table
Load factor α = n/m
Expected time to search = O(1 + α)
Expected time to delete = O(1 + α)
Time to insert = O(1)
Time complexity of search insert and delete is O(1) if α is O(1)

Open Addressing:

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the **hash table** itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed). This approach is also known as **closed hashing.** This entire procedure is based upon probing.

We will understand the types of probing ahead:

- ***Insert(k):*** *Keep probing until an empty slot is found. Once an empty slot is found, insert k.*
- ***Search(k):*** *Keep probing until the slot's key doesn't become equal to k or an empty slot is reached.*
- ***Delete(k)****:* ***Delete operation is interesting****. If we simply delete a key, then the search may fail. So slots of deleted keys are marked speciallyas"deleted".*
  The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Different ways of Open Addressing:

1. Linear Probing:

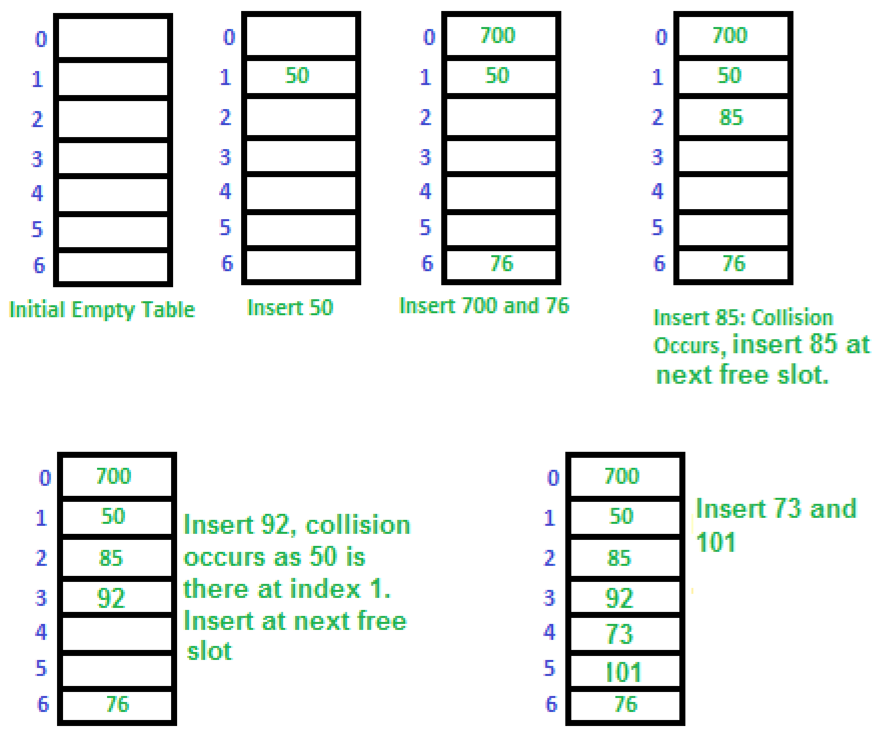
In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

The function used for rehashing is as follows: rehash(key) = (n+1)%table-size.

For example, The typical gap between two probes is 1 as seen in the example below:
Let ***hash(x)*** *be the slot index computed using a hash function and* ***S*** *be the table size*
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S
…………………………………………
…………………………………………

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101,

which means hash(key)= key% S, here S=size of the table =7,indexed from 0 to 6.We can define the hash function as per our choice if we want to create a hash table,although it is fixed internally with a pre-defined formula.


Initial Empty Table
Insert 50
Insert 700 and 76
Insert 85: Collision Occurs, insert 85 at next free slot.
Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot
Insert 73 and 101

Applications of linear probing:

Linear probing is a collision handling technique used in hashing, where the algorithm looks for the next available slot in the hash table to store the collided key. Some of the applications of linear probing include:

- **Symbol tables**: Linear probing is commonly used in symbol tables, which are used in compilers and interpreters to store variables and their associated values. Since symbol tables can grow dynamically, linear probing can be used to handle collisions and ensure that variables are stored efficiently.
- **Caching**: Linear probing can be used in caching systems to store frequently accessed data in memory. When a cache miss occurs, the data can be loaded into the cache using linear probing, and when a collision occurs, the next available slot in the cache can be used to store the data.
- **Databases**: Linear probing can be used in databases to store records and their associated keys. When a collision occurs, linear probing can be used to find the next available slot to store the record.
- **Compiler design**: Linear probing can be used in compiler design to implement symbol tables, error recovery mechanisms, and syntax analysis.
- **Spell checking:** Linear probing can be used in spell-checking software to store the dictionary of words and their associated

frequency counts. When a collision occurs, linear probing can be used to store the word in the next available slot.

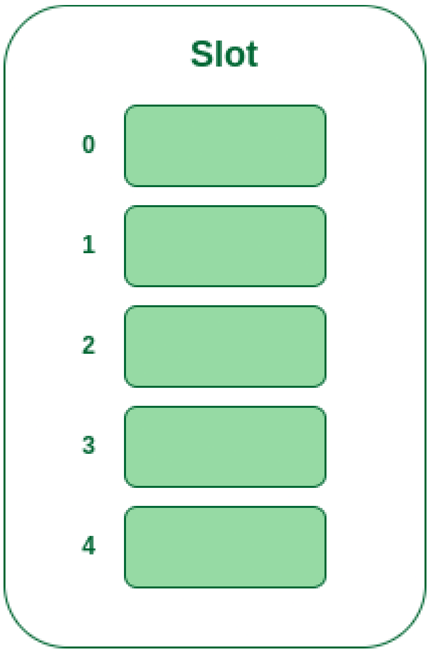
Overall, linear probing is a simple and efficient method for handling collisions in hash tables, and it can be used in a variety of applications that require efficient storage and retrieval of data.

Challenges in Linear Probing :

- **Primary Clustering:** One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.
- **Secondary Clustering:** Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

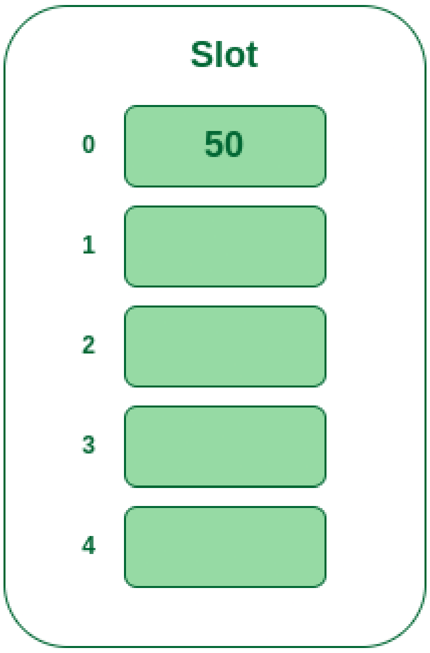
Example: Let us consider a simple hash function as "key mod 5" and a sequence of keys that are to be inserted are 50, 70, 76, 93.

- **Step1:** First draw the empty hash table which will have a possible range of hash values from 0 to 4 according to the hash function provided.



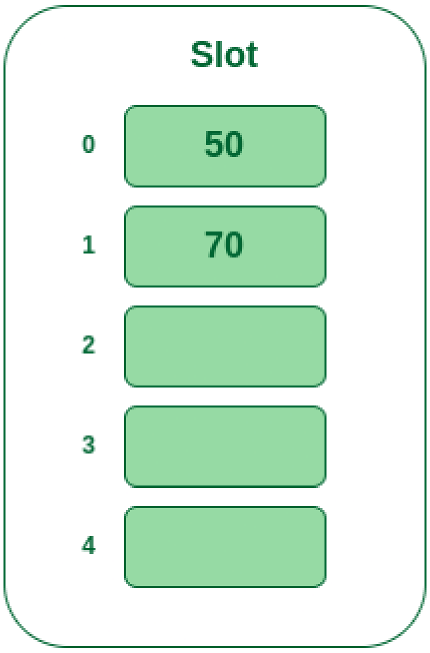
Hash table

- **Step 2:** Now insert all the keys in the hash table one by one. The first key is 50. It will map to slot number 0 because 50%5=0. So insert it into slot number 0.



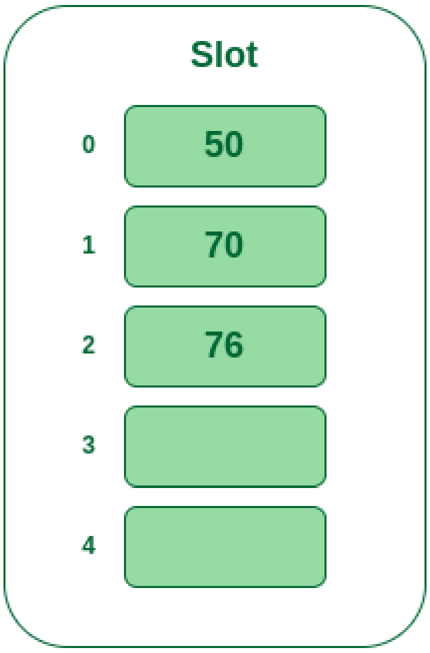
Insert 50 into hash table

- **Step 3:** The next key is 70. It will map to slot number 0 because 70%5=0 but 50 is already at slot number 0 so, search for the next empty slot and insert it.



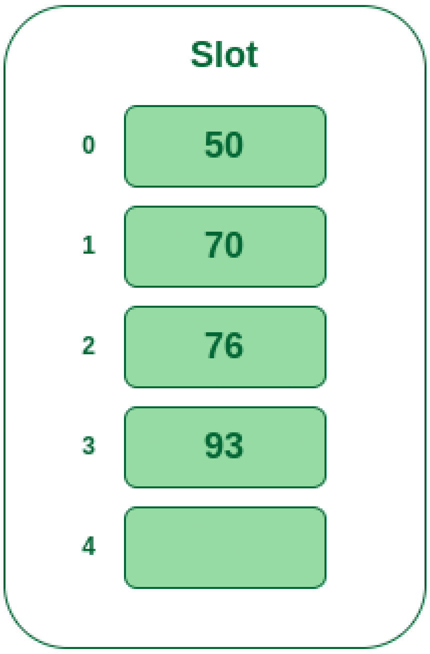
Insert 70 into hash table

- **Step 4:** The next key is 76. It will map to slot number 1 because 76%5=1 but 70 is already at slot number 1 so, search for the next empty slot and insert it.



Insert 76 into hash table

- **Step 5:** The next key is 93 It will map to slot number 3 because 93%5=3, So insert it into slot number 3.



Insert 93 into hash table

2. Quadratic Probing

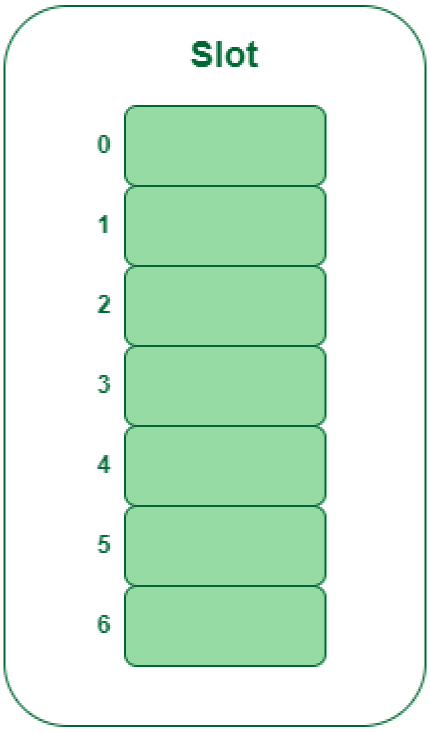
If we observe carefully, then we can understand that the interval between probes will increase proportionally to the hash value. **Quadratic prob**ing is a method with the help of which we can solve the problem of clustering that was discussed above. **This method is also known as the mid-square** method. In this method, we look for the **i^2'th** slot in the **i^{th}** iteration. We always start from the original hash location. If only the location is occupied then we check the other slots.

let hash(x) be the slot index computed using hash function.

*If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S*
*If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S*
*If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S*
……………………………………………
……………………………………………

Example: Let us consider table Size = 7, hash function as Hash(x) = x % 7 and collision resolution strategy to be f(i) = i^2. Insert = 22, 30, and 50.

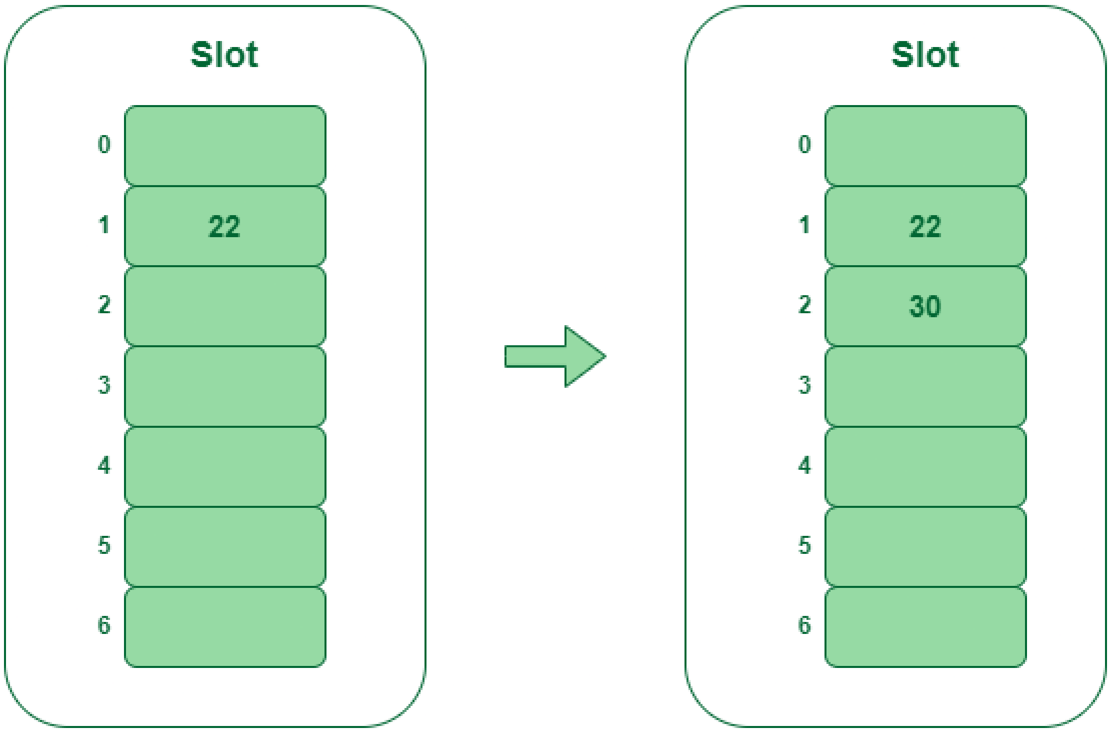
- **Step 1:** Create a table of size 7.



Hash table

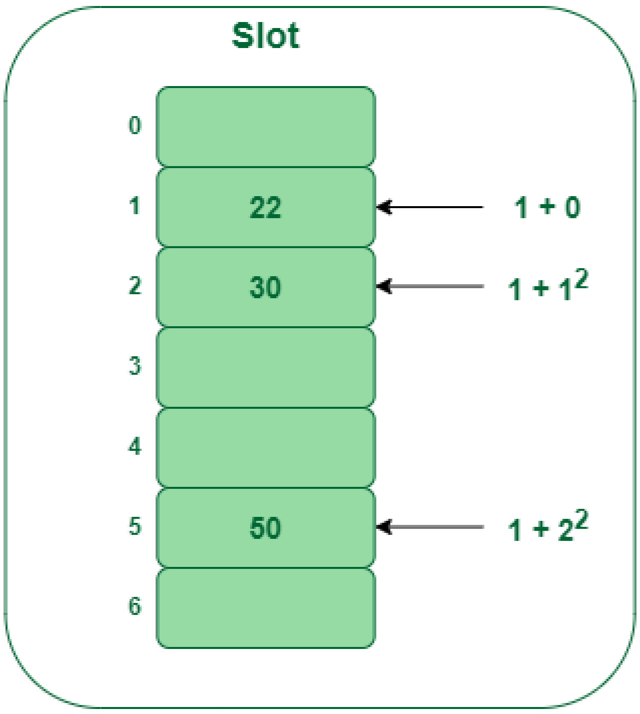
- **Step 2** – Insert 22 and 30

- Hash(22) = 22 % 7 = 1, Since the cell at index 1 is empty, we can easily insert 22 at slot 1.
- Hash(30) = 30 % 7 = 2, Since the cell at index 2 is empty, we can easily insert 30 at slot 2.



Insert keys 22 and 30 in the hash table

- **Step 3:** Inserting 50
    - Hash(50) = 50 % 7 = 1
    - In our hash table slot 1 is already occupied. So, we will search for slot $1+1^2$, i.e. 1+1 = 2,
    - Again slot 2 is found occupied, so we will search for cell $1+2^2$, i.e.1+4 = 5,
    - Now, cell 5 is not occupied so we will place 50 in slot 5.



Insert key 50 in the hash table

3. Double Hashing

The intervals that lie between probes are computed by another hash function. Double hashing is a technique that reduces clustering in an optimized way. In this technique, the increments for the probing sequence are computed by using another hash function. We use another hash function hash2(x) and look for the i*hash2(x) slot in the **i**th rotation.

let hash(x) be the slot index computed using hash function.

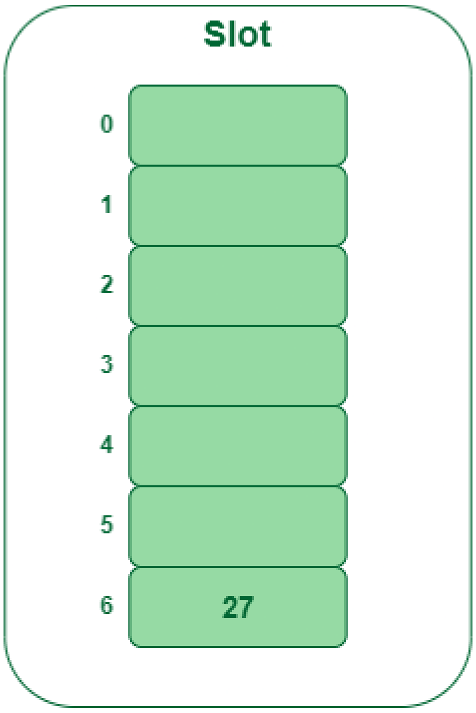
*If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S*
*If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) %S*
*If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) %S*

…………………………………………

…………………………………………

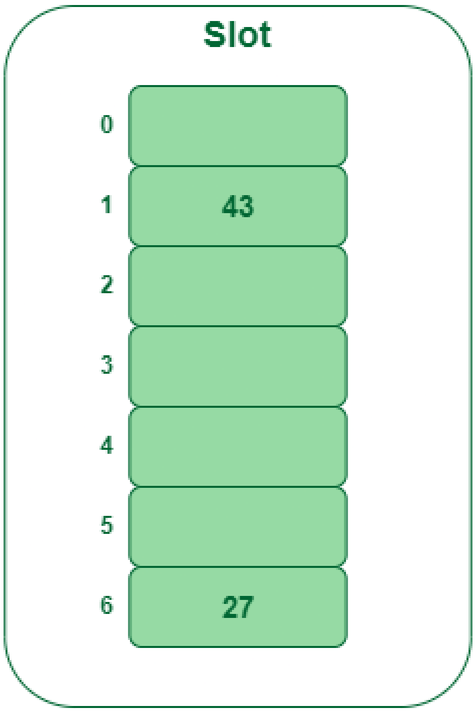
Example: Insert the keys 27, 43, 692, 72 into the Hash Table of size 7. where first hash-function is **h1(k) = k mod 7** and second hash-function is **h2(k) = 1 + (k mod 5)**

- **Step 1:** Insert 27
    - 27 % 7 = 6, location 6 is empty so insert 27 into 6 slot.



Insert key 27 in the hash table

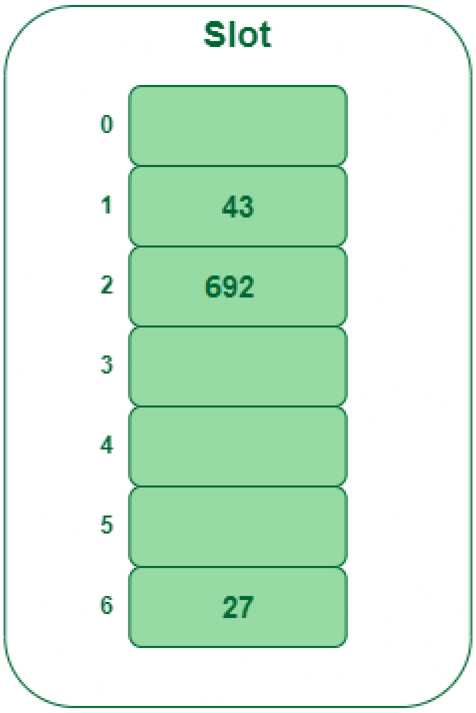
- **Step 2:** Insert 43
    - 43 % 7 = 1, location 1 is empty so insert 43 into 1 slot.



Insert key 43 in the hash table

- **Step 3:** Insert 692
    - 692 % 7 = 6, but location 6 is already being occupied and this is a collision
    - So we need to resolve this collision using double hashing.

*h_{new} = [h1(692) + i * (h2(692)] % 7*
*= [6 + 1 * (1 + 692 % 5)] % 7*
= 9% 7
= 2
Now, as 2 is an empty slot,
so we can insert 692 into 2nd slot.

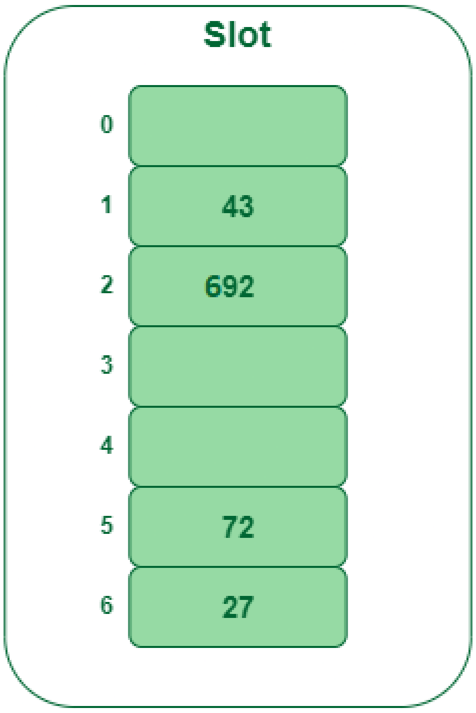


Insert key 692 in the hash table

- **Step 4:** Insert 72
    - 72 % 7 = 2, but location 2 is already being occupied and this is a collision.
    - So we need to resolve this collision using double hashing.

*h_{new} = [h1(72) + i * (h2(72)] % 7*
*= [2 + 1 * (1 + 72 % 5)] % 7*
= 5 % 7
= 5,

Now, as 5 is an empty slot,
so we can insert 72 into 5th slot.



Comparison of the above three:

Open addressing is a collision handling technique used in hashing where, when a collision occurs (i.e., when two or more keys map to the same slot), the algorithm looks for another empty slot in the hash table to store the collided key.

- In **linear probing**, the algorithm simply looks for the next available slot in the hash table and places the collided key there. If that slot is also occupied, the algorithm continues searching for the next available slot until an empty slot is found. This process is repeated until all collided keys have been stored. Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.
- In **quadratic probing**, the algorithm searches for slots in a more spaced-out manner. When a collision occurs, the algorithm looks for the next slot using an equation that involves the original hash value and a quadratic function. If that slot is also occupied, the algorithm increments the value of the quadratic function and tries again. This process is repeated until an empty slot is found. Quadratic probing lies between the two in terms of cache performance and clustering.
- In **double hashing**, the algorithm uses a second hash function to determine the next slot to check when a collision occurs. The

algorithm calculates a hash value using the original hash function, then uses the second hash function to calculate an offset. The algorithm then checks the slot that is the sum of the original hash value and the offset. If that slot is occupied, the algorithm increments the offset and tries again. This process is repeated until an empty slot is found. Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

The choice of collision handling technique can have a significant impact on the **performance of a hash table**.

Linear probing is simple and fast, but it can lead to clustering (i.e., a situation where keys are stored in long contiguous runs) and can degrade performance.

Quadratic probing is more spaced out, but it can also lead to clustering and can result in a situation where some slots are never checked.

Double hashing is more complex, but it can lead to more even distribution of keys and can provide better performance in some cases.

S.No.	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how	Open addressing is used when the frequency and number of keys is known.

S.No.	Separate Chaining	Open Addressing
	frequently keys may be inserted or deleted.	
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Performance of Open Addressing:

Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

m = Number of slots in the hash table

n = Number of keys to be inserted in the hash table

Load factor $\alpha = n/m$ (< 1)

Expected time to search/insert/delete $< 1/(1 - \alpha)$

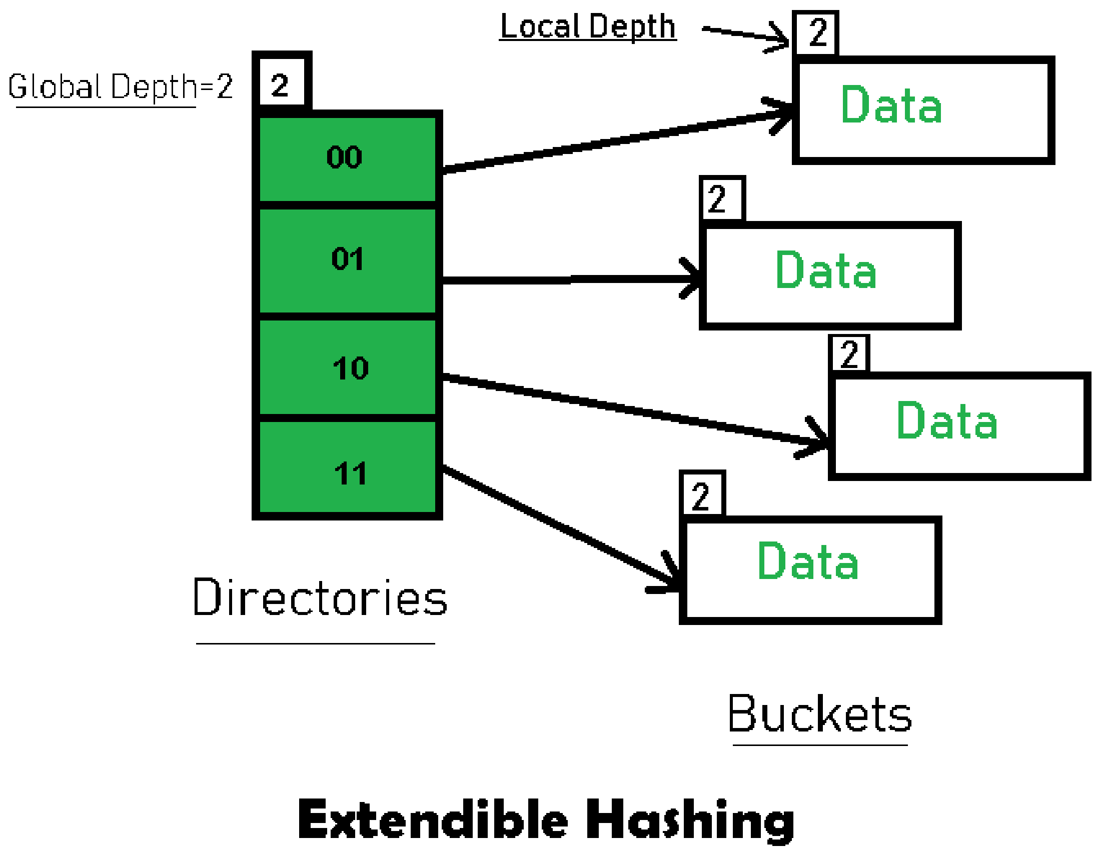
So Search, Insert and Delete take $(1/(1 - \alpha))$ time

DYNAMIC HASHING or Extendible Hashing is a dynamic hashing method wherein directories, and buckets are used to hash data. It is an aggressively flexible method in which the hash function also experiences dynamic chang es.

Main features of Extendible Hashing: The main features in this hashing technique are:

- **Directories:** The directories store addresses of the buckets in pointers. An id is assigned to each directory which may change each time when Directory Expansion takes place.
- **Buckets:** The buckets are used to hash the actual data.

Basic Structure of Extendible Hashing:



Frequently used terms in Extendible Hashing:

- **Directories:** These containers store pointers to buckets. Each directory is given a unique id which may change each time when expansion takes place. The hash function returns this directory id which is used to navigate to the appropriate bucket. Number of Directories = 2^Global Depth.

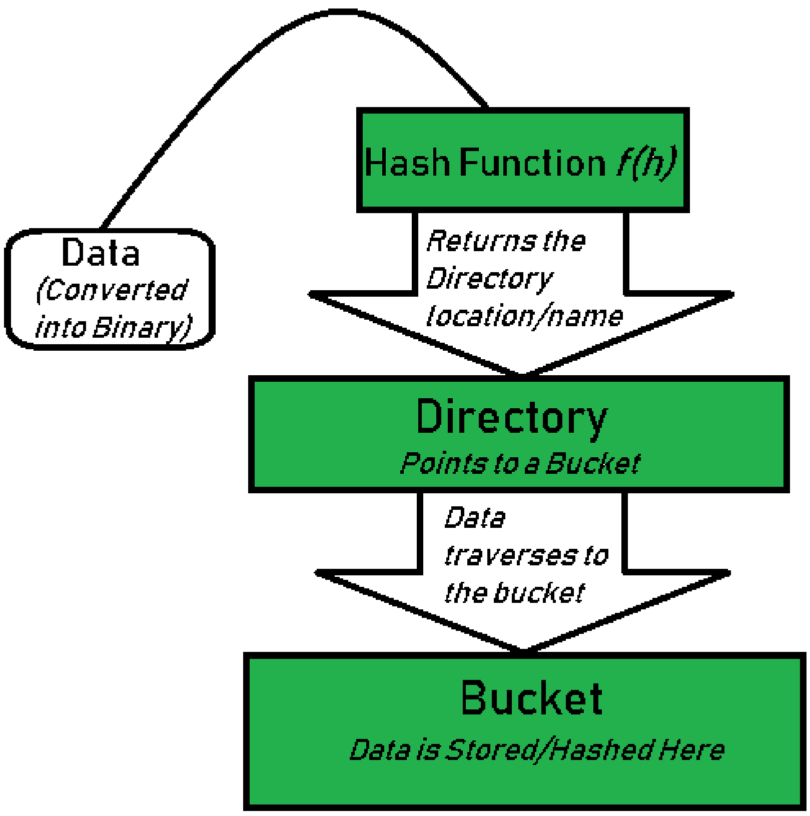
- **Buckets:** They store the hashed keys. Directories point to buckets. A bucket may contain more than one pointers to it if its local depth is less than the global depth.

- **Global Depth:** It is associated with the Directories. They denote the number of bits which are used by the hash function to categorize the keys. Global Depth = Number of bits in directory id.
- **Local Depth:** It is the same as that of Global Depth except for the fact that Local Depth is associated with the buckets and not the directories. Local depth in accordance with the global depth is used to decide the action that to be performed in case an overflow occurs. Local Depth is always less than or equal to the Global Depth.

- **Bucket Splitting:** When the number of elements in a bucket exceeds a particular size, then the bucket is split into two parts.
- **Directory Expansion:** Directory Expansion Takes place when a bucket overflows. Directory Expansion is performed when the local depth of the overflowing bucket is equal to the global depth.

Basic Working of Extendible Hashing:

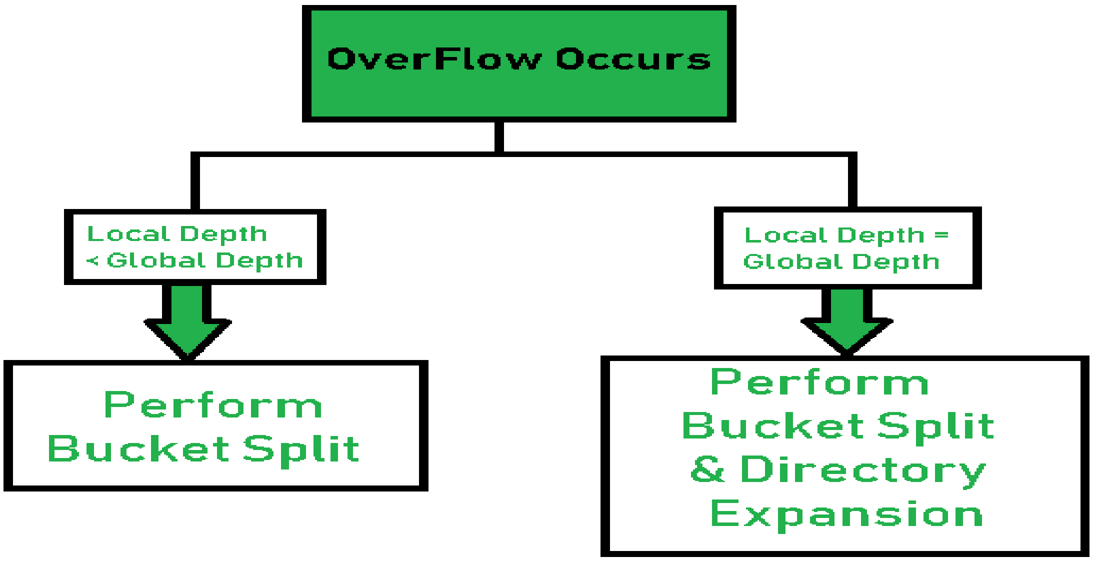
-


Data (Converted into Binary)
Hash Function f(h)
Returns the Directory location/name
Directory
Points to a Bucket
Data traverses to the bucket
Bucket
Data is Stored/Hashed Here

- **Step 1 – Analyze Data Elements:** Data elements may exist in various forms eg. Integer, String, Float, etc.. Currently, let us consider data elements of type integer. eg: 49.
- **Step 2 – Convert into binary format:** Convert the data element in Binary form. For string elements, consider the ASCII equivalent integer of the starting character and then convert the integer into binary form. Since we have 49 as our data element, its binary form is 110001.
- **Step 3 – Check Global Depth of the directory.** Suppose the global depth of the Hash-directory is 3.
- **Step 4 – Identify the Directory:** Consider the 'Global-Depth' number of LSBs in the binary number and match it to the directory id.
  Eg. The binary obtained is: 110001 and the global-depth is 3. So, the hash function will return 3 LSBs of 110**001** viz. 001.
- **Step 5 – Navigation:** Now, navigate to the bucket pointed by the directory with directory-id 001.

- **Step 6 – Insertion and Overflow Check:** Insert the element and check if the bucket overflows. If an overflow is encountered, go to **step 7** followed by **Step 8**, otherwise, go to **step 9**.
- **Step 7 – Tackling Over Flow Condition during Data Insertion:** Many times, while inserting data in the buckets, it might happen that the Bucket overflows. In such cases, we need to follow an appropriate procedure to avoid mishandling of data. First, Check if the local depth is less than or equal to the global depth. Then choose one of the cases below.

  - **Case1:** If the local depth of the overflowing Bucket is equal to the global depth, then Directory Expansion, as well as Bucket Split, needs to be performed. Then increment the global depth and the local depth value by 1. And, assign appropriate pointers.
    Directory expansion will double the number of directories present in the hash structure.
  - **Case2:** In case the local depth is less than the global depth, then only Bucket Split takes place. Then increment only the local depth value by 1. And, assign appropriate pointers.

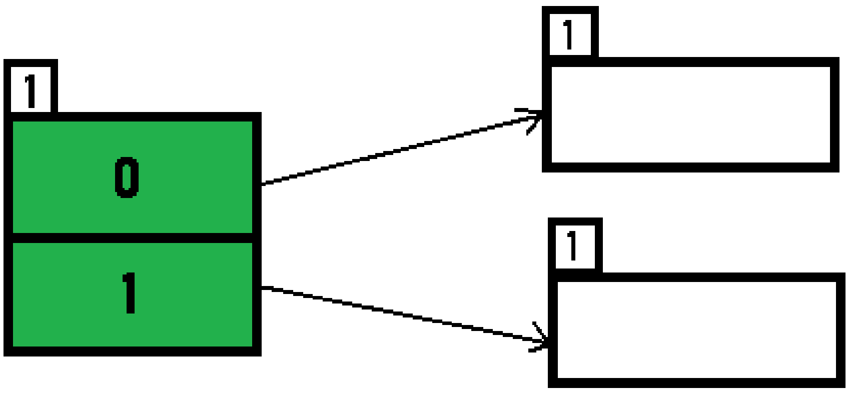

OverFlow Occurs
Local Depth < Global Depth
Local Depth = Global Depth
Perform Bucket Split
Perform Bucket Split & Directory Expansion

- **Step 8 – Rehashing of Split Bucket Elements:** The Elements present in the overflowing bucket that is split are rehashed w.r.t the new global depth of the directory.

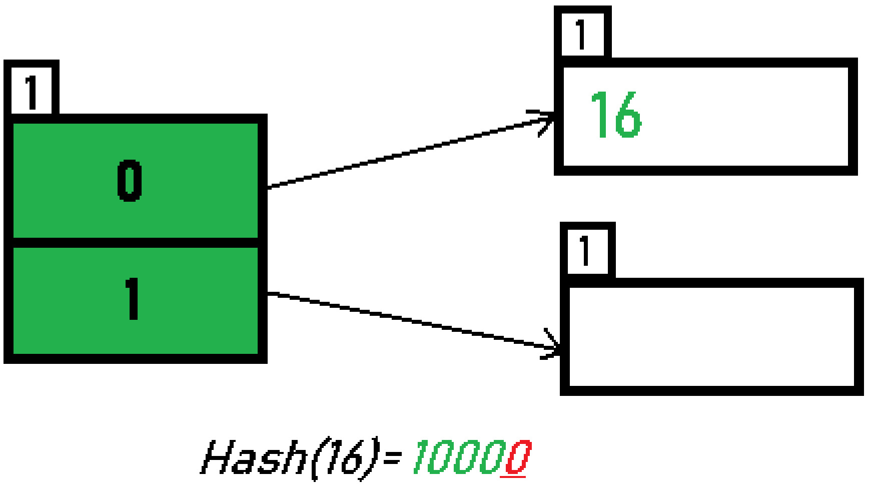
- **Step 9** – The element is successfully hashed.

Example based on Extendible Hashing: Now, let us consider a prominent example of hashing the following elements: **16,4,6,22,24,10,31,7,9,20,26.**
Bucket Size: 3 (Assume)
Hash Function: Suppose the global depth is X. Then the Hash Function returns X LSBs.

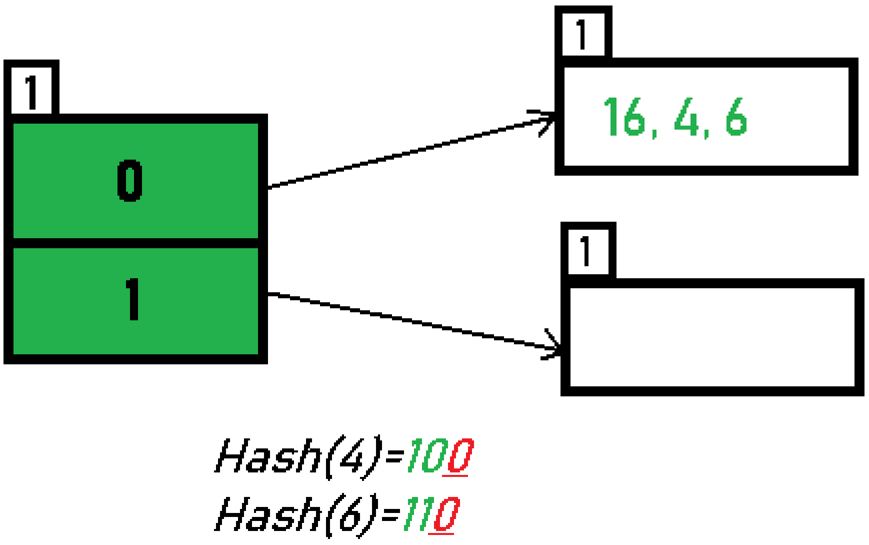
- **Solution:** First, calculate the binary forms of each of the given numbers.
  16- 10000
  4- 00100
  6- 00110
  22- 10110
  24- 11000
  10- 01010
  31- 11111
  7- 00111
  9- 01001
  20- 10100
  26- 11010
- Initially, the global-depth and local-depth is always 1. Thus, the hashing frame looks like this:



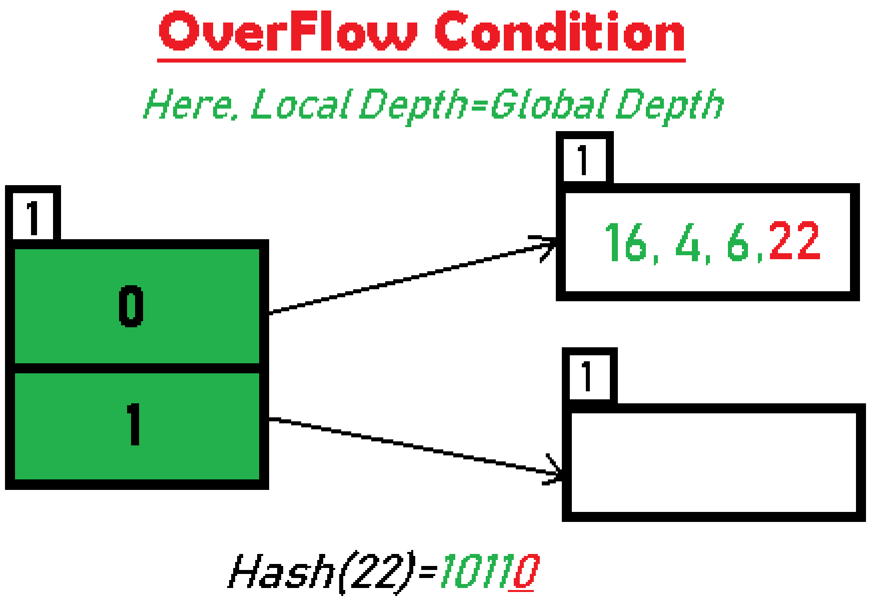
- **Inserting 16:**
  The binary format of 16 is 10000 and global-depth is 1. The hash function returns 1 LSB of 1000**0** which is 0. Hence, 16 is mapped to the directory with id=0.



- **Inserting 4 and 6:**
  Both 4(10**0**) and 6(11**0**)have 0 in their LSB. Hence, they are hashed as follows:

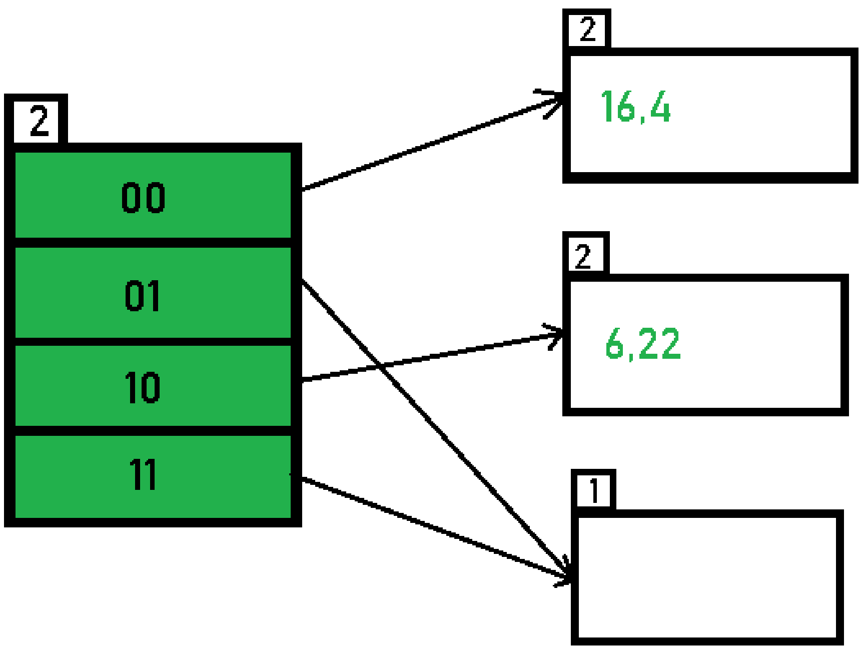


- **Inserting 22:** The binary form of 22 is 1011**0**. Its LSB is 0. The bucket pointed by directory 0 is already full. Hence, Over Flow occurs.



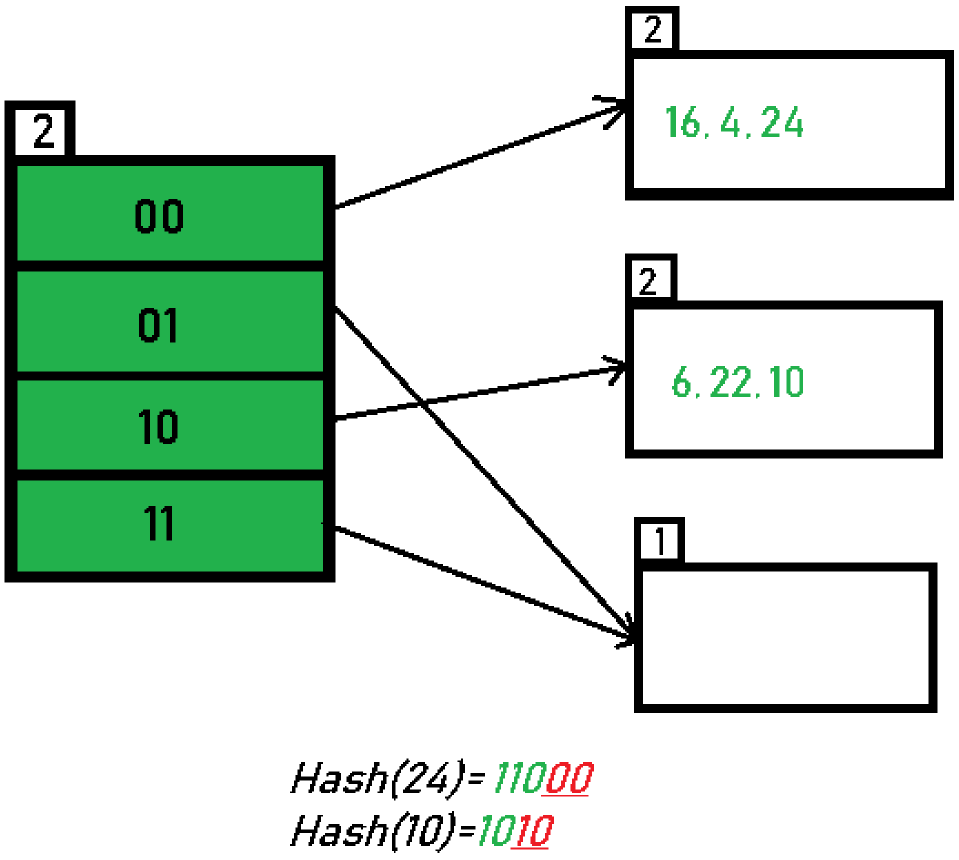
- As directed by **Step 7-Case 1**, Since Local Depth = Global Depth, the bucket splits and directory expansion takes place. Also, rehashing of numbers present in the overflowing bucket takes place after the split. And, since the global depth is incremented by 1, now,the global depth is 2.
- Hence, 16,4,6,22 are now rehashed w.r.t 2
- LSBs.[16(100**00**),4(1**00**),6(1**10**),22(101**10**)]

After Bucket Split and Directory Expansion

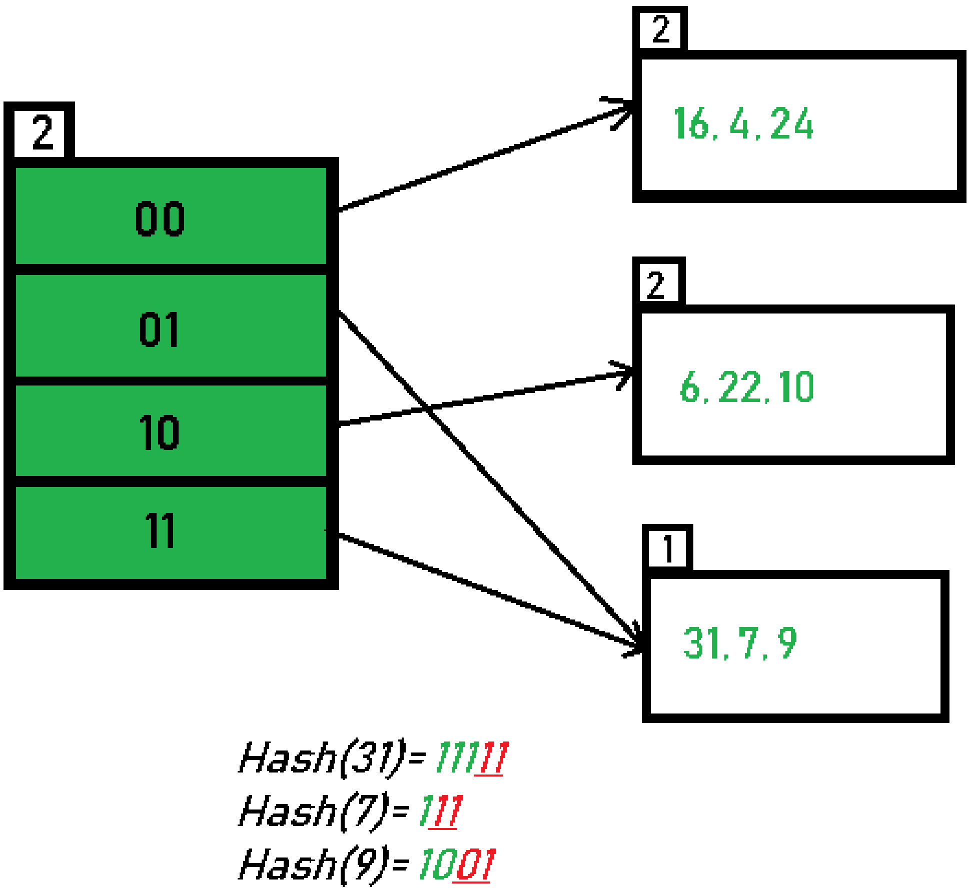


**Notice that the bucket which was underflow has remained untouched. But, since the number of directories has doubled, we now have 2 directories 01 and 11 pointing to the same bucket. This is because the local-depth of the bucket has remained 1. And, any bucket having a local depth less than the global depth is pointed-to by more than one directories.*

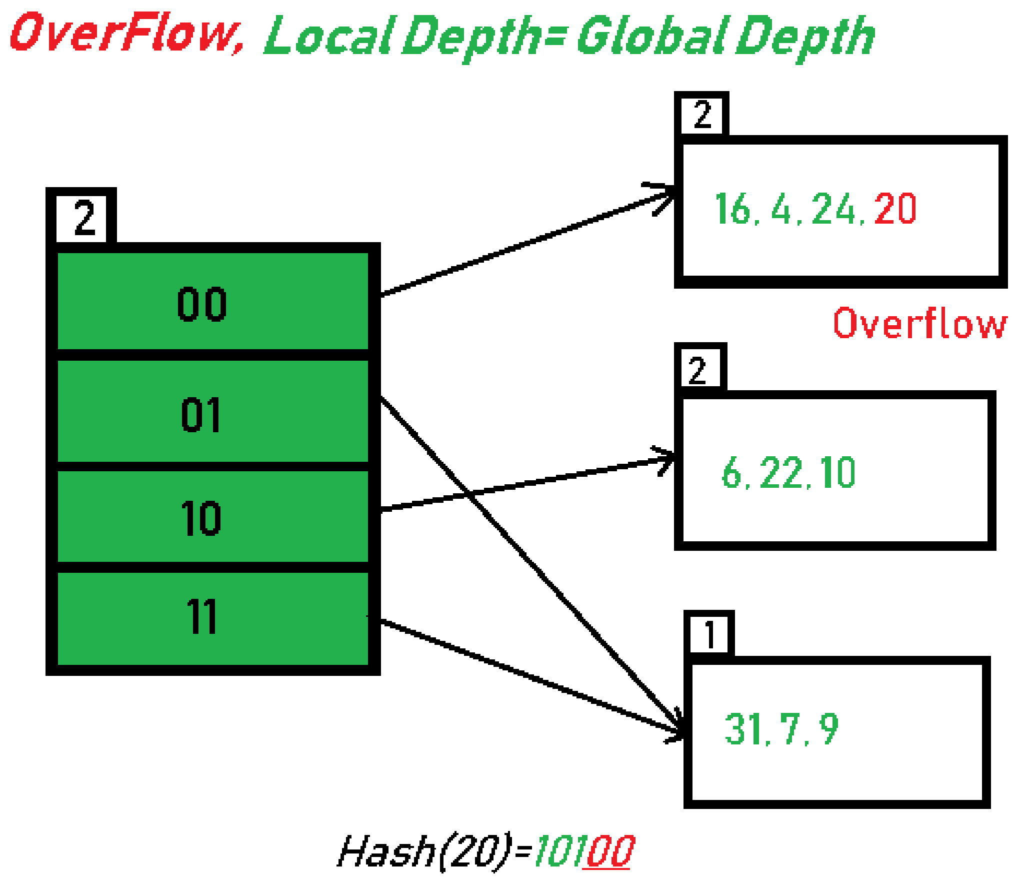
- **Inserting 24 and 10:** 24(110**00**) and 10 (10**10**) can be hashed based on directories with id 00 and 10. Here, we encounter no overflow condition.



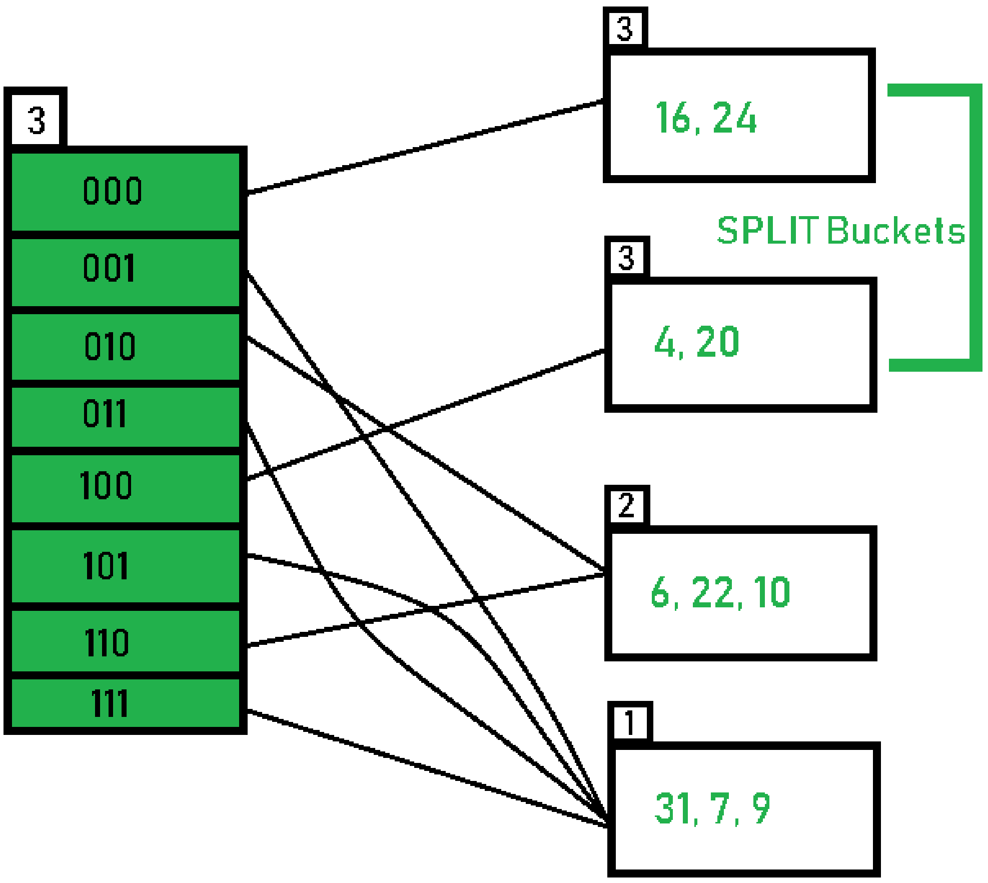
- **Inserting 31,7,9:** All of these elements[31(111**11**), 7(1**11**), 9(10**01**)] have either 01 or 11 in their LSBs. Hence, they are mapped on the bucket pointed out by 01 and 11. We do not encounter any overflow condition here.



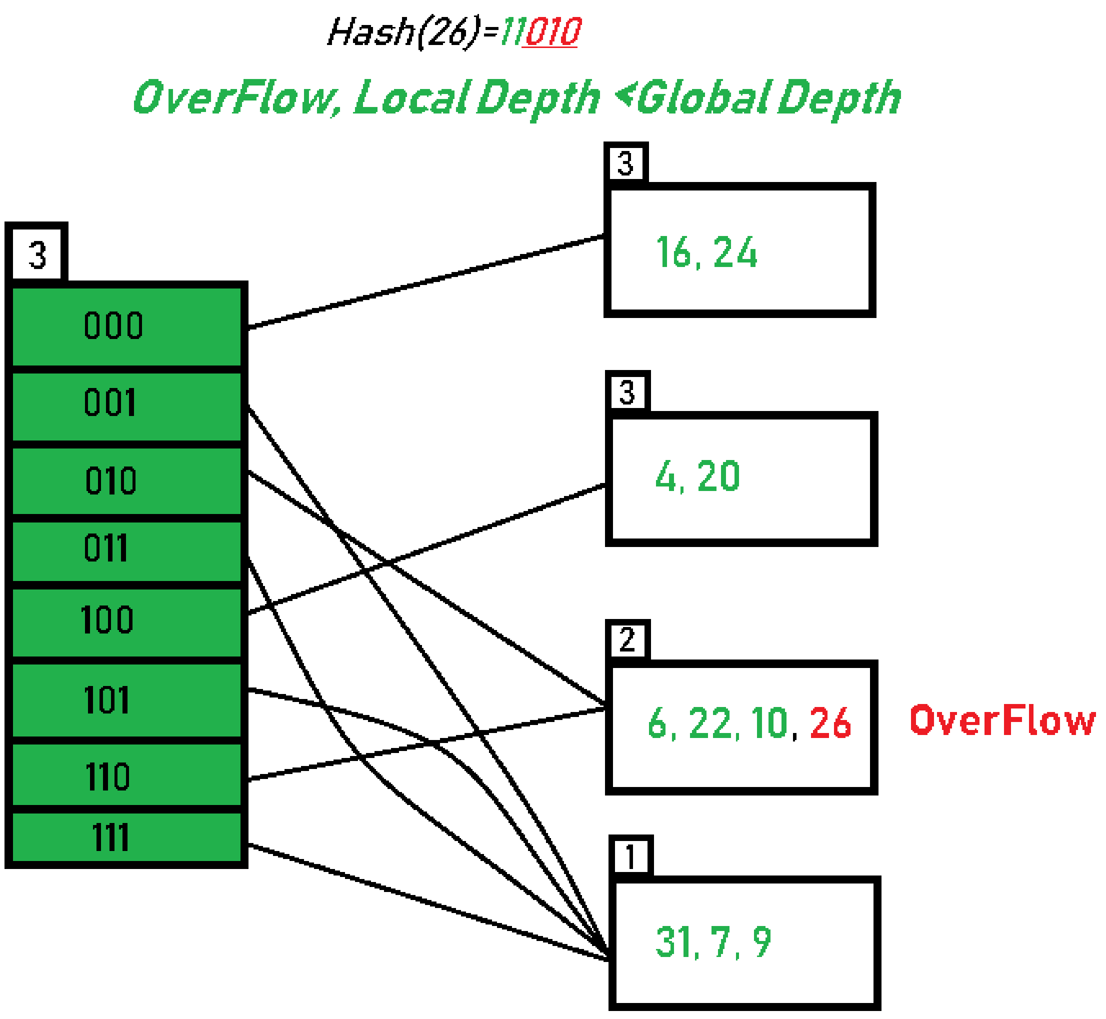
- **Inserting 20:** Insertion of data element 20 (101**00**) will again cause the overflow problem.



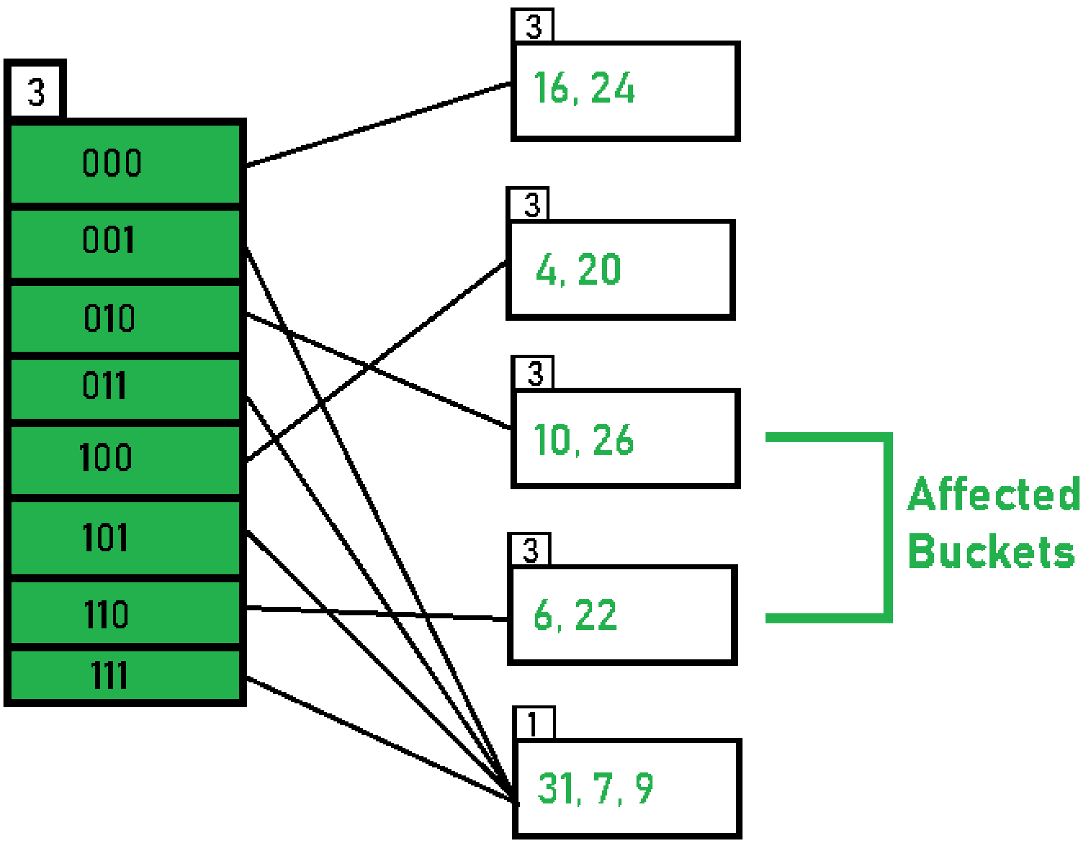
- 20 is inserted in bucket pointed out by 00. As directed by **Step 7-Case 1**, since the **local depth of the bucket = global-depth**, directory expansion (doubling) takes place along with bucket splitting. Elements present in overflowing bucket are rehashed with the new global depth. Now, the new Hash table looks like this:



- **Inserting 26:** Global depth is 3. Hence, 3 LSBs of 26(11**010**) are considered. Therefore 26 best fits in the bucket pointed out by directory 010.



- The bucket overflows, and, as directed by **Step 7-Case 2,** since the **local depth of bucket < Global depth (2<3)**, directories are not doubled but, only the bucket is split and elements are rehashed.
  Finally, the output of hashing the given list of numbers is obtained.



- **Hashing of 11 Numbers is Thus Completed.**

Key Observations:

1. A Bucket will have more than one pointers pointing to it if its local depth is less than the global depth.
2. When overflow condition occurs in a bucket, all the entries in the bucket are rehashed with a new local depth.
3. If Local Depth of the overflowing bucket
4. The size of a bucket cannot be changed after the data insertion process begins.

Advantages:

1. Data retrieval is less expensive (in terms of computing).
2. No problem of Data-loss since the storage capacity increases dynamically.

3. With dynamic changes in hashing function, associated old values are rehashed w.r.t the new hash function.

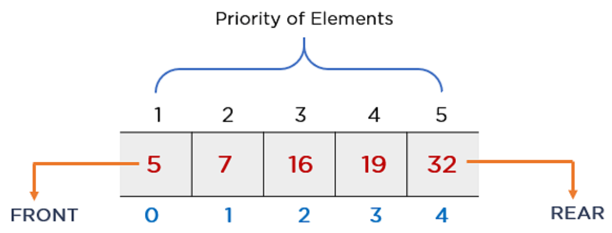
Limitations Of Extendible Hashing:

1. The directory size may increase significantly if several records are hashed on the same directory while keeping the record distribution non-uniform.
2. Size of every bucket is fixed.
3. Memory is wasted in pointers when the global depth and local depth difference becomes drastic.
4. This method is complicated to code.

5.1. Priority Queues: Single and double ended Priority Queues, Leftist Trees

5.1.1. Priority Queues : Introduction

- A priority queue is a data structure which is an extension of normal queue that orders elements based on their priority. The elements in a priority queue are processed in order of their priority, with the highest-priority element processed first.
- New elements are added to the priority queue based on their priority order, and the highest-priority element is always at the front of the queue.
- Heap structure is a classic data structure for representation of priority queue.

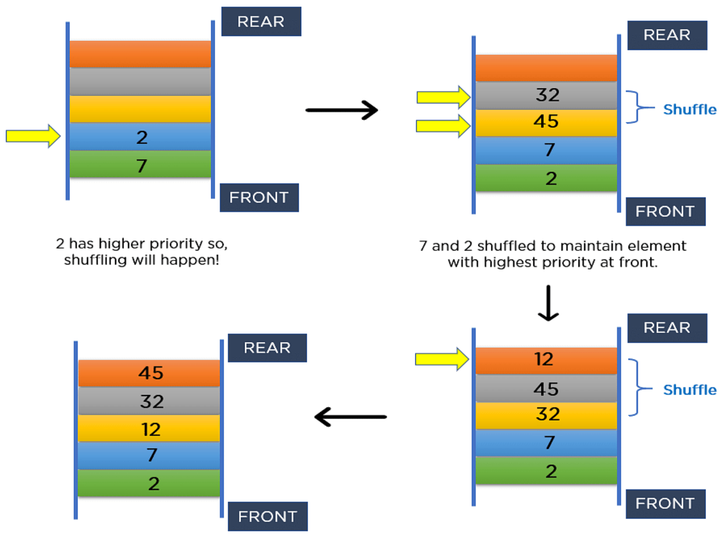


Characteristics of a Priority queue

- Every element in a priority queue has some priority associated with it.
- Every element of this queue must be comparable.
- An element with the higher priority will be deleted before the deletion of the lower priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Example

- Consider an elements to insert 7, 2, 45, 32, and 12 in a priority queue.
- The element with the least value has the highest property. Thus, you should maintain the lowest element at the front node.


REAR
2
7
FRONT
2 has higher priority so, shuffling will happen!
REAR
32
45
7
2
Shuffle
FRONT
7 and 2 shuffled to maintain element with highest priority at front.
REAR
12
45
32
7
2
Shuffle
FRONT
REAR
45
32
12
7
2
FRONT

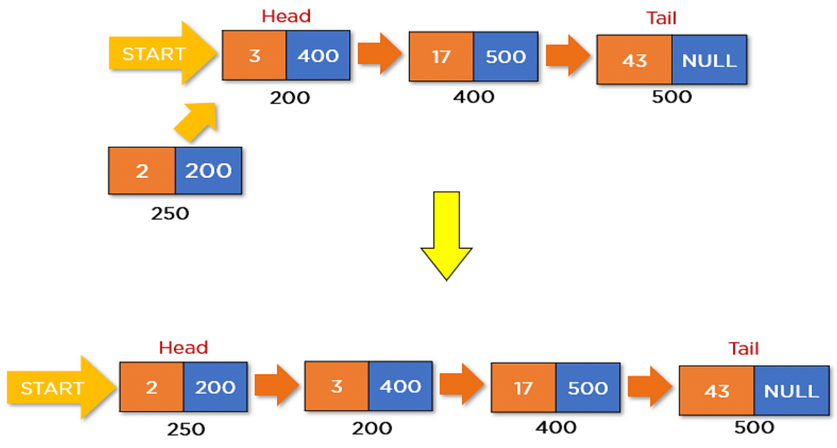
- The above illustration how it maintains the priority during insertion in a queue. But, if you carry the N comparisons for each insertion, time-complexity will become O(N^2).

Representation of Priority Queue

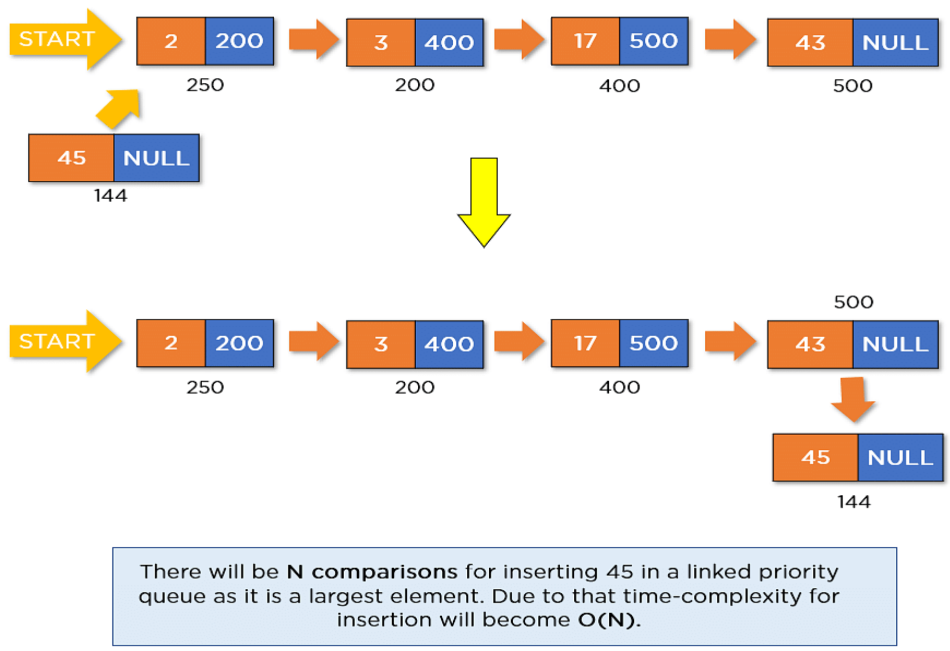
Implementation	Insert	Remove	Peek
Array	$\mathbf{O}(n^2)$	**O(1)**	**O(1)**
Linked List	**O(n)**	**O(1)**	**O(1)**
Binary Heap	**O(log n)**	**O(log n)**	**O(1)**
Binary Tree	**O(log n)**	**O(log n)**	**O(1)**

Priority Queue using Linked List

- Consider a linked queue having 3 data elements 3, 17, 43



- For instance, to insert a node consisting of element 45. Here, it will compare element 45 with each element inside the queue. However, this insertion will cost you O(N). Representation of the linked queue below displays how it will insert element 45 in a priority queue.



There will be **N comparisons** for inserting 45 in a linked priority queue as it is a largest element. Due to that time-complexity for insertion will become **O(N)**.

Types of Priority Queue

- There are two types of priority queues based on the priority of elements.

    - If the element with the smallest value has the highest priority, then that priority queue is called the min priority queue.
    - If the element with a higher value has the highest priority, then that priority queue is known as the max priority queue.

- Furthermore, you can implement the min priority queue using a min heap, whereas you can implement the max priority queue using a max heap.

Applications of Priority Queue in Data Structure

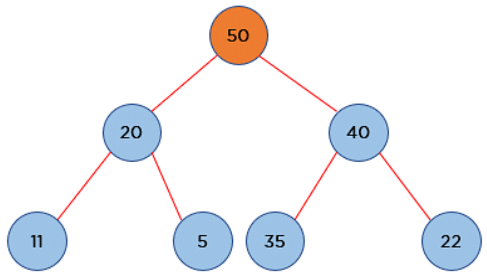
- Used in the Dijkstra's shortest path algorithm.
- IP Routing to Find Open Shortest Path First
- Data Compression in like Huffman code, WINZIP / GZIP
- Used in implementing Prim's algorithm
- Used to perform the heap sort
- Used in Scheduling, Load balancing and Interrupt handling

Operations supported by priority queue.

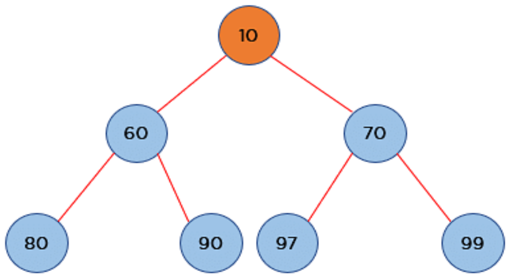
- Insertion (Enqueue): Insert an element into the priority queue along with its priority.
- Deletion (Dequeue): Remove and return the element with the highest priority from the priority queue.
- Peek: Retrieve the element with the highest priority without removing it from the priority queue.
- Size: Get the number of elements currently stored in the priority queue.
- Clear: Remove all elements from the priority queue, making it empty.
- Update Priority: Change the priority of an existing element in the priority queue.

Representation of Priority Queue using Heap

- Heap is Tree like data structure that forms complete binary tree
- Parent node value is always greater than or equal to child node for all nodes in the heap (max Heap) and reverse for min heap



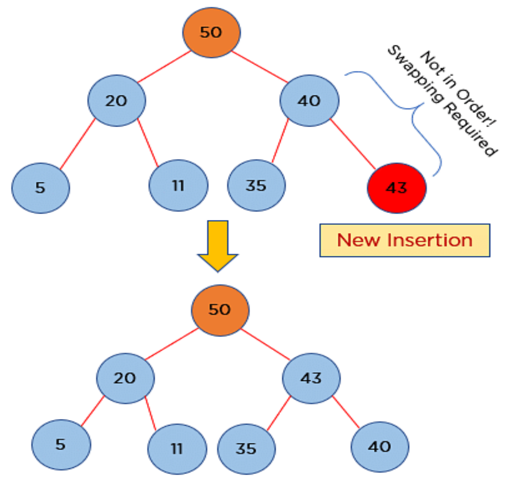
The Value of **Parent Node** is **greater** than **child node.**



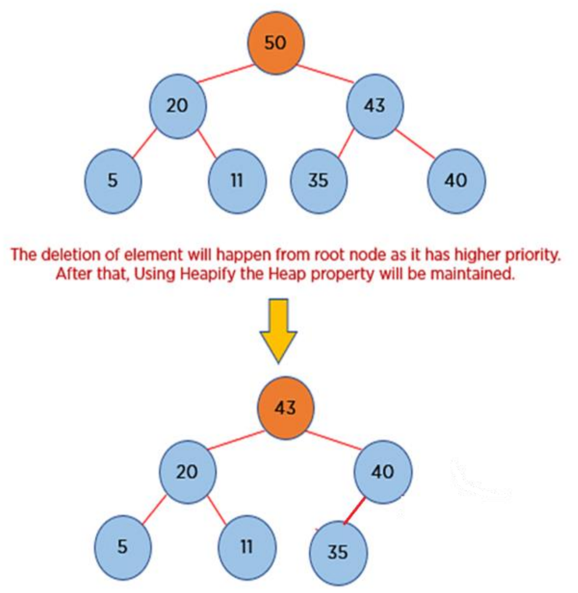
The Value of **Parent Node** is **smaller** than **child node.**

Insert

- The new element will move to the empty space from top to bottom and left to right and heapify.



ι eration **Heapify** manages the shuffling of element in **Heap** data structure.


50
20
43
5
11
35
40
The deletion of element will happen from root node as it has higher priority.
After that, Using Heapify the Heap property will be maintained.
43
20
40
5
11
35

Delete

- The maximum/minimum element is the root node which will be deleted based on max/min heap. (refer above right diagram)

- **Example 2: 32, 15, 20, 30, 12, 25, 16**

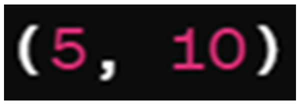
5.1.2. Single ended priority queue

- A single-ended priority queue is a type of priority queue where elements are inserted with associated priorities, and removal (dequeuing) is performed only from one end, typically from the front. In this type of priority queue, the elements with the highest priority are dequeued first.
- 2 types
  - Ascending Priority Queue (Min Priority Queue)
    - Return an element with minimum priority
    - Insert an element at arbitrary priority
    - Delete an element with minimum priority
  - Descending Priority Queue (Max Priority Queue)
    - Return an element with Maximum priority
    - Insert an element at arbitrary priority
    - Delete an element with maximum priority

- Common Operations
  - Return an element with minimum/Maximum priority
  - Insert an element at arbitrary priority
  - Delete an element with maximum / minimum priority

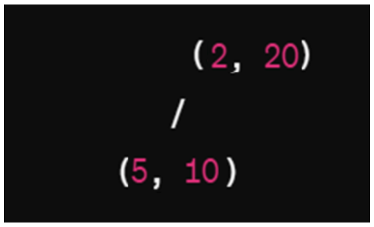
- Example: Consider elements to insert: (5, 10), (2, 20), (8, 30), (1,40), (7, 50)

  Insert (5, 10)
  - Start with an empty priority queue.
  - The first element (5, 10) is inserted as the root node since the priority queue is initially empty.



```
(5, 10)
```

  Insert (2, 20)
  - The second element (2, 20) is inserted as the left child of the root since it has a higher priority than the root (5). Here, **Heapify Up** is applied

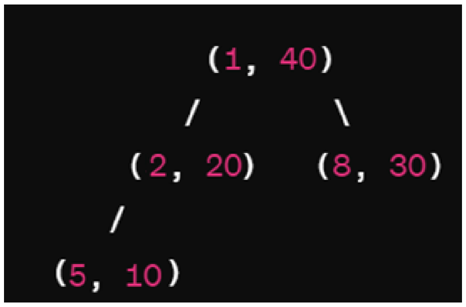


```
        (2, 20)
       /
   (5, 10)
```

- Similarly, Insert (8, 30) & Insert (5, 10)

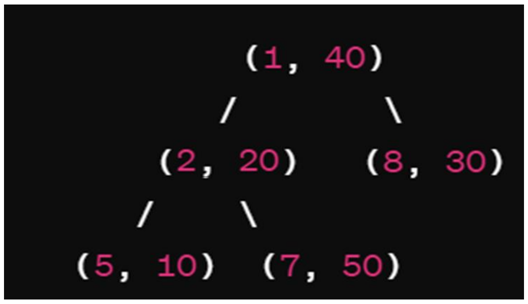
Insert (1, 40)

- The fourth element (1, 40) is inserted as the left child of the left child of the root since it has the highest priority among all elements.



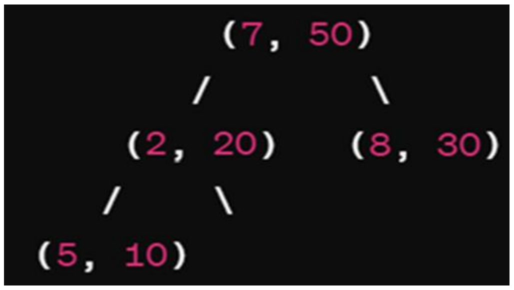
Insert (7, 50)

- The fifth element (7, 50) is inserted as the right child of the left child of the root since it has a higher priority than the root (5) but lower priority than (2) and (1).

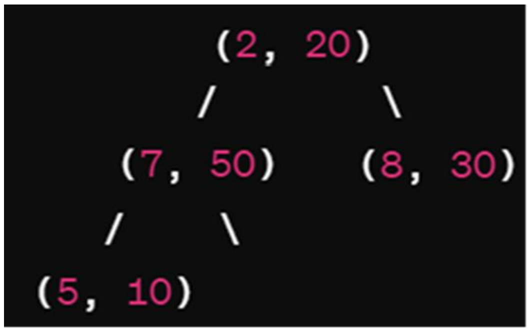


- **Delete min**
  - The minimum element (1, 40) is deleted from the priority queue. The root node is replaced with the last node of the heap, and then the last node is deleted.

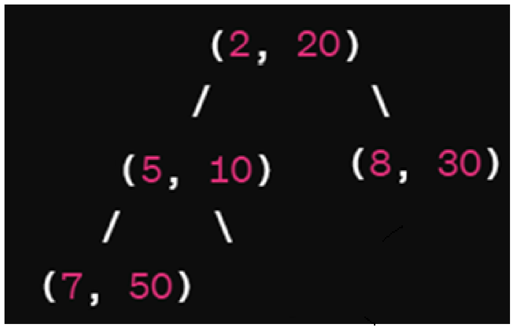
  - After deleting the minimum element, check the child nodes of (7, 50) and place smaller priority node to the root.



- After adjusting, now check the heap property for node (7,50). Node (7, 50) has 2 child nodes, apply min heap and move min priority node to root.



```
          (2, 20)
         /       \
    (7, 50)    (8, 30)
   /     \
(5, 10)
```



```
          (2, 20)
         /       \
    (5, 10)    (8, 30)
   /     \
(7, 50)
```

5.1.3. Double ended priority queue

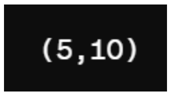
- Common Operations
  - Return an element with minimum priority
  - Return an element with maximum priority
  - Insert an element at arbitrary priority
  - Delete an element with minimum priority
  - Delete an element with maximum priority

Insertion

- Insertion is same as insertion in single ended priority queue

Inserting element (5, 10):

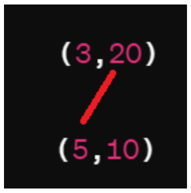
- Initially, the priority queue is empty.
- The first element (5,10) is inserted into the priority queue.
- Since this is the first element, it becomes the root of the tree.



```
(5,10)
```

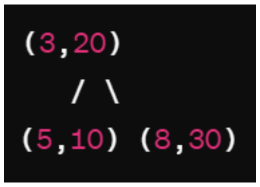
Inserting element (3, 20):

- The second element (3,20) is inserted into the priority queue.
- Since the priority of (3,20) is smaller than the priority of the root (5,10), it becomes the left child of the root.


```
  (3,20)
   /
 (5,10)
```

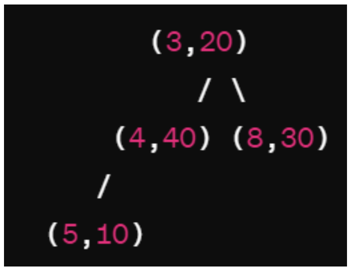
Inserting element (30,8):

- The third element (8,30) is inserted into the priority queue.
- Since the priority of (8,30) is greater than the priority of the root (5,10), it becomes the right child of the root.
- (8,30) is inserted as the right child of (5,10) since its priority is higher than the root.


```
(3,20)
   / \
(5,10) (8,30)
```

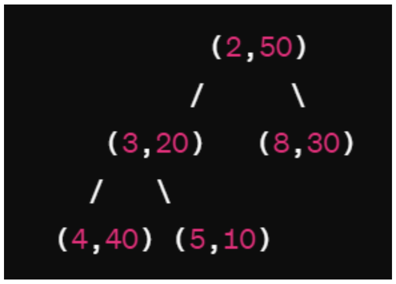
Inserting element (4, 40):

- The fourth element (4,40) is inserted into the priority queue.
- Since the priority of (4,40) is greater than the priority of its parent (3,20) but smaller than the priority of the root (5,10), it becomes the right child of (3,20).
- (4,40) is inserted as the right child of (3,20) since its priority is higher than its parent but lower than the root.


```
        (3,20)
           / \
      (4,40) (8,30)
     /
  (5,10)
```

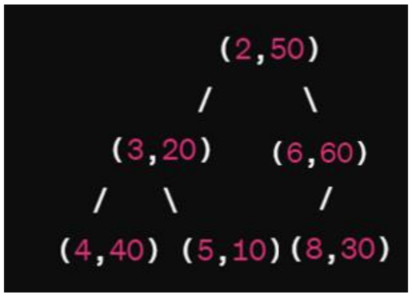
Inserting element (2, 50):

- The fifth element (2,50) is inserted into the priority queue.
- Since the priority of (2,50) is smaller than the priority of the root (3,20), it becomes the left child of the root.
- (2,50) is inserted as the left child of (3,20) since its priority is lower.



Inserting element (60,6):

- The sixth element (60,6) is inserted into the priority queue.
- Since the priority of (60,6) is greater than the priority of the root (10,5), it becomes the right child of the root.
- After insertion, the priority queue contains six elements: (50,2), (20,3), (10,5), (40,4), (30,8), and (60,6).



Returning Min/Max priority is same as single ended priority queue

Deleting Min/Max priority is same as single ended priority queue

Returning Maximum priority element using Min heap

- Start at the root node of the min-heap.
- Compare the priority of the root node with the priorities of its children.
- If any child has a higher priority than the root node, recursively explore that subtree.
- Continue this process until reaching a leaf node, which represents the end of the traversal.
- The element with the highest priority encountered during the traversal will be the maximum priority element in the min-heap.

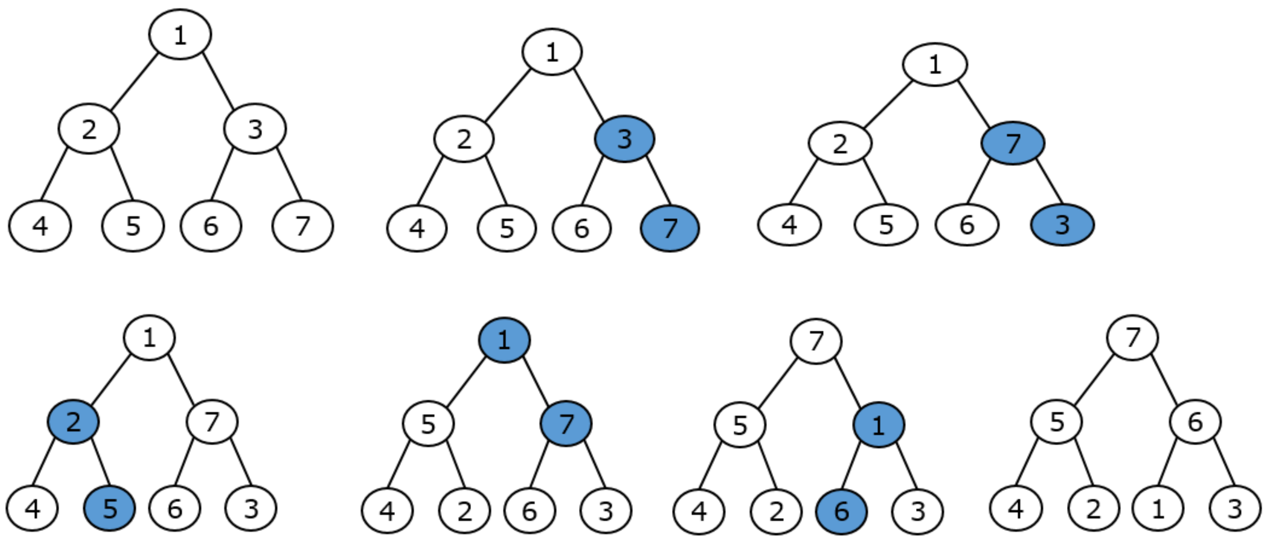
Delete an element with maximum priority

- Convert the min-heap into a max-heap.

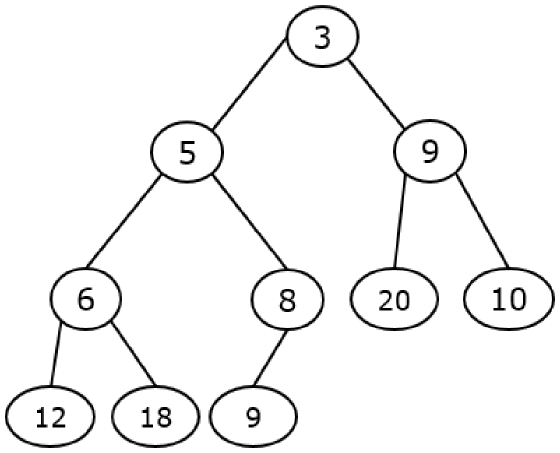
- The maximum priority element will now be at the root of the max-heap. Delete this element.
- Restore the max-heap property by adjusting the heap structure as necessary.

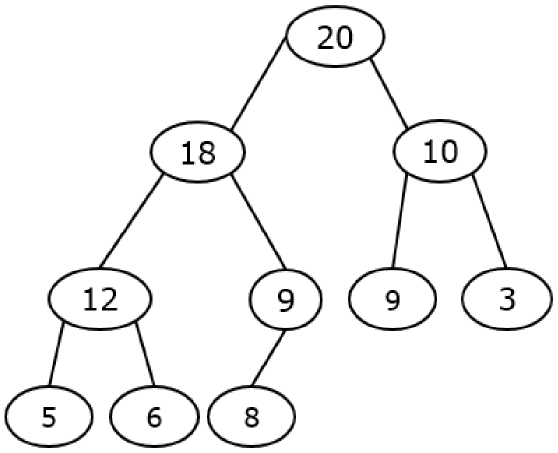
Conversion of min heap to max heap

- Start at the last non-leaf node of the heap (i.e., the parent of the last leaf node). For a binary heap, this node is located at the index floor((n – 1)/2), where n is the number of nodes in the heap.
- For each non-leaf node, perform a "heapify" operation to fix the heap property.
    - In a min heap, this operation involves checking whether the value of the node is greater than that of its children, and if so, swapping the node with the smaller of its children.
    - In a max heap, the operation involves checking whether the value of the node is less than that of its children, and if so, swapping the node with the larger of its children.
- Repeat step 2 for each of the non-leaf nodes, working your way up the heap. When you reach the root of the heap, the entire heap should now be a max heap.
- Consider the tree (fig .1) convert to max heap

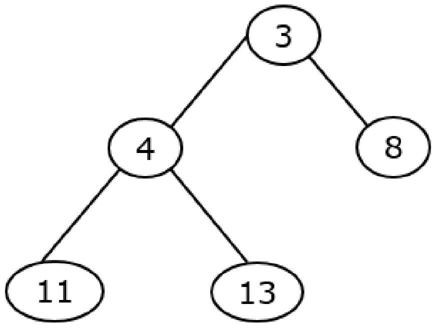


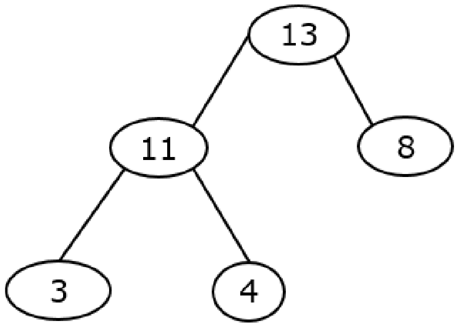
- Example 2: 3, 5, 9, 6, 8, 20, 10, 12, 18, 9





- Example 3: 3, 4, 8, 11, 13

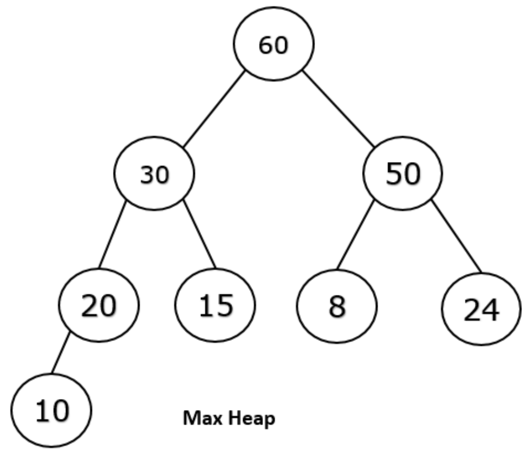


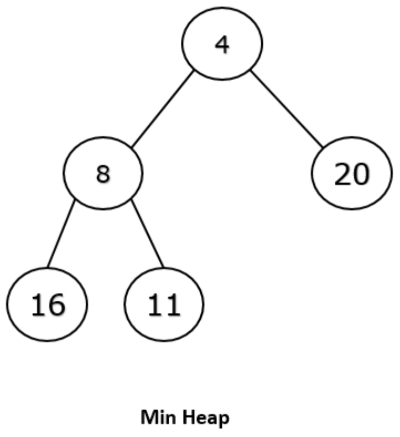


5.1.4. Leftist Trees

Limitations of Binary Heap

- A binary heap is a complete binary tree in which each and every node is greater than descendants or children node.
  $Max\ Heap: node(x) \geq descendents\ (x)$ $\quad$ $Min\ Heap: node(x) \geq descendents\ (x)$





- Consider to join above 2 trees and make a single binary heap
- We should create an array and copy m+n elements

$$T(n) = O(m+n) + O(m+n) * O(\log \mathrm{m} + n)$$
$$T(n) = O(t) + O(t) * O(\log t)$$
$$T(n) = O(n \log n)$$

$where\ O(m+n)\ is\ the\ complexity\ to\ copy\ nodes\ from\ both\ trees$

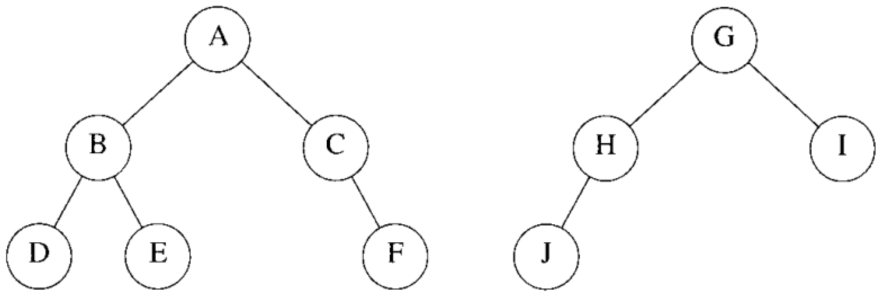
$$O(m+n) * O(\log + n)\ \ is\ Insertion\ of\ m + n\ elements\ \&\ insertion\ of\ each\ ele$$

- The complexity of binary heap $O(n \log n)$ is more compared to leftist trees $O(\log n)$

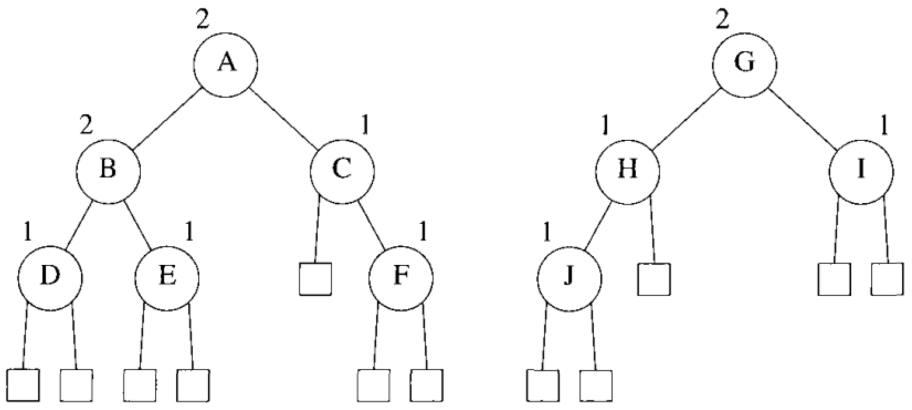
Leftist Heap

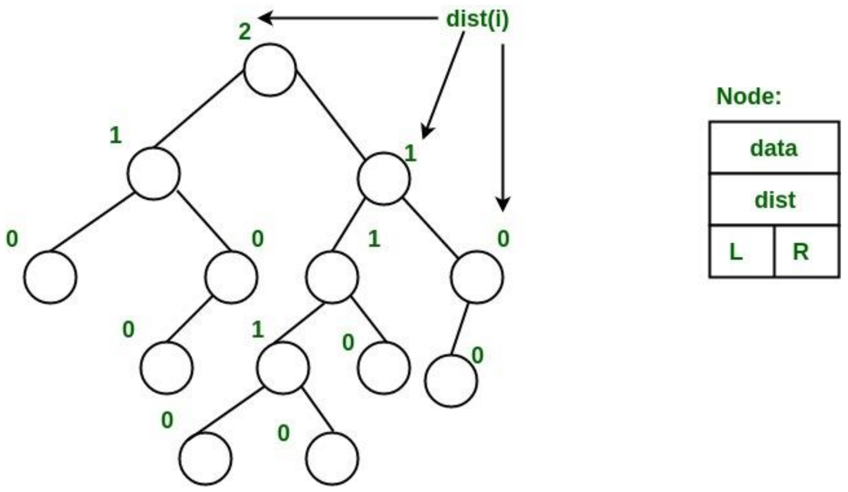
- Let n be the total number of elements in the two priority queues that are to be combined. If heaps are used to represent priority queues, then the combine operation takes O(n) time.
- Using a leftist tree, the combine operation as well as the normal priority queue operations take logarithmic time.
- Leftist trees are binary trees that prioritize balancing during insertion and deletion
- They ensure that the left subtree always has a greater or equal height compared to the right subtree.
- The leftist property allows for efficient merging of two leftist trees.
- To define a leftist tree, we need to know about the concept of an extended binary tree.

- An extended binary tree is a binary tree in which all empty binary subtrees have been replaced by a square node.
- Figure below shows two example binary trees.



- Their corresponding extended binary trees are shown below. The square nodes in an extended binary tree are called external nodes. The original (circular) nodes of the binary tree are called internal nodes.



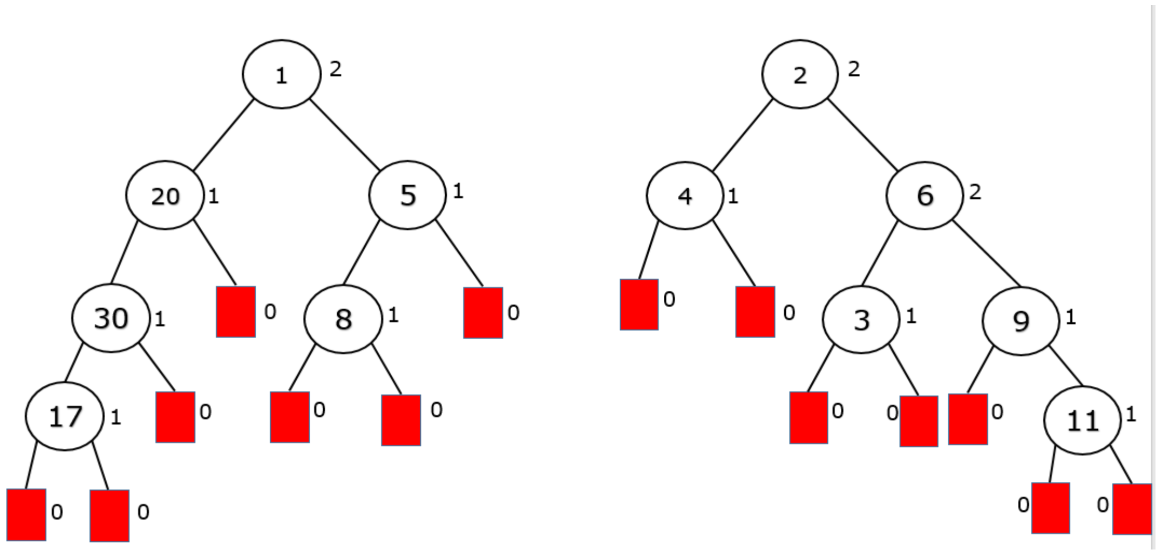


- Let X be a node in an extended binary tree. Let left_child (x) and right_child (x), respectively, denote the left and right children of the internal node x.
- Define shortest (x) to be the length of a shortest path from x to an external node. It is easy to see that shortest (x) satisfies the following recurrence

$$shortest(x) = \begin{cases} 0, & if\ x\ is\ external\ node \\ 1 + \min\{shortest(left\_child(x)), shortest(right\_child(x)), & otherwise \end{cases}$$

The number outside each internal node x of above figure is the value of shortest(x)

- Example 2



Definition: A leftist tree is a binary tree such that if it is not empty, for every internal node x

$$shortest(left\_child(x)) \geq shortest(right\_child(x))$$

Properties of Leftist trees

- The length of the rightmost path from any node x to an external node is shortest(x)
- Number of internal nodes in the subtree with root x is atleast $2^{shortest(x)} - 1$
- If the subtree with root x has n nodes then $s(x)$ is atmost $log_2(n+1)$.
    - This can be proved using 2nd property $n \geq 2^{s(x)} - 1$

$$n + 1 = 2^{s(x)}$$ taking log on both sides

$$s(x) \leq log_2(n + 1)$$

Lemma 1: Let x be the root of a leftist tree that has n (internal) nodes.

a. $n \geq 2^{shortest(x)} - 1$
b. The rightmost root to external node path is the shortest root to external node path. Its length is shortest (x)

Proof: (a) From the definition of shortest(x) it follows that there are no external nodes on the first shortest(x) levels of the leftist tree. Hence, the leftist tree has at least

$$\sum_{i=1}^{shortest(x)} 2^{i-1} = 2^{shortest(x)} - 1$$ internal nodes.

(b) This follows directly from the definition of a leftist tree.

- Leftist trees are represented with nodes that have the fields left-child, right-child, shortest, and data.

```
typedef struct {
        int key;
        /*---------------*/
} element;

struct leftist {
        struct leftist *left_child;
        element data;
        struct leftist *right_chiId;
        int shortest;
        } ;
```

Definition:

- A **min-leftist tree (max leftist tree)** is a leftist tree in which the key value in each node is smaller)than the key values in its children (if any). In other words, a min (max) leftist tree is a leftist tree that is also a min (max) tree.

- Figure below depicts two min-leftist trees. The number inside a node x is the key of the element in x and the number outside x is shortest (x). The operations insert, delete min(delete max), and combine can be performed in logarithmic time using a min (max) leftist tree.
- Examples of Leftist trees computing s(x)

Criteria to be followed for Leftist Heap

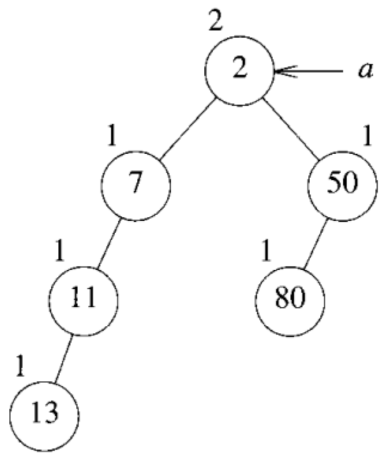
- Must be Binary tree
- For all x, Shortest(left(x) >= Shortest(right(x)
- Must follow heap property

Operations of Leftist tree

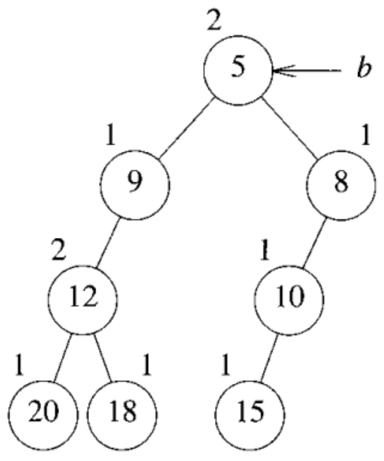
- **Merge**
- **Insertion**
- **Deletion**
- **Initialization of Heap**:

Merge Operation (Melding)

- Consider an example 1:

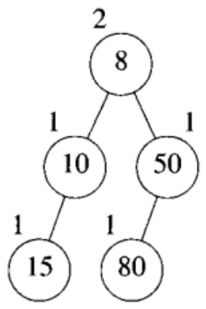


(a)

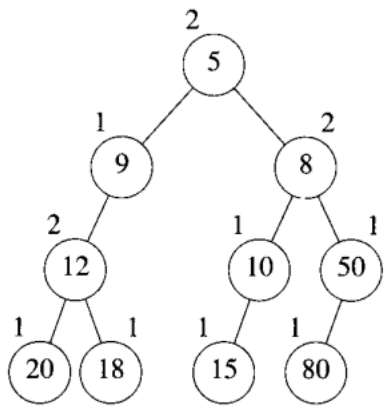


(b)

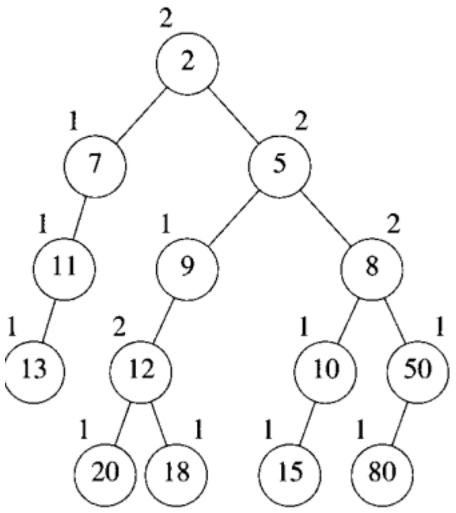
- Merge

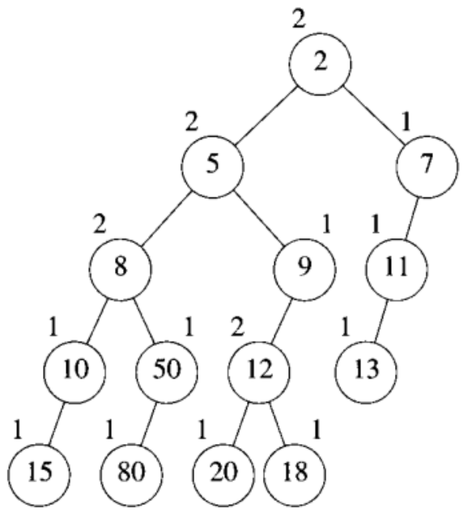


(a)

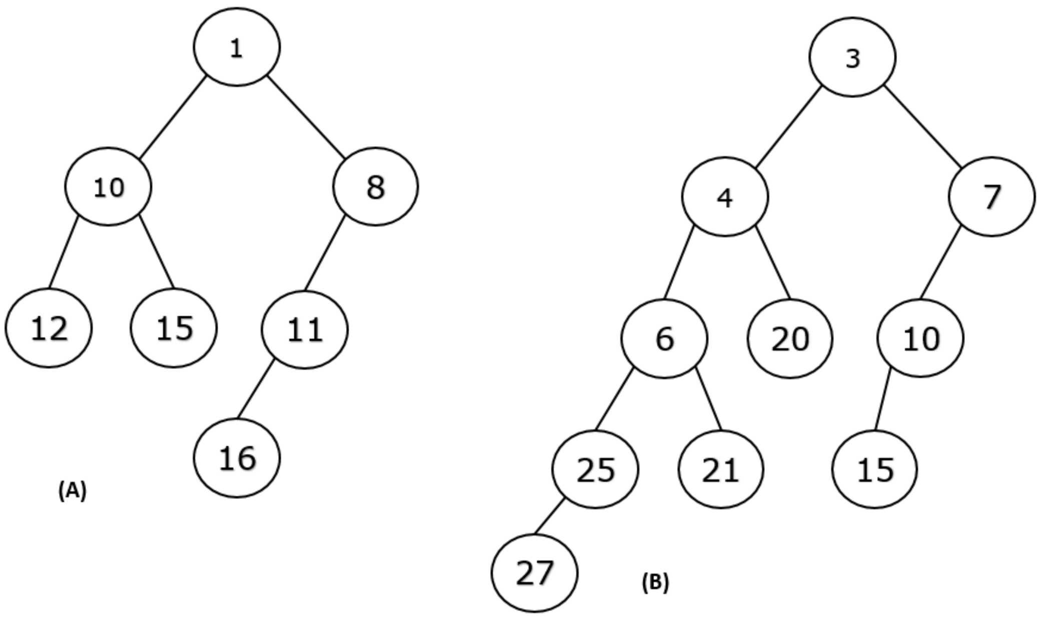


(b)

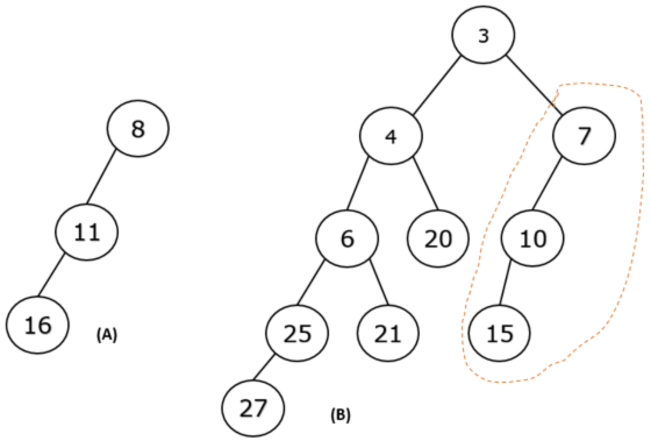




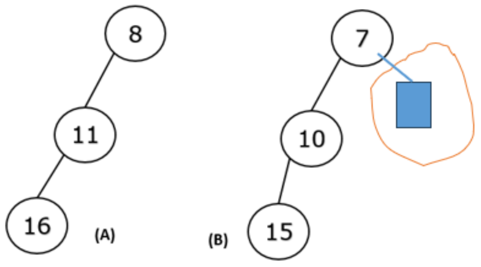
- Example 2:
    - Step 1: Consider 2 Leftist trees



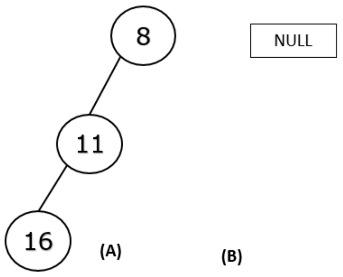
- Step 2: To apply merge for above 2 leftist trees, Find the minimum root in both leftist trees. Minimum root is 1 and pass right subtree of min root 1 along with first tree (Apply recursive call). This process will repeat until any one leftist tree without nodes i.e. reaches NULL.



- Step 3: To apply merge for above 2 leftist trees, Find the minimum root in both leftist trees. Minimum root is 3 and pass right subtree of min root 3 along with first tree

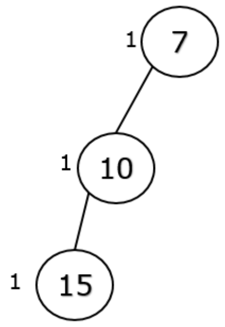


- Step 4: To apply merge for above 2 leftist trees, Find the minimum root in both leftist trees. Minimum root is 7 and pass right subtree of min root 7 along with first tree. Here, right subtree is NULL

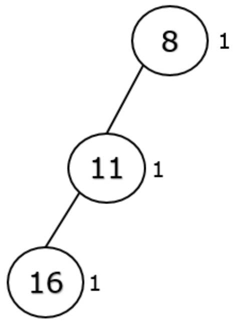


- Step 5: Since, right leftist tree is NULL (base condition is reached), return left tree as result to previous step (Step 4). Attach the result obtained and apply merge concept.
  - To apply merge concept, find the shortest(x) value for both trees.

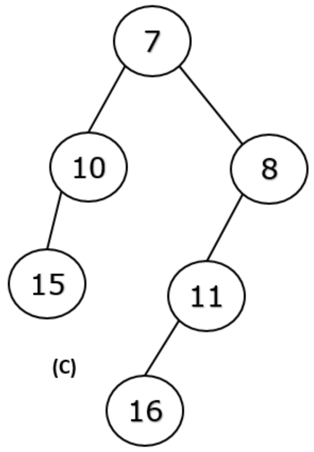
    For all x, Shortest(left(x) >= Shortest(right(x)
  - Since, Shortest(left(x) >= Shortest(right(x) then add as right child of Minimum root. After adding result is show in C
  - Pass the result to step 2 (Return of recursive call)



(A)



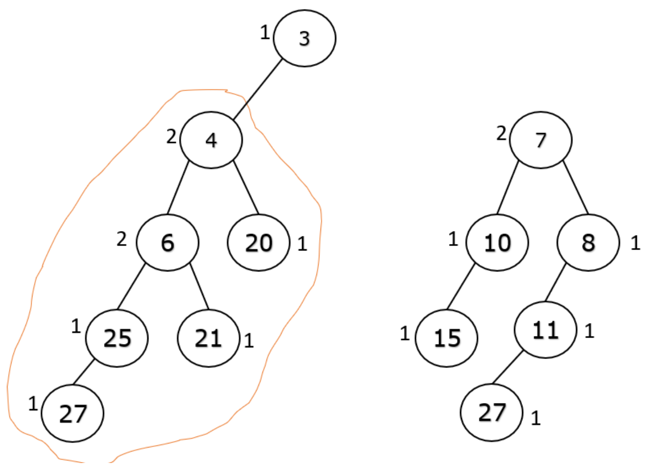
(B)

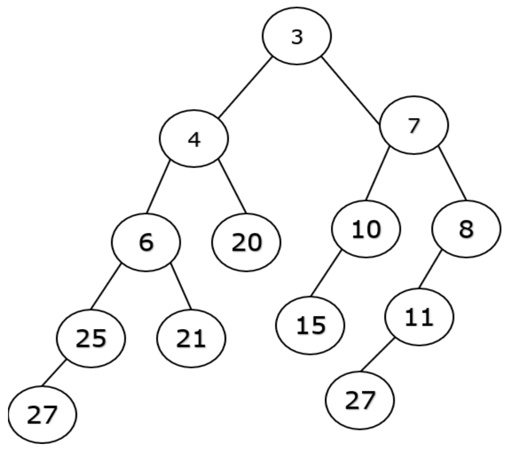


(C)

- Step 6: Consider the smallest root in step 2, merge result obtained in step 5 with step 2.
  - Merge the left subtree of step 2 (remaining part i.e left part - since right subtree is of step 2 is already processed).
  - To apply merge concept, find the shortest(x) value for both trees.

    For all x, Shortest(left(x) >= Shortest(right(x)
  - Compare the Shortest value of root left subtree i.e. 4 and root of second leftist tree. Since criteria is satisfied, add second tree (Root 7) as right child of Root 3

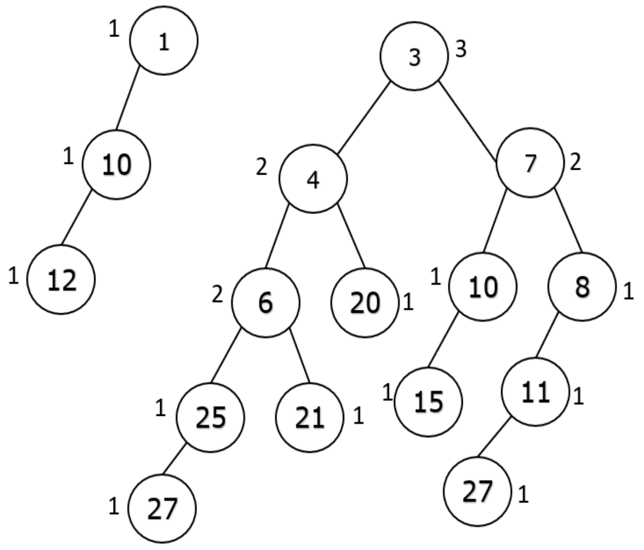


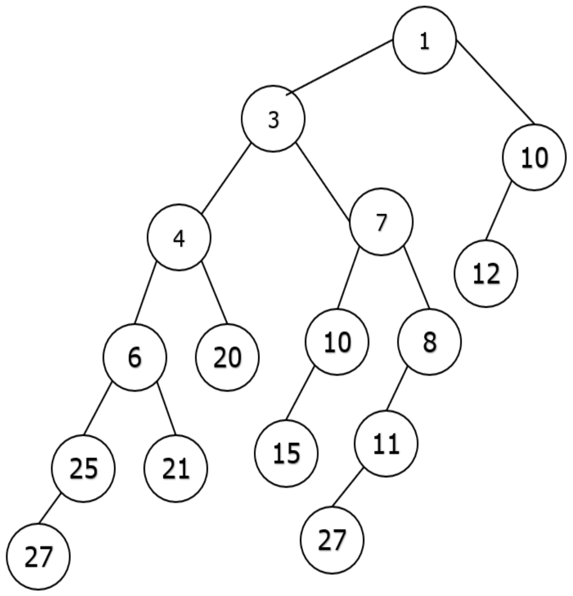


- Transfer the resultant leftist tree to step 1, ignoring transferred tree in Step1.

- Step 7: Consider the smallest root in step 1, merge result obtained in step 6 with step 1.
  - Merge the left subtree of step 2 (remaining part i.e left part, since right subtree is of step 1 is already processed).
  - To apply merge concept, find the shortest(x) value for both trees.

    For all x, Shortest(left(x) >= Shortest(right(x)
  - Compare the Shortest value of root left subtree i.e. 1 and root of second leftist tree. Since criteria is not satisfied, swap left subtree of root 1 and second leftist tree to obtain final tree.





- C Function to merge two leftist trees

```
Node *Merge(Node * root1, Node * root 2)
{
  if(root1 == NULL)                    //Base Condition
    return root2;
  if(root2 == NULL)
    return root1;

 Node *FinalRoot, *left, *RetHeap;
```

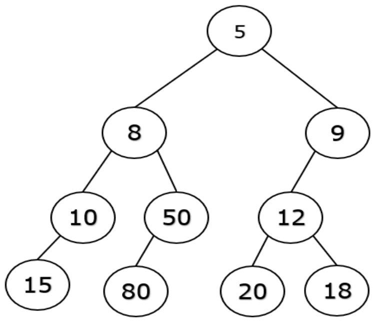
```
        if(root1→data < root2→data)
        {
          RetHeap = Merge(root1→right, root2)                    //Heap order property
          FinalRoot = root1;
        }
        else
        {
         RetHeap = Merge(root1, root2→right)                     //Heap order property
         FinalRoot = root2;
        }

        if(FinalRoot→left→Svalue >= RetHeap→Svalue)
        {
          FinalRoot→right = RetHeap;
        }
        else
        {
          Node *temp = FinalRoot→left;
          FinalRoot→left = RetHeap;
          FinalRoot→right = temp;
        }
        return FinalRoot;
    }
```

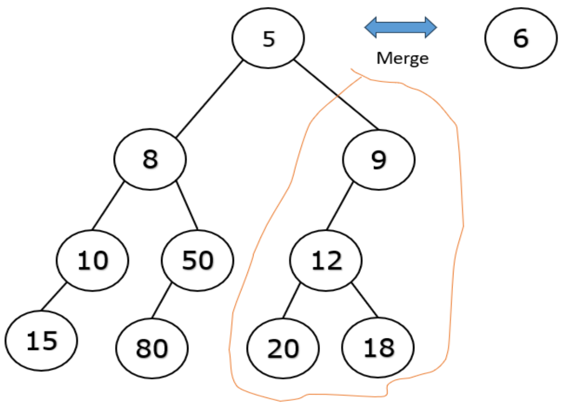
- Time Complexity : $T(n)=O(\log n)$

Insert Operation

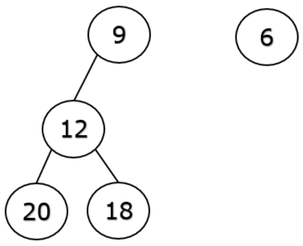
- Consider the leftist tree



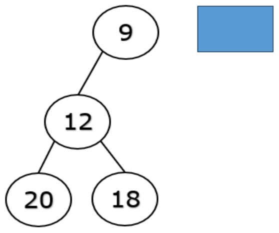
- Step 1: Insert node 6, Apply Merge operations



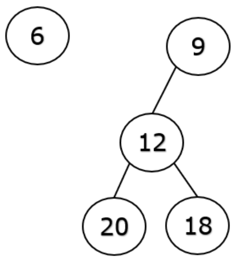
- Step 2: Find the minimum root and pass its right subtree along with 2nd tree for further



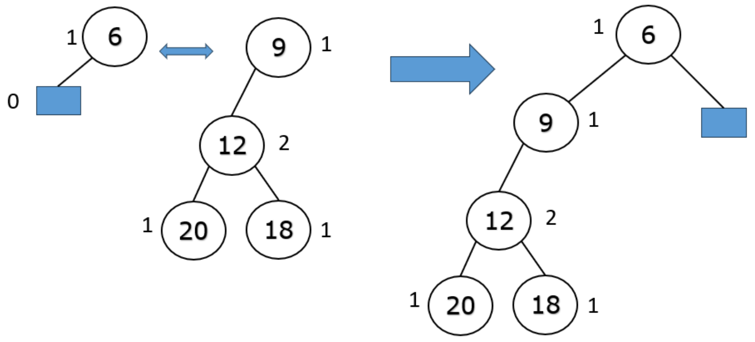
- Step 3: Smallest root in both leftist trees is 6, consider its right subtree of 6 along with remaining leftist tree.



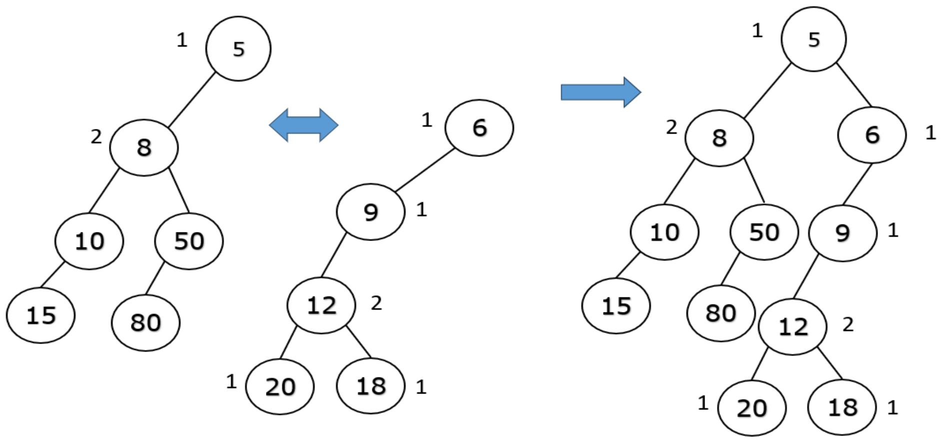
- Step 4: Since right leftist tree (root2) is empty return left subtree as result to the previous step. The remaining tree and result of step 3 is



- Find the S value to merge these two leftist trees. The S value of left child of smaller root (Root 6) is not greater the Root of right leftist tree so, swap and the result is



- Step 5: Pass the resultant tree to Step 1 to merge. Find the Shortest value for both leftist trees. Smaller root is 5, check the Shortest value of Left of Root 5 with result obtained from step 4 i.e shortest value of Root 6. S(8)>=S(6) so, add Root 6 as right child of 5.



- Time Complexity :
  - For insertion Insert O(1) and Merge O(log n) = O(log n)

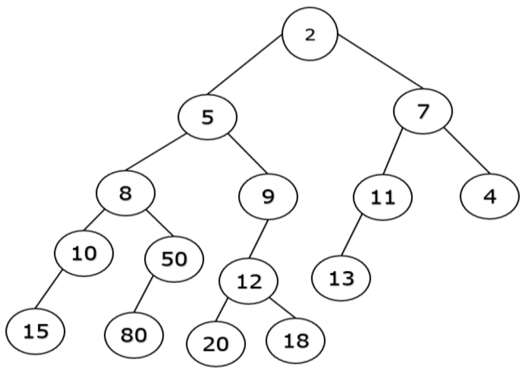
Initialize Heap

6	**2**	**9**	**8**	**3**	**4**	**11**	**18**	**7**	**24**	**1**	**5**

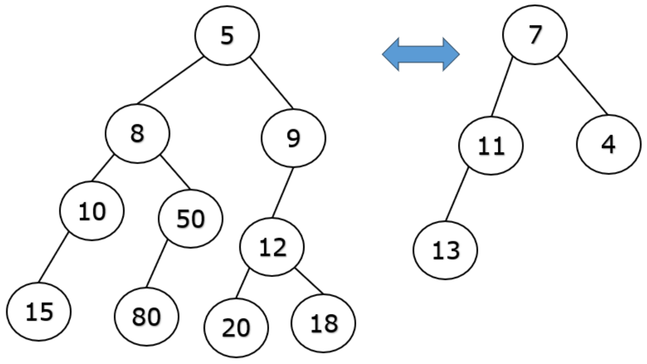
- Create a Empty leftist heap.
- Read the elements one by one and apply merge
- Complexity will be
    - For each insertion O(log n) and Total N insertions = O (n log n)

Delete Operation

- Delete the root element. We will get 2 leftist tree and apply merge for these leftist trees
- Complexity will be
    - For Deletion O(1) and Merge O(log n) = O(log n)
- Consider an example



- Delete Root node, we will get 2 leftist trees, apply merge



Let $n=4$ and $(a_1, a_2, a_3, a_4) = (10, 15, 20, 25)$. Let $(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$ and $(q_0, q_1, q_2, q_3, q_4) = (2, 3, 1, 1, 1)$. The p's and q's have been multiplied by 16 for convenience. Initialy, $w_{ii} = q_i$, $c_{ii} = 0$, and $r_{ii} = 0$, $0 \le i \le 4$.

Sol:

$$w(i,j) = w(i,j-1) + p(j) + q(j)$$

$$c(i,j) = \min_{i<k\le j} \{c(i,k-1) + c(k,j)\} + w(i,j)$$

$$r(i,j) = k$$

	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
$j-i=0$	$w_{00}=2$, $c_{00}=0$, $r_{00}=0$	$w_{11}=3$, $c_{11}=0$, $r_{11}=0$	$w_{22}=1$, $c_{22}=0$, $r_{22}=0$	$w_{33}=1$, $c_{33}=0$, $r_{33}=0$	$w_{44}=1$, $c_{44}=0$, $r_{44}=0$
$j-i=1$	$w_{01}=8$, $c_{01}=8$, $r_{01}=1$	$w_{12}=7$, $c_{12}=7$, $r_{12}=2$	$w_{23}=3$, $c_{23}=3$, $r_{23}=3$	$w_{34}=3$, $c_{34}=3$, $r_{34}=4$	
$j-i=2$	$w_{02}=12$, $c_{02}=19$, $r_{02}=1$	$w_{13}=9$, $c_{13}=12$, $r_{13}=2$	$w_{24}=5$, $c_{24}=8$, $r_{24}=3$		
$j-i=3$	$w_{03}=14$, $c_{03}=25$, $r_{03}=2$	$w_{14}=11$, $c_{14}=19$, $r_{14}=2$			
$j-i=4$	$w_{04}=16$, $c_{04}=32$, $r_{04}=2$				

Initial Conditions

$W(i,i) = q_i$

$C(i,i) = 0$

$r(i,i) = 0$

$W(0,0) = q_0$

$W(0,0) = 2$

$C(0,0) = 0$

$\boxed{r(0,0) = 0}$

$W(1,1) = q_1$

$W(1,1) = 3$

$c(1,1) = 0$

$\boxed{r(1,1) = 0}$

$W(2,2) = q_2$

$W(2,2) = 1$

$C(2,2) = 0$

$\boxed{r(2,2) = 0}$

$W(3,3) = q_3$

$W(3,3) = 1$

$C(3,3) = 0$

$\boxed{r(3,3) = 0}$

$W(4,4) = q_4$

$W(4,4) = 1$

$C(4,4) = 0$

$\boxed{r(4,4) = 0}$

$W_{01} = w(i, j-1) + p(j) + q(j)$

$= W(0,0) + p(1) + q(1)$

$= 2+3+3$

$\boxed{W_{01} = 8}$

$C_{01} = \min_{i<k\leq j} \{ c(i, k-1) + c(k, j) \} + w(i, j)$

$= \min_{0<k\leq 1} \{ c(0,0) + c(1,1) \} + w(0,1)$

$k=1$

$= \min\{0+0\} + 8$

$= 0+8$

$\boxed{C_{01} = 8}$

$r(0,10) = k$

$\boxed{r(0,1) = 1}$

$W(1,2) = w(1,1) + p(2) + q(2)$

$= 3+3+1$

$\boxed{W(1,2) = 7}$

$C(1,2) = \min_{1<k\leq 2} \{ c(1,1) + c(2,2) \} + w(1,2)$

$k=2$

$= \min\{0+0\}+7$

$\boxed{C_{12} = 7}$

$\boxed{\begin{array}{l} r_{12} = K \\ r_{12} = 2 \end{array}}$

. . . . . .

$W_{02} = w(i, j-1) + p(j) + q(j)$

$= w(0,1) + p(2) + q(2)$

$= 8 + 3 + 1$

$\boxed{W_{02} = 12}$

$C_{02} = \min\limits_{0<K\leq 2} \{ c(0,0) + c(1,2) \} + w(0,2)$

$K=1$

$= \min\{0+7\} + 12$

$= 7+12$

$\boxed{C_{02} = 19}$

$\boxed{\begin{array}{l} r_{02} = k \\ r_{02} = 1 \end{array}}$

$W(1,3) = w(1,2) + p(3) + q(3)$

$= 7+1+1$

$\boxed{W_{13} = 9}$

$$C(1,3) = \min_{1 \leq k \leq 3} \{ C(1,1) + C(2,3) \} + W(1,3)$$

When $k=2$

$k=3$

$$= \min_{k=2} \{0+3\}+9$$

$$C(1,3) = \min_{k=2} \{12\}$$

$$C(1,3) = \min_{k=3} \{ C(1,2)+C(2,3) \} + W(1,3)$$

$$= \min \{7+3\}+9 \quad = 19$$

When $k=2$ $\min \begin{cases} 12 \\ 19 \end{cases}$ take the minimum

$k=3$

Value & thus $\boxed{C_{13} = 12}$

$$\boxed{\begin{matrix} r_{13} = k \\ r_{13} = 2 \end{matrix}}$$

- - - - - - -

thus Construct the tree

$$\begin{tikzcd}
 & r(i,j)=k \arrow[dl, dash] \arrow[dr, dash] & \\
r(i,k-1) & & r(k,j)
\end{tikzcd}$$

$$\begin{tikzcd}[column sep=tiny]
 & & & r(0,4)=2 \arrow[dll, dash] \arrow[drr, dash] & & & & \\
 & r(0,1)=1 \arrow[dl, dash] \arrow[dr, dash] & & & & r(2,4)=3 \arrow[dl, dash] \arrow[dr, dash] & & \\
r(0,0)=0 & & r(1,1)=0 & & r(2,2)=0 & & r(3,4)=4 \arrow[dl, dash] \arrow[dr, dash] & \\
 & & & & & r(3,3)=0 & & r(4,4)=0
\end{tikzcd}$$

thus the optimal binary search tree is

$\Downarrow$

$$\begin{tikzcd}
 & 15 \arrow[dl, dash] \arrow[dr, dash] & \\
10 & & 20 \arrow[dr, dash] & \\
 & & & 25
\end{tikzcd}$$