## UNIT – IV

Searching- Linear Search,Binary Search, Hashing-Hash functions,Collision-Handling schemes,Hashing in java.util,Dictionary ADT,Linear list representation,Skip list representation,Hash table representation,Comparison of Searching methods.
Sorting- Bubble Sort,Insertion Sort,Shell sort,Heap Sort,Radix Sort,Quick sort,Merge sort, Comparison of Sorting methods,Sorting in java.util.

# SEARCHING

Searching is a technique that searches an element in a given list of elements. The two popular search algorithms are:

1. Sequential Search or Linear Search
2. Binary Search

## LINEAR SEARCH

This is a simplest technique. Assuming the search list is an array or a linked list, the approach is to iterate through all the elements until a match is found.

We begin search by comparing the first element of the list with search element. If it matches, the search ends. Otherwise, we will move to next element and compare. In this, the search element is compared with all the elements until a match occurs or when there are no elements left to be compared (i.e, search element is absent in the list). This method requires no ordering of elements in the list.

For example, consider the following list of elements:

8 5 9 15 43 2 17 65 23 11

Suppose we want to search the element 17(i.e. key=17), we first compare the key with the first element which is 8. Since both are not matching, we have to move to next element and compare. By repeating this process we will find the key after 7 comparisons. That means, the desired key is at $7^{th}$ position in the list. On the other hand, suppose the key is 31, the comparison test fails from first element resulting in an unsuccessful search. This is because the key is not in the given list.

Time complexity involved to search an item matching a key in the input list for the worst, best and average cases.

**Best Case:** In which the first comparison returns a match, it requires a single comparison and hence it is O(1).
**Worst Case:** If there are N items in the list, then it is obvious that in the worst case(i.e when there are no search element in the list) N comparisons are required. Hence the worst-case performance of this algorithm is roughly proportional to N and represented as O(N).

**Average Case:** The average time depends on the probability that the key will be found in the list. Thus the average case roughly requires N/2 comparisons to search the element. That means, the average time, as in worst-case, is proportional to N and hence it is O(N).

```
void linsearch(int list[],int size,int key)
{
int i,f=0;

for(i=0;i<size;i++)
{
if(key==list[i])
{
System.out.println("THE ELEMENT IS FOUND AT POSITION:" +(i+1));
f++;
}
}
if(f==1)
return;
else
System.out.println("THE ELEMENT IS NOT FOUND");
}
```

## BINARY SEARCH

Binary search performs well when compared to the linear search. However, the constraint of binary search to work is the items in the list must be in a sorted order(either in ascending or descending order). This is prerequisite condition for binary search. Note that this is not at all required in the case of a sequential search. However, the binary search cannot be applied always because of the sorted order limitation. If the list to be search for a specific item is not in the sorted order, binary search will not work. The complexity of binary search is O(log n).

The binary search uses an approach called as divide and conquer. The logic of divide and conquer is to divide the problem into different subtasks and solve it with smaller set of the data.

Suppose in a given list of elements L in ascending order, we wish to search for the position of an element equal to S. Let minindex is the first element position of given list and maxindex is the last element position in the list. The binary search begins at the mid positiom computed by integer division ((minindex+maxindex)/2), and compare it with the value X. If they are equal, the search is successful and the mid position is returned. Otherwise, if X is lower than the item at midposition, the search continues in the first half of the list, and if X greater than the item at the midposition , the search continues in the second half of the list. The process repeats until the item is found, or there are no elements in the sublist. The terminating condition is found when the minindex exceeds the maxindex.

The best, worst and average cases fort Binary Search are:

**BestCase:** item in the middle. Binary Search examines only one (middle) position.

**WorstCase:** item in the lost possible division. The maximal number of items in an array of length N can be divided is $\log_2 N$.

**AverageCase:** somewhere in between ; $1/2\text{Log}_2N$

**Example:**

Consider a list A={7, 12, 23, 45, 67, 89, 90}. Observe that the elements in the list are in sorted order. Here the size of the list is 7. The minimum index of the list, say minindex, is 1, and the maximum index in the list, say maxindex, is 7. If we want to search an element N=67 in the list A using the binary search algorithm , the steps to be followed are given as follows:

1. Compare the element with middle element of the list. That is, middle element index, midposition=(minindex+maxindex)/2=(1+7)/2=4. So compare search element with element 4 in the list.

   67 = A[4] (67 >45)

2. The element 4 in the list=45 is less than the search element. As the elements are in the sorted order and the search element is greater than the element 4, the search element must be present only after 45 in the list (second half). Now search for the element in the list by considering only the next elements after 45 in the list (excluding the middle element i.e 45).

   Consider {67, 89, 90}

3. Now the minindex is 4+1=5 and maxindex is 7.

4. Now the item to be compared is middle element=(5+7)/2=12/2=6.

5. Compare the search element N=67 with the middle element=89.

   67 = A[6] (67 < 89)

6. As the search element is less than the middle element 89, the search element might be present only before the element 89 in the list. That is between minindex element and the midposition element.

7. Now search for the element between minindex and midposition element by considering the middle element index as the max index.

8. Now the maxindex=5 and the minindex=5.

9. As the minindex and maxindex are same there is only one element to consider to be compared with the search element.

10. As the element remaining to be considered for comparison with the search element is same i.e. 67. The search is successful.

The binary search return 5, which is the element(=67) position in the list.

```
int binsearch(int list[],int size,int key)
{
int top,bottom,middle;
top=0;
bottom=size-1;
while(top<=bottom)
{
middle=(top+bottom)/2;
if(key==list[middle])
return middle;

else

if(key<list[middle])
bottom=middle-1;

else
top=middle+1;
}
return(-1);

}
```

# DICTIONARY ADT

Dictionary is a collection of pairs of key and value where every value is associated with the corresponding key.
Basic operations that can be performed on dictionary are:

1. Insertion of value in the dictionary
2. Deletion of particular value from dictionary
3. Searching of a specific value with the help of key

**Linear List Representation**

The dictionary can be represented as a linear list. The linear list is a collection of pair and value. There are two method of representing linear list.

1. Sorted Array- An array data structure is used to implement the dictionary.
2. Sorted Chain- A linked list data structure is used to implement the dictionary.

**Structure of linear list for dictionary:**

```
class node
{
int key;
int value;
}*next,*head;
```

**Insertion of new node in the dictionary:**

Consider that initially dictionary is empty then

head = NULL

We will create a new node with some key and value contained in it.

New

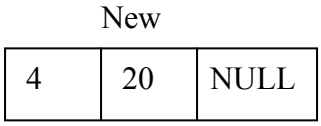
| | | |
|---|---|---|
| 1 | 10 | NULL |

Now as head is NULL, this new node becomes head. Hence the dictionary contains only one record. this node will be ‘curr’ and ‘prev’ as well. The ‘cuur’ node will always point to current visiting node and ‘prev’ will always point to the node previous to ‘curr’ node. As now there is only one node in the list mark as ‘curr’ node as ‘prev’ node.

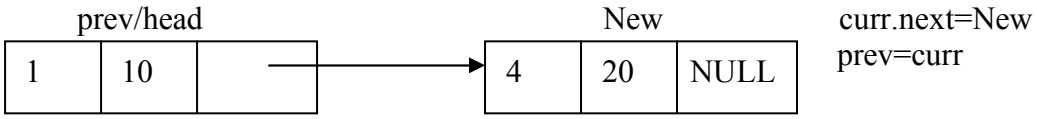
New/head/curr/prev

| | | |
|---|---|---|
| 1 | 10 | NULL |

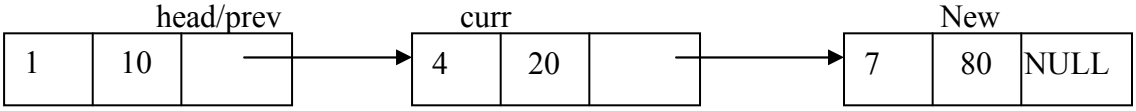
Insert a record, key=4 and value=20,




Compare the key value of 'curr' and 'New' node. If New.key > Curr.key then attach New node to 'curr' node.

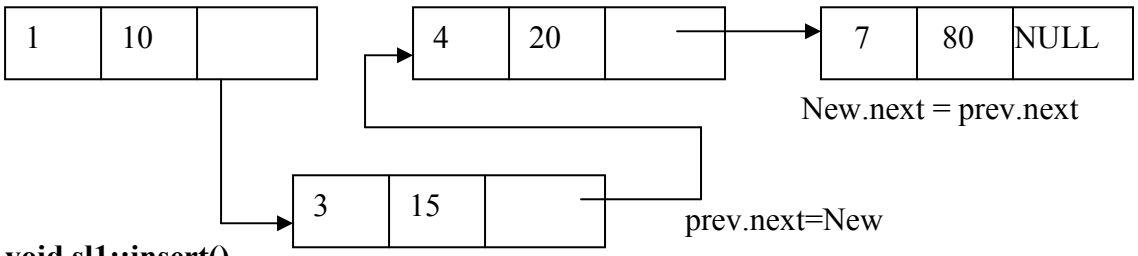



Add a new node <7,80> then




If we insert <3,15> then we have to search for it proper position by comparing key value.

(curr.key < New.key) is false. Hence else part will get executed.



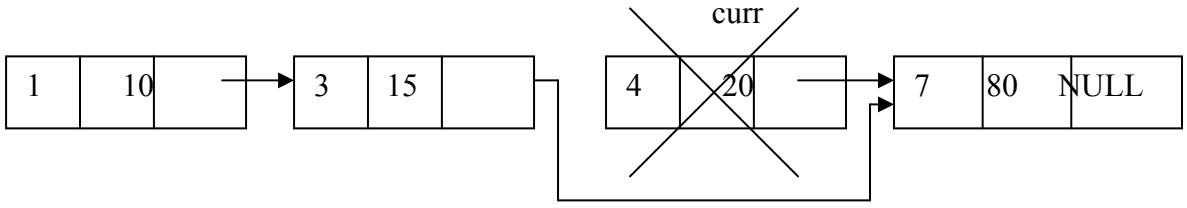

```
void sl1::insert()
{
node *New, *curr, *prev;
int k,val;
New.key=k;
New.value=val;
New.next=NULL;
if(head==NULL)        //creating head node
{
head=New;
```

```
curr=head;
prev=curr;
}
else
{
curr=head;
prev=prev;
while((curr.key<New.key)&&(curr.next!=NULL))
{
prev=curr;
curr=curr.next;
}
if(curr.next==NULL)
{
if(curr.key<New.key)
{
curr.next=New;
prev=curr;
}
else
{
New.next=prev.next;
prev.next=New;
}
}
else
{
New.next=prev.next;
prev.next=New;
}
}
}
```

**The delete operation**

Case 1:

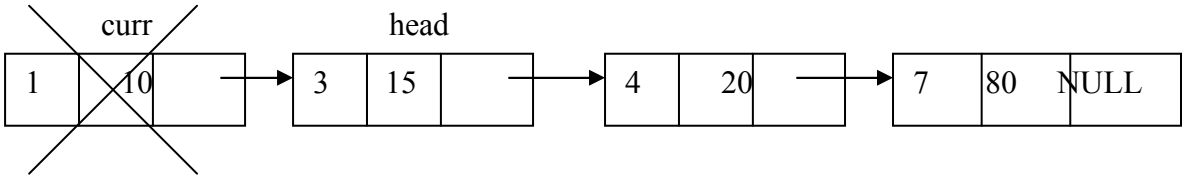
Initially assign ‘head’ node as ‘curr’ node
Then ask for a key value of the node which is to be deleted. Then starting from head node key value of each jode is cked and compared with the desired node’s key value. We will get node which is to be deleted in variable ‘curr’. The node given by variable ‘prev’ keeps track of previous node of ‘cuu’ node. For eg, delete node with key value 4 then



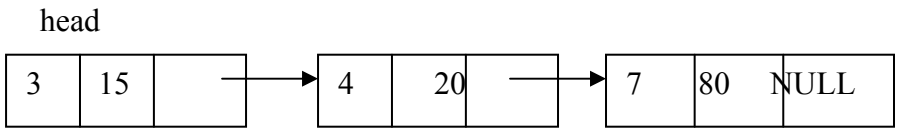

Case 2:

If the node to be deleted is head node
i.e.. if(curr==head)

Then, simply make 'head' node as next node and delete 'curr'




Hence the list becomes



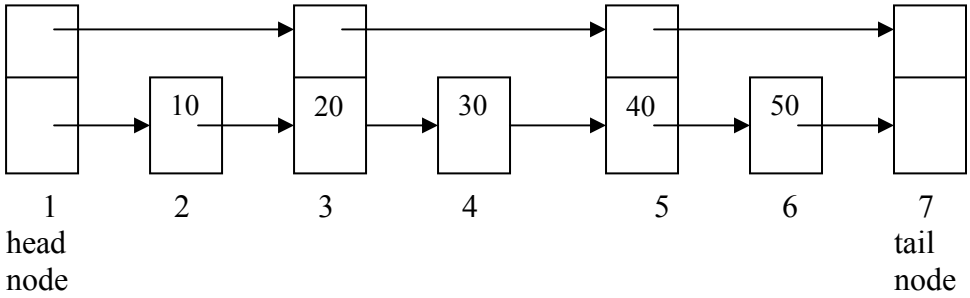

```
void sl1::del()
{
node *curr, *prev;
int k;
curr=head;
while(curr!=NULL)
{
if(curr.key==k)              //traverse till required node to delete
break;
prev=curr;
curr=curr.next;
}
if(curr==NULL)
cout<<"Node not found";
else
{
if(curr==head)
head=curr.next;
else
```

```
prev.next=curr.next;
}
}
```

## SKIP LIST REPRESENTATION

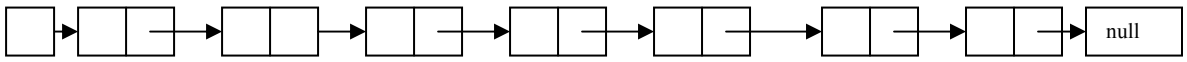
Skip list is a variant list for the linked list. Skip lists are made up of a series of nodes connected one after the other. Each node contains a key and value pair as well as one or more references, or pointers, to nodes further along in the list. The number of references each node contains is determined randomly. This gives skip lists their probabilistic nature, and the number of references a node contains is called its node level.

There are two special nodes in the skip list one is head node which is the starting node of the list and tail node is the last node of the list.

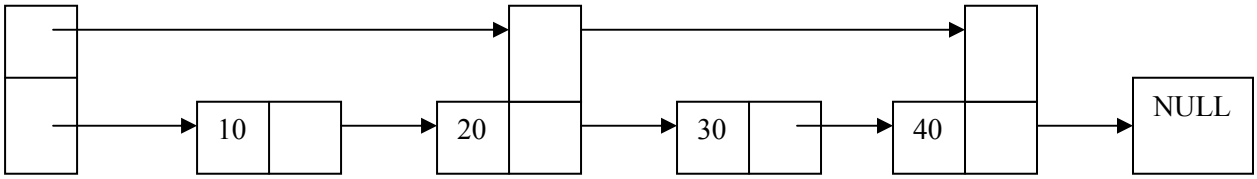



The skip list is an efficient implementation of dictionary using sorted chain. This is because in skip list each node consists of forward references of more than one node at a time.

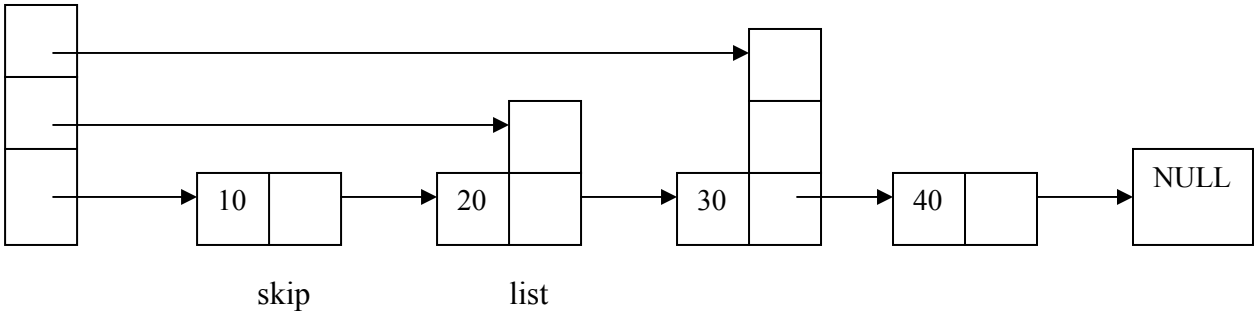
Eg:



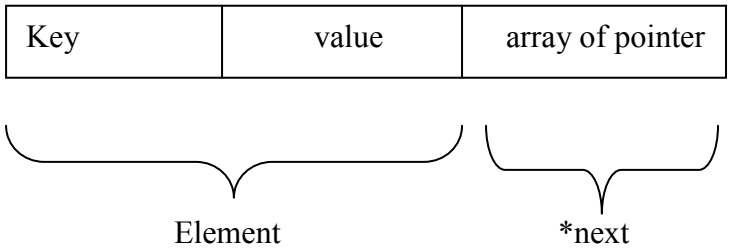

Now to search any node from above given sorted chain we have to search the sorted chain from head node by visiting each node. But this searching time can be reduced if we add one level in every alternate node. This extra level contains the forward pointer of some node. That means in sorted chain come nodes can holds pointers to more than one node.




If we want to search node 40 from above chain there we will require comparatively less time. This search again can be made efficient if we add few more pointers forward references.




The individual node looks like this:




**Searching:**

The desired node is searched with the help of a key value.

```
boolean retrieve(int key_val)
{
node *current = head;
int level=max_level;
if(current == NULL)
return false;
while(level >= 0)
{
while(current.next[level] != NULL && current.next[level].data <= key_val)
current = current.next[level];
if(current.data == key_val)
return true;
level--
}
return false;
}
```

Searching for a key within a skip list begins with starting at header at the overall list level and moving forward in the list comparing node keys to the key_val. If the node key is less than the key_val, the search continues moving forward at the same level. If o the other hand, the node key is equal to or greater than the key_val, the search drops one level and continues forward. This process continues until the desired key_val has been found if it is present in the skip list. If it is not, the search will either continue at the end of the list or until the first key with a value greater than the search key is found.

**Insertion:**

There are two tasks that should be done before insertion operation:

1. Before insertion of any node the place for this new node in the skip list is searched. Hence before any insertion to take place the search routine executes. The last[] array in the search routine is used to keep track of the references to the nodes where the search, drops down one level.
2. The level for the new node is retrieved by the routine randomelevel()

```
temp[i] = head;
x= new SkipNode<T> (lvl, value)
for(int i=0; i<=lvl; i++)
{
x.forward[i] = temp[i].forward[i];
temp[i].forward[i] = x;
}
```

**Determining the level of each node:**

```
template <class K, class E>
int skipLst<K,E>::randomlevel()
{
int lvl=0;
while(rand() <= Lvl_No)
lvl=lvl+1;
if(lvl<=MaxLvl)
return lvl;
else
return MaxLvl;
}
```

**Deletion:**

First of all, the deletion makes use of search algorithm and searches the node that is to be deleted. If the key to be deleted is found, the node containing the key is removed.

```
template<class K, class E>
void skipLst<K,E>::delet(K& Key_val)
{
if(key_val>=tailKey)
return;

skipNode<K,E>* temp = search(Key_val);
if(temp.elemnt.key != Key_val)
return;

for(int i=0;i<=levels;i++)
{
if(last[i].next[i] == temp)
last[i]=>next[i] = temp.next[i];
}

while(level>0 && header.next[level] == tail)
levels--;
delete temp;
len--;
}
```

## HASH TABLE REPRESENTATION

- Hash table is a data structure used for storing and retrieving data very quickly. Insertion of data in the hash table is based on the key value. Hence every entry in the hash table is associated with some key.
- Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is then dependent upon the size of the hash table.
- The effective representation of dictionary can be done using hash table. We can place the dictionary entries in the hash table using hash function.

### HASH FUNCTION

- Hash function is a function which is used to put the data in the hash table. Hence one can use the same hash function to retrieve the data from the hash table. Thus hash function is used to implement the hash table.
- The integer returned by the hash function is called hash key.

For example: Consider that we want place some employee records in the hash table The record of employee is placed with the help of key: employee ID. The employee ID is a 7 digit number for placing the record in the hash table. To place the record 7 digit number is converted into 3 digits by taking only last three digits of the key.
If the key is 496700 it can be stored at $0^{th}$ position. The second key 8421002, the record of those key is placed at $2^{nd}$ position in the array.

Hence the hash function will be-
H(key) = key%1000

Where key%1000 is a hash function and key obtained by hash function is called hash key.

- **Bucket and Home bucket:** The hash function H(key) is used to map several dictionary entries in the hash table. Each position of the hash table is called bucket.

The function H(key) is home bucket for the dictionary with pair whose value is key.

**TYPES OF HASH FUNCTION**

There are various types of hash functions that are used to place the record in the hash table-

1. **Division Method:** The hash function depends upon the remainder of division.

Typically the divisor is table length.

For eg; If the record 54, 72, 89, 37 is placed in the hash table and if the table size is 10 then

h(key) = record % table size

4=54%10

2=72%10

9=89%10

7=37%10

**2. Mid Square:**

In the mid square method, the key is squared and the middle or mid part of the result is used as the index.
If the key is a string, it has to be preprocessed to produce a number.
Consider that if we want to place a record 3111 then

$3111^2 = 9678321$

for the hash table of size 1000
H(3111) = 783 (the middle 3 digits)

**3. Multiplicative hash function:**

The given record is multiplied by some constant value. The formula for computing the hash key is-

H(key) = floor(p *(fractional part of key*A)) where p is integer constant and A is constant real number.

Donald Knuth suggested to use constant A = 0.61803398987

If key 107 and p=50 then

H(key) = floor(50*(107*0.61803398987))
= floor(3306.4818458045)
= 3306

At 3306 location in the hash table the record 107 will be placed.

**4. Digit Folding:**

The key is divided into separate parts and using some simple operation these parts are combined to produce the hash key.
For eg; consider a record 12365412 then it is divided into separate parts as 123 654 12 and these are added together

H(key) = 123+654+12
= 789
The record will be placed at location 789

**5. Digit Analysis:**

The digit analysis is used in a situation when all the identifiers are known in advance. We first transform the identifiers into numbers using some radix, r. Then examine the digits of each identifier. Some digits having most skewed distributions are deleted. This deleting of digits is

continued until the number of remaining digits is small enough to give an address in the range of the hash table. Then these digits are used to calculate the hash address.

## COLLISION

the hash function is a function that returns the key value using which the record can be placed in the hash table. Thus this function helps us in placing the record in the hash table at appropriate position and due to this we can retrieve the record directly from that location. This function need to be designed very carefully and it should not return the same hash key address for two different records. This is an undesirable situation in hashing.

**Definition:** The situation in which the hash function returns the same hash key (home bucket) for more than one record is called collision and two same hash keys returned for different records is called synonym.

Similarly when there is no room for a new pair in the hash table then such a situation is called overflow. Sometimes when we handle collision it may lead to overflow conditions. Collision and overflow show the poor hash functions.

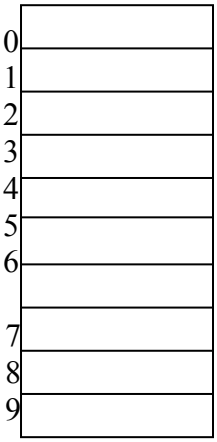
For example,

Consider a hash function.

H(key) = recordkey%10 having the hash table size of 10

The record keys to be placed are

131, 44, 43, 78, 19, 36, 57 and 77




Now if we try to place 77 in the hash table then we get the hash key to be 7 and at index 7 already the record key 57 is placed. This situation is called **collision**. From the index 7 if we look for next vacant position at subsequent indices 8.9 then we find that there is no room to place 77 in the hash table. This situation is called **overflow**.

## COLLISION RESOLUTION TECHNIQUES

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

There are two methods for detecting collisions and overflows in the hash table:

1. Chaining
2. Open addressing (linear probing)

The two more difficult collision handling techniques are-

1. Quadratic probing
2. Double hashing

## 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.

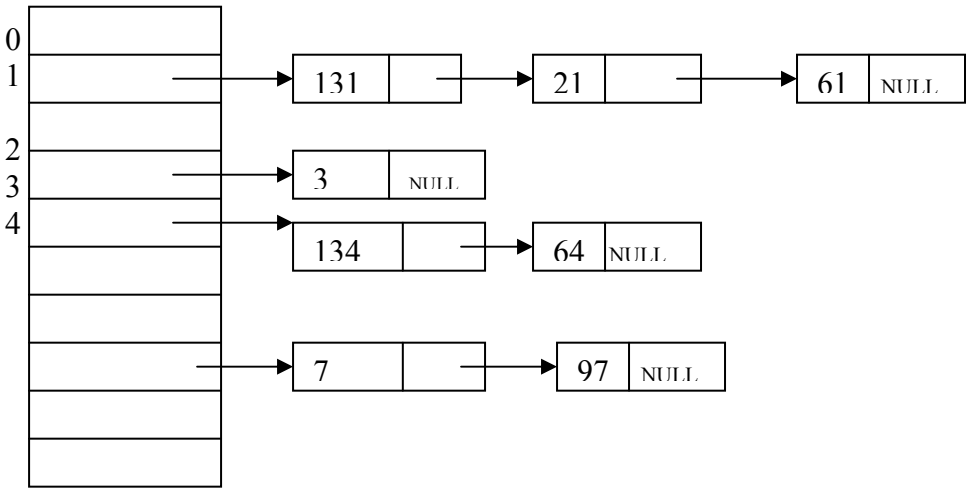
For eg;

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(key) = 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(key) = key % tablesize
H(key) = key % 10

For instance the element 131 can be placed at

H(key) = 131 % 10
= 1

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(key)=21%10
H(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 is the location over there is empty 29 will be placed at $0^{th}$ 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

| Key |
|---|
| 39 |
| 29 |
| 8 |
| |
| |
| |
| |
| |
| 18 |
| 19 |

this cluster problem can be solved by quadratic probing.

## 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.

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

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

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.

$H_i$ (key) = (Hash(key) + $i^2$) % m

Consider i = 0 then

$(17 + 0^2) \% 10 = 7$

$(17 + 1^2) \% 10 = 8$, 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

| Key |
|---|
| 90 |
| 11 |
| 22 |
| |
| |
| 55 |
| |
| 37 |
| |
| |

Now to place 87 we will use quadratic probing.

| Key |
|---|
| 90 |
| 11 |
| 22 |
| |
| |
| 55 |
| |
| 87 |
| 37 |
| 17 |
| 49 |

$(87 + 0)\ \%\ 10 = 7$
$(87 + 1)\ \%\ 10 = 8$… but already occupied
$(87 + 2^2)\ \%\ 10 = 1$.. already occupied
$(87 + 3^2)\ \%\ 10 = 6$

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

| Key |
|---|
| 90 |
| 11 |
| 22 |
| |
| |
| 55 |
| 87 |
| 37 |
| 17 |
| 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

$H_1(key) = key\ mod\ tablesize$

$H_2(key) = M - (key\ mod\ M)$

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 $H_1(key)$.
Insert 37, 90, 45, 22

| Key |
|---|
| 90 |
| |
| 22 |
| |
| |
| 45 |
| |
| 37 |
| |
| 49 |

$H_1(37) = 37 \% 10 = 7$
$H_1(90) = 90 \% 10 = 0$
$H_1(45) = 45 \% 10 = 5$
$H_1(22) = 22 \% 10 = 2$
$H_1(49) = 49 \% 10 = 9$

| Key |
|---|
| 90 |
| 17 |
| 22 |
| |
| |
| 45 |
| |
| 37 |
| |
| 49 |

Now if 17 to be inserted then

$H_1(17) = 17 \% 10 = 7$
$H_2(key) = M - (key \% M)$

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

Hence M = 7

$H_2(17) = 7-(17 \% 7)$
$= 7 - 3 = 4$

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

Now to insert number 55

$H_1(55) = 55 \% 10 = 5 \longrightarrow$ Collision

$H_2(55) = 7-(55 \% 7)$
$= 7 - 6 = 1$

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 |
| 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 is 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.,

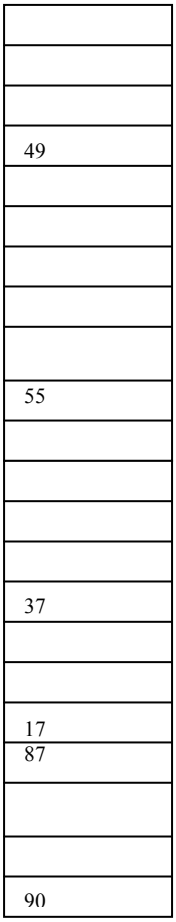
**H(key) = key mod tablesize**

37 % 10 = 7
90 % 10= 0
55 % 10 = 5
22 % 10 = 2
17 % 10 = 7 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(key) key mod 23

37 % 23 = 14
90 % 23 = 21
55 % 23 = 9
22 % 23 = 22
17 % 23 = 17
49 % 23 = 3
87 % 23 = 18




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

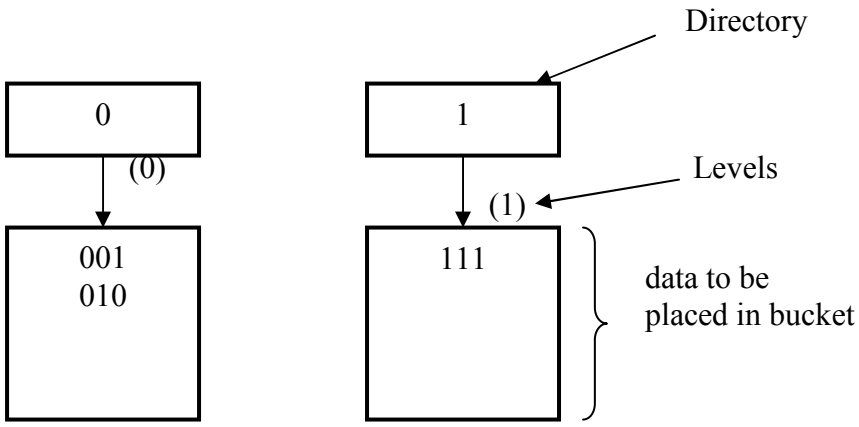
**Advantages:**

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.

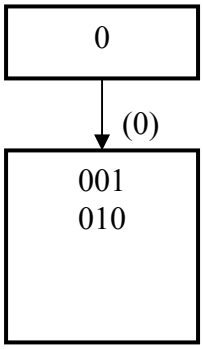
For eg:




- 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



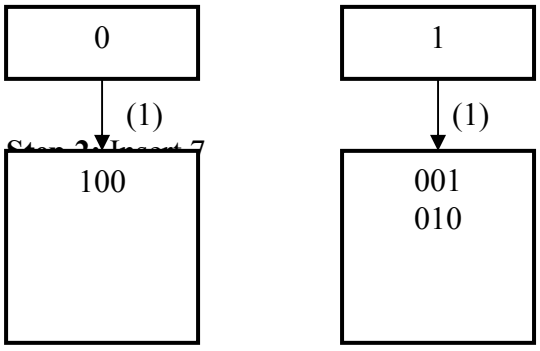

1 = 001

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

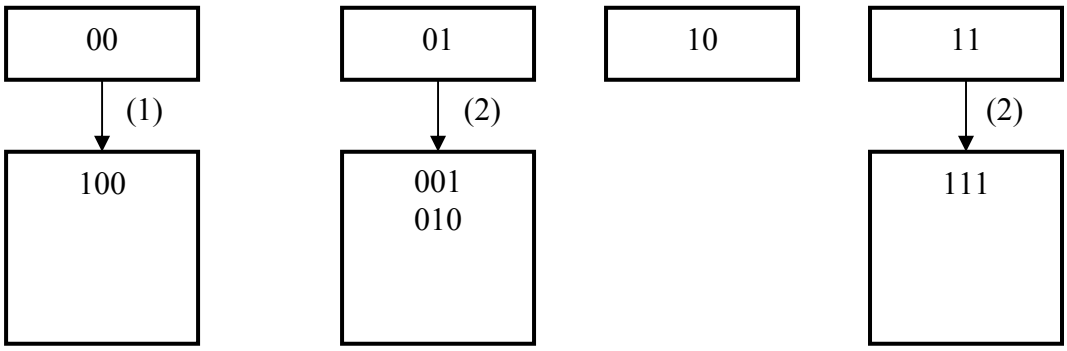



4 = 100
5 = 101

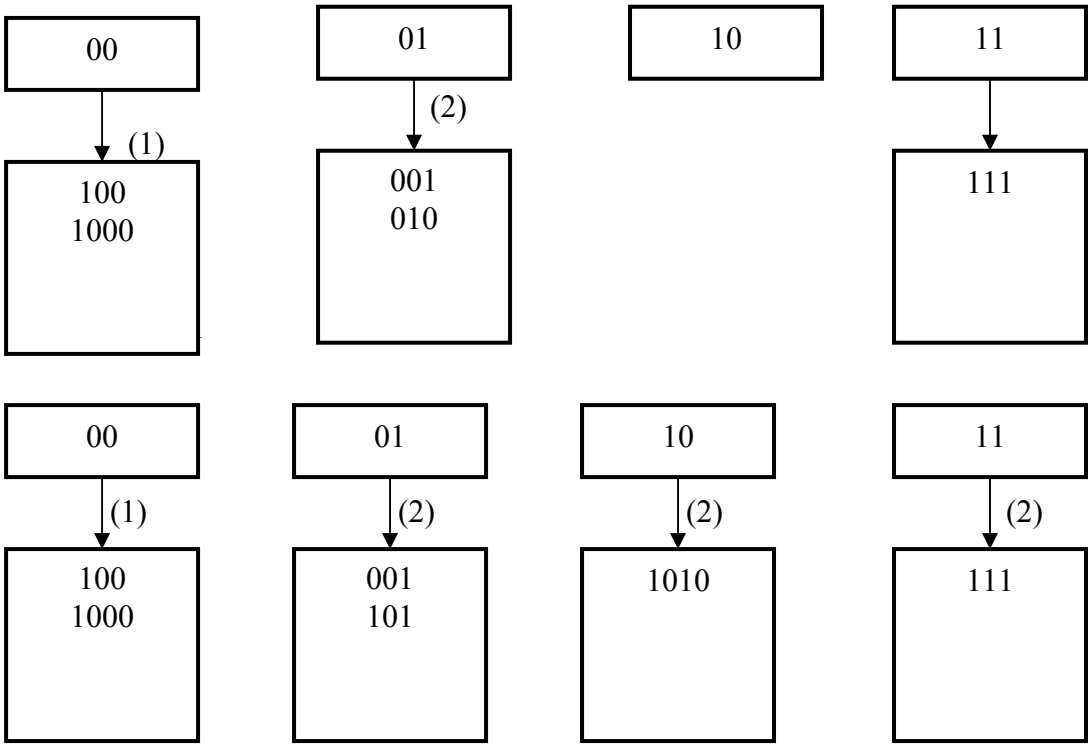
Based on last bit the data is inserted.

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.



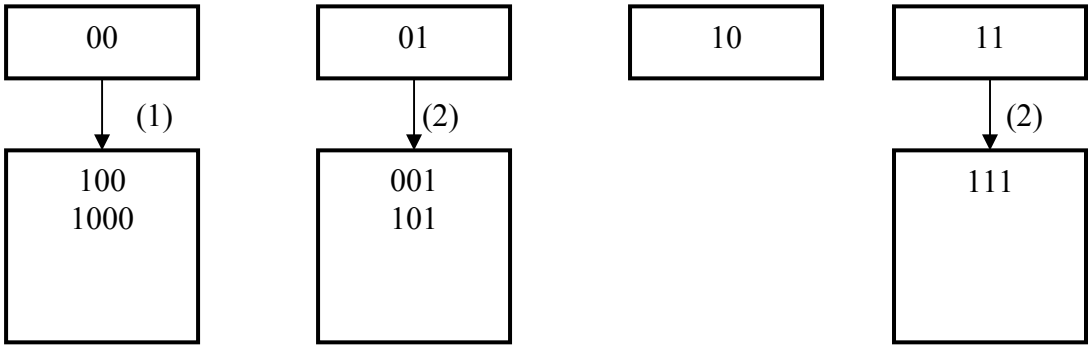

**Step 3:** Insert 8 i.e. 1000



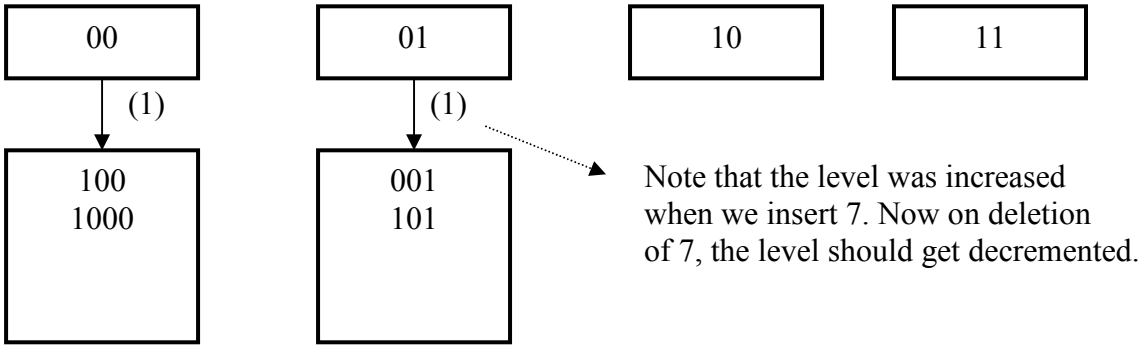

Thus the data is inserted using extensible hashing.

**Deletion Operation:**

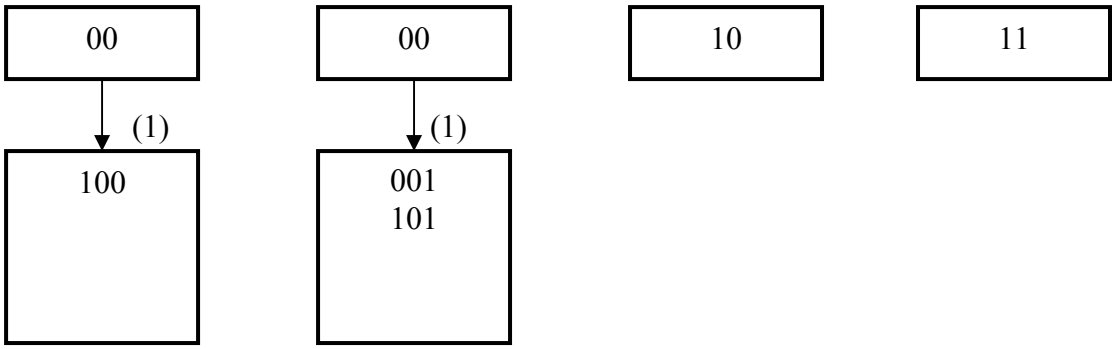
If we wan tot delete 10 then, simply make the bucket of 10 empty.




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. Fro browser program while caching the web pages, hashing is used.

# SORTING

The process of getting data elements into order is called sorting. The order may be ascending or descending as required. There are many sorting algorithms available. But no algorithm is best in all cases. Ideally, different algorithms are used depending on whether one is sorting a small set or a large set, on whether the individual elements of the set occupy a lot of storage, on how easy it is to compare two elements to figure out which one should precede the other, and so on. That means different applications require different sorting methods.

Sorting algorithms can be computationally expensive, particularly as data sets become large. They can be classified into simple algorithms and sophisticated algorithms. To sort n data elements, simple techniques require the order of n square comparisons where as sophisticated techniques require the $O(n \log_2 n)$ comparisons. In general, sorting methods involves the exchange of elements and moving a portion of data. For a large data elements set, these operations require considerable amount of processing time. Thus one of the main objective of designing sorting algorithms is to minimize exchanges or movement of data.

# BUBBLE SORT

The bubble sort is the simplest but not very efficient of sorts. It compares adjacent elements in a list of data elements, and swaps them if they are not in order. Each pair of adjacent elements is compared and swapped until the smallest (or largest) element "bubbles up" to the top, thus the name **bubble sort.** Repeat this process till all the elements are in sorting order. This technique is often called exchange sort, adjacent sort or sinking sort.

The step-by-step procedure to sort in ascending (or descending) order is given below:

- Compare two numbers at a time. Starting with the first two numbers.
- If the top number is larger (or smaller) than second number, then exchange the two numbers.
- Go down one number and compare that number to the number that follows it. These two form a new pair.
- Continue this process until no exchange has been made in an entire pass through the list.

Example:

2 6 5 4 1 7 3

**Pass 1**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Compare 2 with 6, no exchange | 2 | 6 | 5 | 4 | 1 | 7 | 3 |
| Compare 6 with 5, exchange | 2 | 5 | 6 | 4 | 1 | 7 | 3 |
| Compare 6 with 4, exchange | 2 | 5 | 4 | 6 | 1 | 7 | 3 |
| Compare 6 with 1, exchange | 2 | 5 | 4 | 1 | 6 | 7 | 3 |
| Compare 6 with 7, no exchange | 2 | 5 | 4 | 1 | 6 | 7 | 3 |
| Compare 7 with 3, exchange | 2 | 5 | 4 | 1 | 6 | 3 | 7 |

During pass 1, 6 compares, 4 exchanges.

**Pass 2**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Compare 2 with 5, no exchange | 2 | 5 | 4 | 1 | 6 | 3 | 7 |
| Compare 5 with 4, exchange | 2 | 4 | 5 | 1 | 6 | 3 | 7 |
| Compare 5 with 1, exchange | 2 | 4 | 1 | 5 | 6 | 3 | 7 |
| Compare 5 with 6, no exchange | 2 | 4 | 1 | 5 | 6 | 3 | 7 |
| Compare 6 with 3, exchange | 2 | 4 | 1 | 5 | 3 | 6 | 7 |

During pass 2, 5 compares, 3 exchanges.

**Pass 3**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Compare 2 with 4, no exchange | 2 | 4 | 1 | 5 | 3 | 6 | 7 |
| Compare 4 with 1, exchange | 2 | 1 | 4 | 5 | 3 | 6 | 7 |
| Compare 4 with 5, no exchange | 2 | 1 | 4 | 5 | 3 | 6 | 7 |
| Compare 5 with 3, exchange | 2 | 1 | 4 | 3 | 5 | 6 | 7 |

During pass 3, 4 compares, 2 exchanges.

**Pass 4**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Compare 2 with 1, exchange | 1 | 2 | 4 | 3 | 5 | 6 | 7 |
| Compare 2 with 4, no exchange | 1 | 2 | 4 | 3 | 5 | 6 | 7 |
| Compare 4 with 3, exchange | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

During pass 4, 3 compares, 2 exchanges.

**Pass 5**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Compare 1 with 2, no exchange | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Compare 2 with 3, no exchange | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

During pass 5, 2 compares, no exchange.

**Pass 6**

Compare 1 with 2, no exchange 1 2 3 4 5 6 7

During pass 6, 1 compare, no exchange.

Note that after the first pass the largest element (7 in this example) is in the last position of the array. In the $i^{th}$ pass, $i^{th}$ largest element bubbles up to the $i^{th}$ position from the top of the array. Since each pass places a new element into its proper position, it requires N-1 passes for N elements.

In each pass, at least one element will be placed in order. For example, if we are sorting in highest to lowest element, the smallest in the list we are searching will be captured by the swapping procedure. i.e., it will successfully swapped until it rises to the top of the array. If the data set contains N elements, for the next pass, we don't have to check the top element, and can stop at element N-1. For the pass that follows after that, stop at N-2, N-3 and so on..
In the above example all the elements are in sorted position in pass 4 and hence there is no exchange in the last two passes. These unnecessary passes can be avoided by introducing a simple check: when there are no exchanges in a particular pass, the sorting is completed and the process can be terminated.

The algorithm requires N-1 passes: each pass places one item in its correct place. Obviously, the $N^{th}$ element would also be in the correct place. The $i^{th}$ pass makes N-1 comparisons.

So, the time

$$T(N) = (N-1) + (N-2) + \ldots\ldots + 2 + 1$$

$$= \sum_{i=1}^{N-1} i = N/2\ (N-1) = O(N^2)$$

The bubble sort is simple but least efficient. If the data set contains N elements, the maximum number of passes is N-1. Similarly, the number of tests per pass ranges between N-1(for the first pass) and 1(for the last pass)- with an average value of N/2. Thus the upper bound on the total number of tests is roughly (N-1)*N/2. That is roughly proportional to $N^2$. Bubble sorting should be avoided whenever the number of elements to be sorted can be large.

```
public void bubbleSort()
    {
    int out, in, noexchange;

    for(out=nElems-1; out>1; out--)   // outer loop (backward)
    {  noexchange = 1;
      for(in=0; in<out; in++)        // inner loop (forward)
      {
        if( a[in] > a[in+1] )       // out of order?
```

```
      {
        swap(in, in+1);          // swap them
        noexchange=0;
      }
    }
    if(noexchange==1)
    return;
  }
  }  // end bubbleSort()
```

## INSERTION SORT

Insertion sort resembles the arrangement of cards in card-player game. When a new card is picked, it is inserted into its proper place.

By engaging in the activity of sorting a hand of cards as each successive card is dealt you might have observed the following sequence of events:

The first card is dealt and, since it is the only card, it is already in your hand in the correct place. The second card is dealt and then decide to place it either before or after the first card. With the third card the decision that must made is whether to insert it either before the first or second cards or after the second card. This basic behavior goes on as each new card is received.

In more general terms, assume that already have I cards sorted in one hand. When we receive the $(i+1)^{th}$ card "look through" the I sorted cards and "insert" new card in "its proper place". This is the basic idea that describes a well known algorithm known as **insertion sort.**

Example:

Arrange the following data elements into ascending order

5 2 1 3 6 4

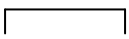
Consider the first element. For one element no sort is required. Hence, start with second element and compare it with first element and put it into correct place.

5 2 1 3 6 4

Since second element is less than first element, place it in first position.

2 5 1 3 6 4

Next take the third element and insert into its proper position in the first three positions.

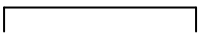


2 5 1 3 6 4

Since the third element is less than both first and second element, insert it in the first position and move the other two elements into next positions in the same order.

1 2 5 3 6 4

Similarly pick the fourth element and insert into its proper place in the positions from first to fourth.

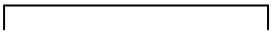


1 2 5 3 6 4

After insertion the data looks like

1 2 3 5 6 4

Similarly pick the fifth element and insert into correct place between the positions first and fifth.

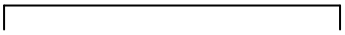


1 2 3 5 6 4

Here the fifth element is already in the sorted position.

1 2 3 5 6 4

The last element in this example is picked and place into its proper place.



1 2 3 5 6 4

Since the correct position for the last element (4) is fourth position, insert it into that position and move the greater elements down to the list.

1 2 3 4 5 6

Insertion sort require n-1 passes. The maximum number of iterations in **worst case** is $1 + 2 + 3 + \ldots\ldots\ldots\ldots + (n-1) = n(n-1)/2$. Thus the complexity of insertion sort is $O(n^2)$.
The **best case** analysis of insertion sort should show some improvement in performance over the worst case. It turns out that the best case input consists of list that is already sorted. Thus the best case for insertion sort – for comparisons and swaps- is a very favorable n-1 iterations i.e, with the complexity O(n).

```
public void insertionSort()
  {
  int in, out;

  for(out=1; out<nElems; out++)     // out is dividing line
    {
    int temp = a[out];          // remove marked item
    in = out;                   // start shifts at out
```

```
        while(in>0 && a[in-1] >= temp) // until one is smaller,
          {
          a[in] = a[in-1];           // shift item to right
          --in;                      // go left one position
          }
        a[in] = temp;                // insert marked item
        }  // end for
      }  // end insertionSort()

    }  // end class ArrayIns
```

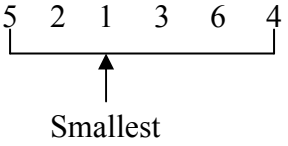
## SELECTION SORT

The Selection Sort searches all of the elements in a list until it finds the smallest element. It "swaps" this with the first element in the list. Next it finds the smallest of the remaining elements, and "swaps" it with the second element. Repeats this process until only the last two elements in the list are compared.

Example: Consider the data elements

5 2 1 3 6 4

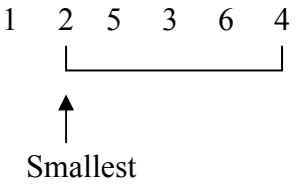
This method starts by searching for the smallest element in the list.




Swap the smallest element with the first element.

1 2 5 3 6 4

After the smallest element in the first position, continue the process by taking second element and looking for the next smallest element, in the remaining unsorted array.

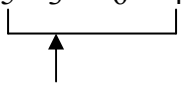



In the special case, the next smallest element is in the second position already. Swapping the element with itself keeps it in the same position.

1 2 5 3 6 4

After the next smallest element is in the second position, continue as above for the next smallest element.

1 2 5 3 6 4



Smallest

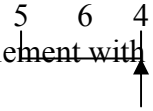
Swap this smallest element with the third element

1 2 3 5 6 4

Next search for the next smallest element from the fourth element.

1 2 3 5 6 4

Swap this smallest element with the fourth element.



1 2 3 4 6 5

Continue in the same way for the next smallest element.

1 2 3 4 5 6

The number of comparisons is N-1 in the first pass, N-2 in the second pass, N-3 in the third pass and so on.. and N-I in the $i^{th}$ pass. Hence, there are at most (N-1)+(N-2)+….+1 = N(N-1)/2 comparisons of keys, time complexity is $O(N^2)$.

Selection Sort is efficient than bubble sort and insertion sort in the sense that there are no more than N-1 actual exchanges. Thus the selection is very much suitable in the applications in which excessive swapping is a problem.

Since the number of comparisons can not be limited in a particular pass, there is no significance if the input data is completely sorted or unsorted. That is, performance for both worst case and best case is same in Selection Sort.

Thus, Bubble, Insertion, Selection Sorts algorithms with $O(n^2)$ complexity are suitable for small number of items only.

```
public void selectionSort()
  {
  int out, in, min;

  for(out=0; out<nElems-1; out++)   // outer loop
    {
    min = out;                    // minimum
    for(in=out+1; in<nElems; in++) // inner loop
```

```
      if(a[in] < a[min] )        // if min greater,
         min = in;              // we have a new min
    swap(out, min);              // swap them
    }  // end for(outer)
  }  // end selectionSort()
```

# QUICK SORT

As the name suggests, Quick sort is a fast sorting algorithm invented by C.A.R.Hoare. It has $O(N\log_2 N)$ for best and average case performance; and $O(N^2)$ for worst-case performance.

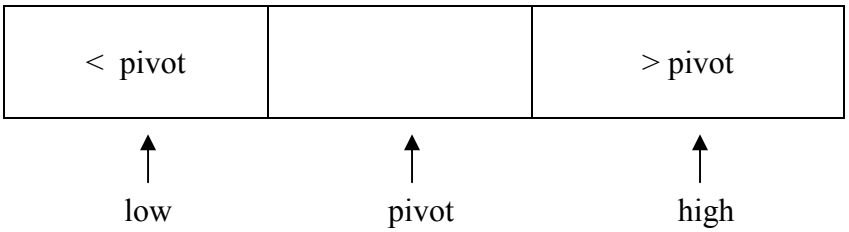
Quicksort approaches sorting in a radically different way from the other sorts. Rather than successively sorting the same data set, it attempts to partition the data set into two sections, then sort each section separately- adopting divide and conquer strategy.

In quick sort, we divide the array of items to be sorted into two partitions and then call the quick sort procedure recursively to sort the two partitions. To partition the data elements, a **pivot** element is to be selected that all the items in the lower part are less than the pivot and all those in the upper part greater than it.

Consider an array A of N elements to be sorted. Select a pivot element among the N elements. The selection of pivot element is some what arbitrary, however, the first element is a convenient one.

The array is divided into two parts so that the pivot element is placed into its proper position satisfying the following properties:

1. All elements to the left of pivot are less than the pivot element.
2. All elements to the right of pivot are greater than or equal to the pivot element.

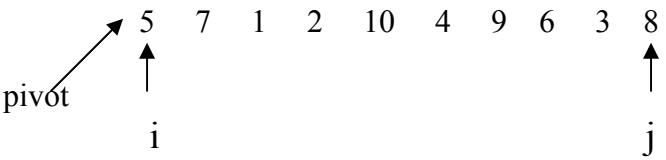



At this stage, the array is divided into two sub arrays. The same process is then repeated recursively on the two parts of the array on either side of the pivot in the same way. Subsequent sub arrays that might result during the process are also sorted like wise until all the elements in the array are in sorted position.

Illustrate the QuickSort method by applying it to the following array of elements.

5 7 1 2 10 4 9 6 3 8

Take the first element as the pivot element, i.e. pivot=A[1]=5. Now, consider two indices I and j to represent the lower and upper bounds of the elements to be sorted.



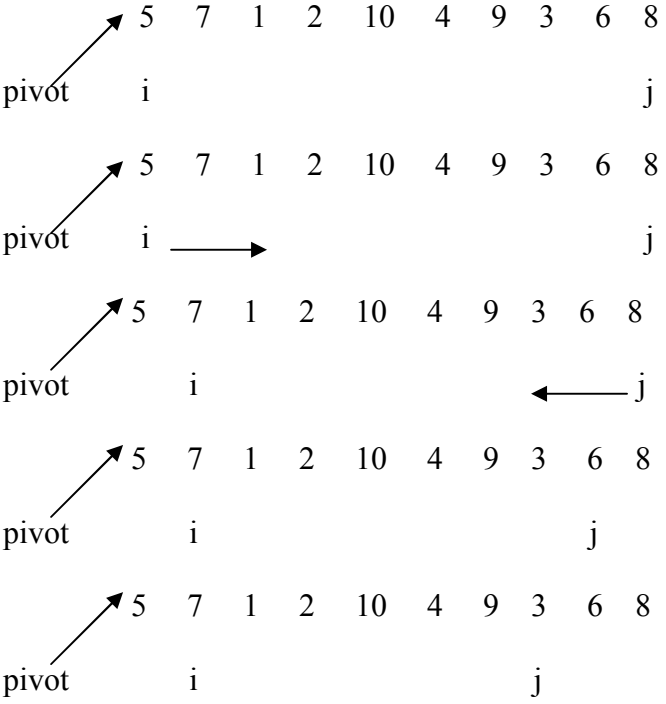

In the partitioning phase, move the elements less than or equal to pivot toward the left and those greater than or equal to toward the right.

The following are the steps for partitioning the array and keeping the pivot at correct place:

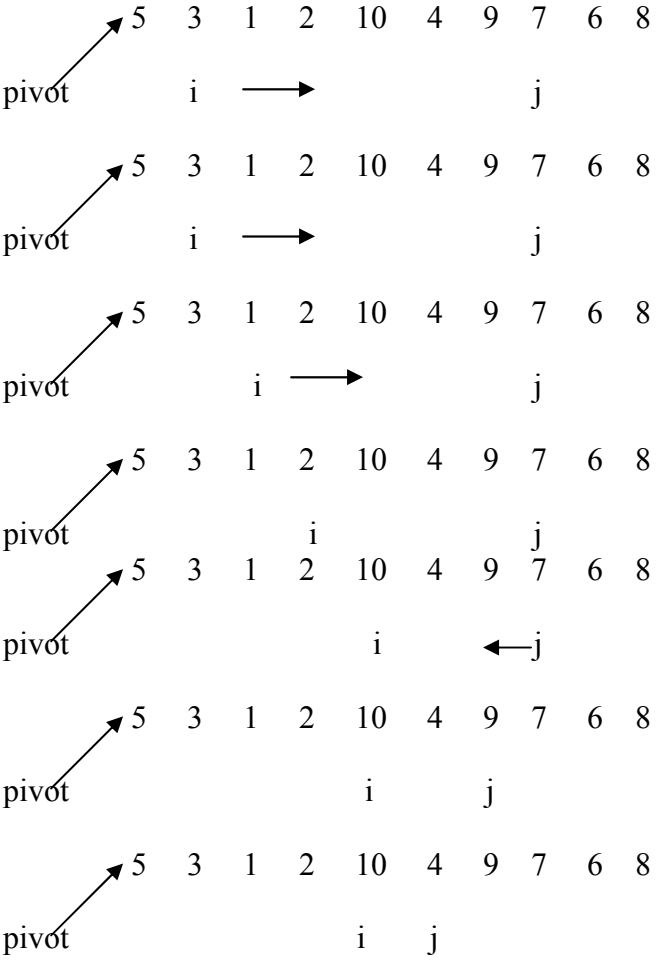
1. Select first element of array A as pivot.
2. Initialize i and j to first and last elements of the array respectively.
3. Increment i, until A[i]>pivot; stop
4. Decrement j, until A[j]<pivot; stop
5. If i<j, Exchange A[i] and A[j]
6. Repeat steps 3, 4, and 5 until i>j, i.e. when I and j cross each other.
7. Exchange the pivot element with the element pointed to by j which is correct place for pivot.

Note that in step 5, the swapping keeps the larger element to the right and smaller element to the left, relative to the pivot.

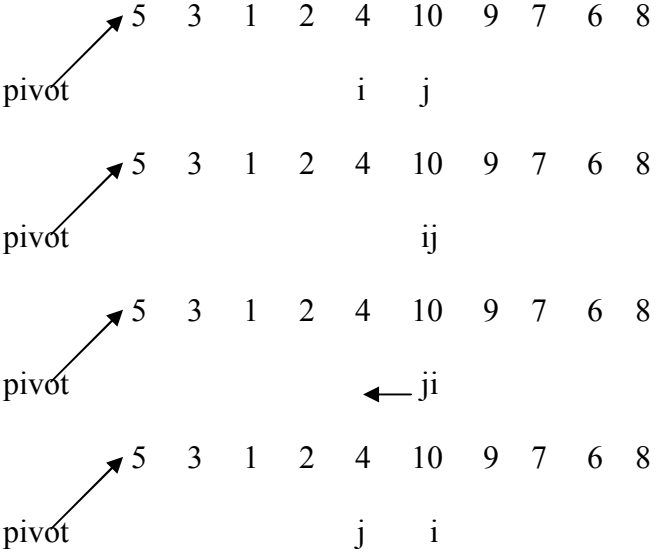
The sequence of steps indicating the movement of I and j for keeping the pivot is shown below.



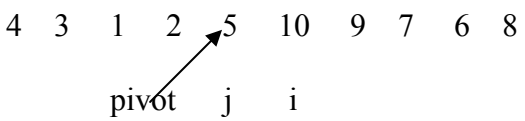
Since i<=j swap A[i] and A[j] and repeat the process.




Since i<=j, swap A[i] and A[j] and repeat the process.



Since i>j, now swap the pivot with A[j].




At this stage the pivot 5 is in its proper place, i.e. elements left to it are less than or equal to 5 and elements to its right are greater than or equal to 5. Now the problem is break down into two smaller problems: sorting the array between start and j-1 and sorting the array between j+1 and end. The quick sort process can be applied to each of these sub arrays until all the elements are in sorted order.

[5 7 1 2 10 4 9 3 6 8]

[4 3 1 2] 5 [10 9 7 6 8]

[2 3 1] 4 5 [10 9 7 6 8]

[1] 2 [3] 4 5 [10 9 7 6 8]

1 2 [3] 4 5 [10 9 7 6 8]

1 2 3 4 5 [10 9 7 6 8]

1 2 3 4 5 [8 9 7 6] 10

1 2 3 4 5 [7 6] 8 [9] 10

1 2 3 4 5 6 [7] 8 [9] 10

1 2 3 4 5 6 7 8 [9] 10

1 2 3 4 5 6 7 8 9 10

The time analysis for QuickSort is explained below:

The requirement to sort N elements using quicksort procedure involves 3 components (a) time taking for partitioning the array, which is roughly proportional to N, (b) time required for sorting lower subarray, and (c) time required for sorting upper subarray. Assume that there are K elements in the lower subarray.

Therefore,

$T(N) = T(K) + T(N-K-1) + cN$ where c is proportional constant.

In the worst case i.e when the array is already almost sorted, the array is always partitioned into two subarrays in which one of them is always empty. Thus for the worst case analysis,

$$T(N) = T(N-1) + cN \quad \text{for } N > 1$$

We can get the equations for next recursive calls by replacing N with N-1, N-2, ……. 2.

$$T(N-1) = T(N-2) + c(N-1)$$
$$T(N-2) = T(N-3) + c(N-2)$$
……..
……..
$$T(2) = T(1) + c(2)$$
$$T(1) = T(0) + c(1) = c(1)$$

Substituting these values in the first equation, we have

$$T(N) = c\sum_{i=1}^{N} = c.\ (N+1)N/2 = O(N^2)$$

The best case can be viewed when the partition takes place exactly at half of the original array. Thus,

$$T(N) = 2\ T(N/2) + cN$$

$$T(N)/2 = T(N/2)/(N/2) + c$$

Proceeding similarly as in worst case for recursive calls, we have

$$T(2)/2 = T(1)/1 + c$$

Assume the total number of elements N is a power of 2, such that $N = 2^M$ i.e. $M=\log_2 N$, there are M steps to add since the array is divided into exactly half. Therefore,

$$T(N)/N = T(1)/1 + c \log_2 N$$

$$T(N) = N + c\ N \log_2 N = O(N \log_2 N)$$

The analysis for average case performance is some what complicated, but it can be also analyzed as $O(N \log_2 N)$.

QuickSort is a good general purpose sorting approach that can be used for sorting most arrays. Its efficient is also relatively good. Because it uses element exchanges, it also requires relatively small amounts of memory.

**Algorithm:** Partition the set A(m:p-1) about A(m)

procedure PARTITION(m, p)

```
//Within A(m), (m+1),…,A(p-1) the elements are
//rearranged in such a way that if initially t=A(m),
//then after completion A(q) = t, for some queue between m amd p-1
//A(k)≤t for m≤k<q and A(k)≥t for q<k<p
//The final value of p is q

integer m, p, i; global A(m:p)
v←A(m);
i←m;
loop
        loop i←i+1 until A(i)≥v repeat
        loop p←p-1 until A(p)≤v repeat
if i<p
then call INTERCHANGE(A(i), A(p))
else
exit
endif
repeat
A(m) ←A(p); A(p) ←v
end PARTITION
```

**Algorithm:** Sorting by partitioning

```
procedure QUICKSORT(p, q)
//sorts the elements A(p),……,A(q) which resides
//in the global array A(1:n) into ascending order
//A(n+1) is considered to be defined
//and must be≥ all elements in A(p:q); A(n+1) = +∞
integer p,q; global n, A(1:n)
if p<q
then j ← q+1
call PARTITION(p,j);
call QUICKSORT(p, j-1);
call QUICKSORT(j+1, q);
endif
end QUICKSORT
```

**//QSort.java**

```
    void qsort(int list[],int lb,int ub)
    {
    int i,j,key,b;
    boolean flag=true;
    if(lb<ub)
    {
    i=lb;
```

```
j=ub+1;
key=list[lb];

while(flag)
{
i++;
while ((list[i]<key) && (i<=ub))
i++;
j--;
while((list[j]>key) && (j>=lb))
j--;
if(i<j)
{
b=list[i];
list[i]=list[j];
list[j]=b;
}
else
flag=false;
}
b=list[lb];
list[lb]=list[j];
list[j]=b;
}
else
return;
qsort(list,lb,j-1);
qsort(list,j+1,ub);
return;
}
}
```

## MERGE SORT

The merge sort algorithm closely follows the divide – and – conquer paradigm. The divide and conquer paradigm breaks the problem into several sub problems that are similar to the original problem but are smaller in size, solve the sub problems recursively and then combine these solutions to create a solution to the original problem.

The divide and conquer paradigm involves three steps at each level of the recursion.

(i) Divide the problem into number of sub problems.
(ii) Conquer the sub problems by solving them recursively. If the sub problems sizes are small enough, however, just solves the sub problems in a straight forward manner.
(iii) Combine the solutions to the sub problems into the solution for the original problem.

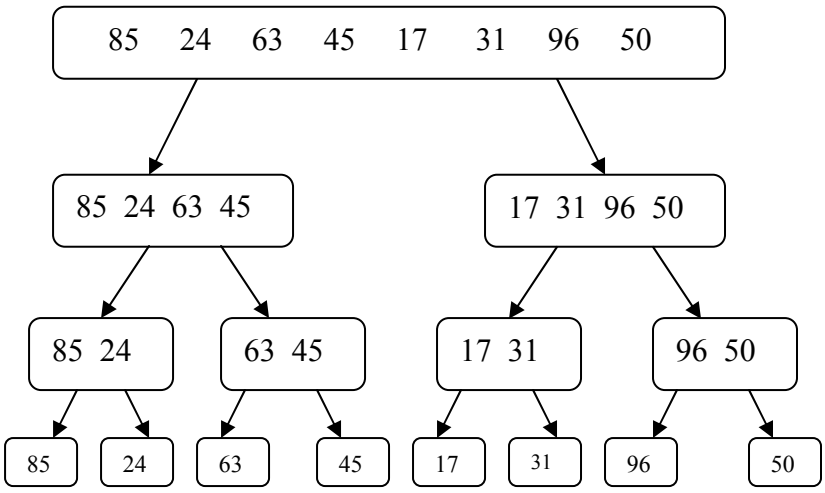
The merge sort algorithm closely follows the divide – and – conquer paradigm. Intuitively it operates as follows:

(i) **Divide:** Divide the n-element sequence to be sorted into two subsequences of n/2 elements each.
(ii) **Conquer:** Sort the two sub sequences recursively using merge sort.
(iii) **Combine:** Merge the two sorted sub sequences to produce the sorted order.

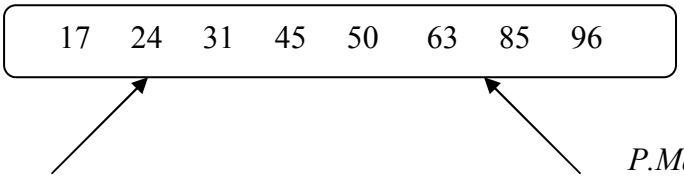
The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step. To perform the merging, we use auxiliary procedure MERGE(A,p,q,r). Where A is an array and p, q, r are indices numbering elements of the array such that p <= q < r. The procedure assumes that the sub array A[p..q] and A[q+1 .. r] are inserted order. It merges them to form a single sorted sub array that replaces the current sub array A[p..r].

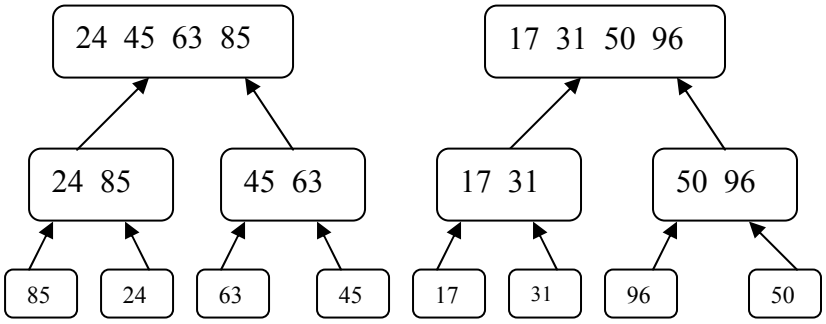
MERGE-SORT(A,p,q)
1: If p<r
2: then q ← [(p+r)/2)
3: MERGE-SORT (A,p,q)
4: MERGE-SORT (A,q+1,r)
5: MERGE(A,p,q,r)

**Example:** Merge sort tree T for an execution of the merge-sort algorithm on a sequence with 8 elemen|s :
(a) input sequences processed at each node of T.




(b) Output sequences generated at each node of T.






**//MERGE SORT**

```
class Merge
{
void msort(int list[],int p,int q)
{
int mid;

if(p<q)
{
mid=(p+q)/2;
msort(list,p,mid);
msort(list,mid+1,q);
merge(list,p,mid,q);
}
}
void merge(int list[],int l,int m,int u)
{
int b[]=new int[10];
int i,j,k;
i=l;
j=m+1;
k=l;
while((i<=m) && (j<=u))
{
if(list[i]<=list[j])
{
b[k]=list[i];
++i;
}
else
{
b[k]=list[j];
++j;
```

```
}
++k;
}
if(i>m)
{
while(j<=u)
{
b[k]=list[j];
++j;
++k;
}
}
else
{
while(i<=m)
{
b[k]=list[i];
++i;
++k;
}
}
for(int r=l;r<=u;r++)
list[r]=b[r];
}
}



class MSort
{
public static void main(String args[])
{
int size,list[];
Merge m=new Merge();


list=new int[10];

 System.out.println("Enter elements into the list:");
for(int i=0;i<10;i++)
{
list[i]=MyInput.readInt();
}

m.msort(list,0,9);
System.out.println("The list after sorting is..");
```

```
for(int i=0;i<list.length;i++)

System.out.println(list[i]);


}
}
```

## HEAP SORT

Heap sort is a method in which a binary tree is used. In this method first the heap is created using binary tree and then heap is sorted using priority queue.

Eg:

25 57 48 38 10 91 84 33

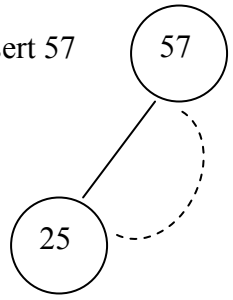
In the heap sort method we first take all these elements in the array "A"

| A[0] | A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] |
|---|---|---|---|---|---|---|---|
| 25 | 57 | 48 | 38 | 10 | 91 | 84 | 33 |

Now start building the heap structure. In forming the heap the key point is build heap in such a way that the highest value in the array will always be a root.

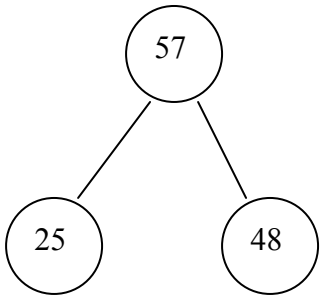
Insert 25


Insert 57



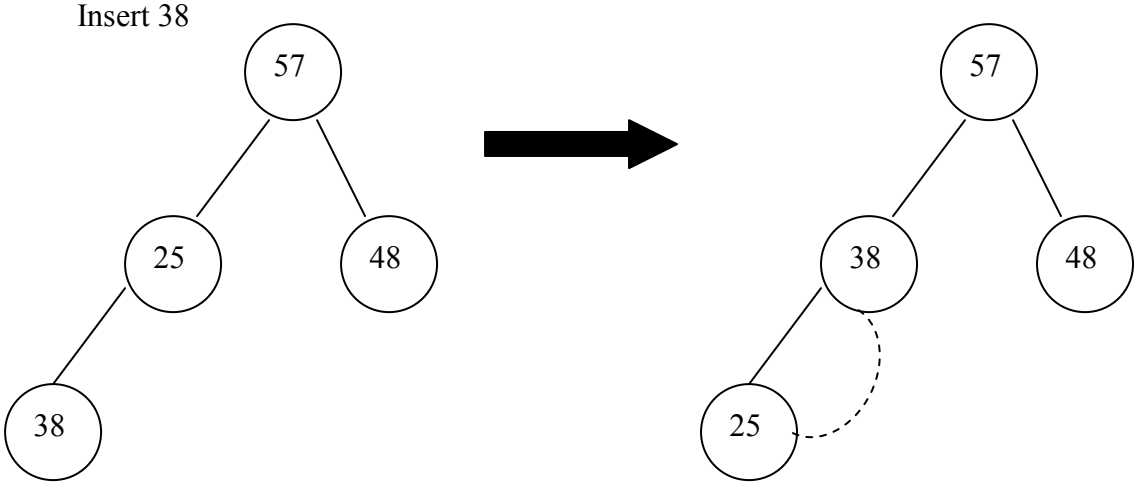

Since 25<57. Therefore 57 is root and 25 is left child

Insert 48



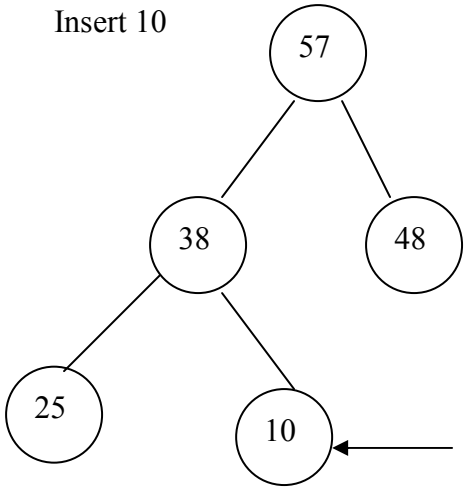

Now since 48 is less than 57. So it cannot be a root. So the root is 57. But 48>25 so it cannot be the left child of 25. So attach 48 as a right child of 57.

Insert 38



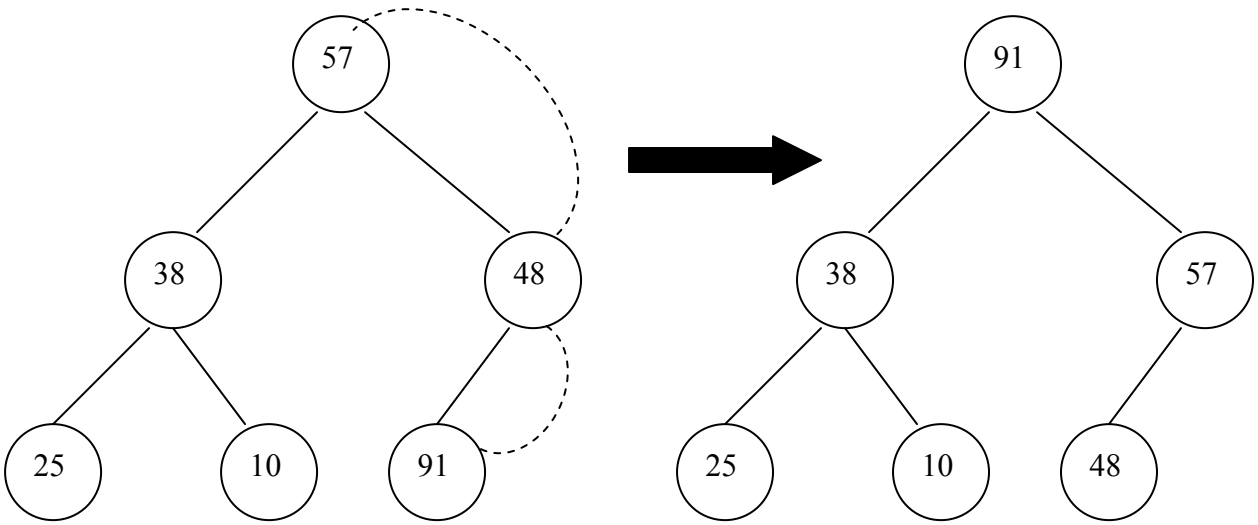

As 38 is higher than 25 so 38 becomes parent of 25

Insert 10



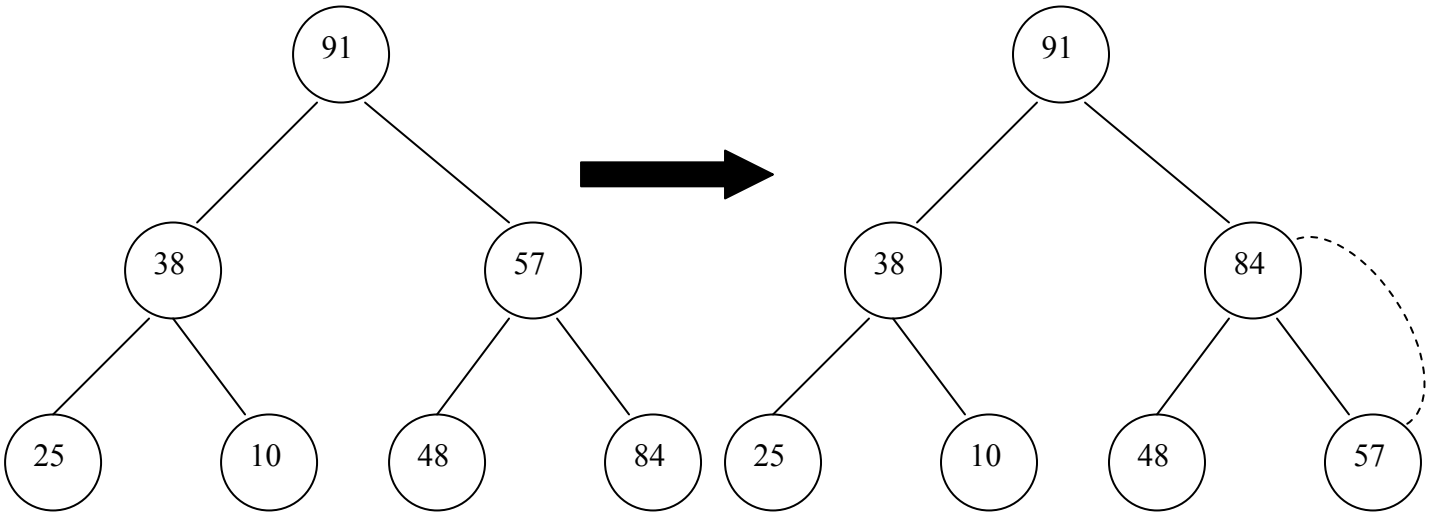

The element 10 is attached in the left sub trees of 57 i.e. as a right child of 38. The 10 can be attached as a left child of 25 or it can be attached as left child of 25 or it can be attached as left child of 48 even. But always we will assume to complete left sub tree having both left and right children so for the sake of completion the node 10 is attached to the right of 38

Insert 91



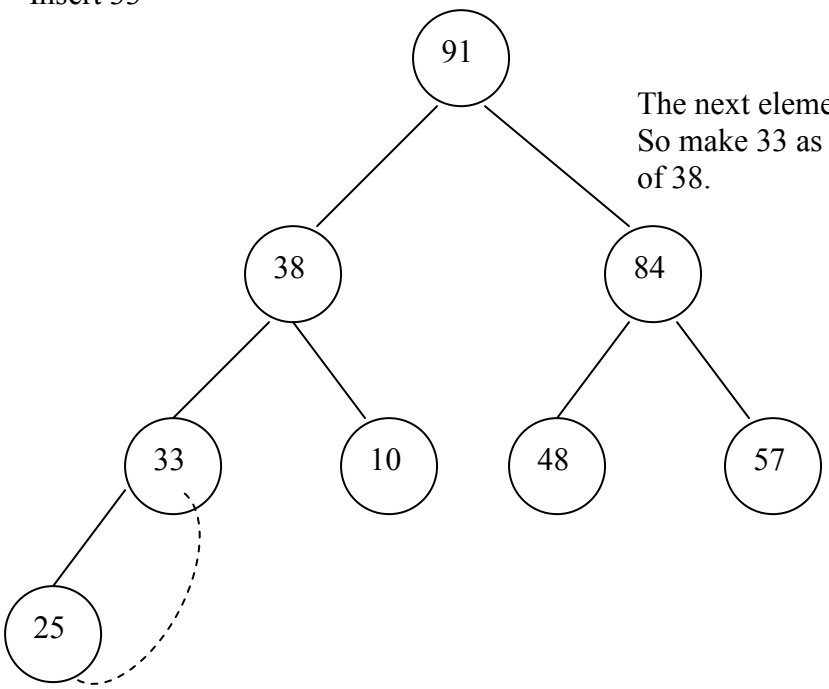

91 is the largest element compared to all other
elements , naturally it will be the root node.

Insert 84



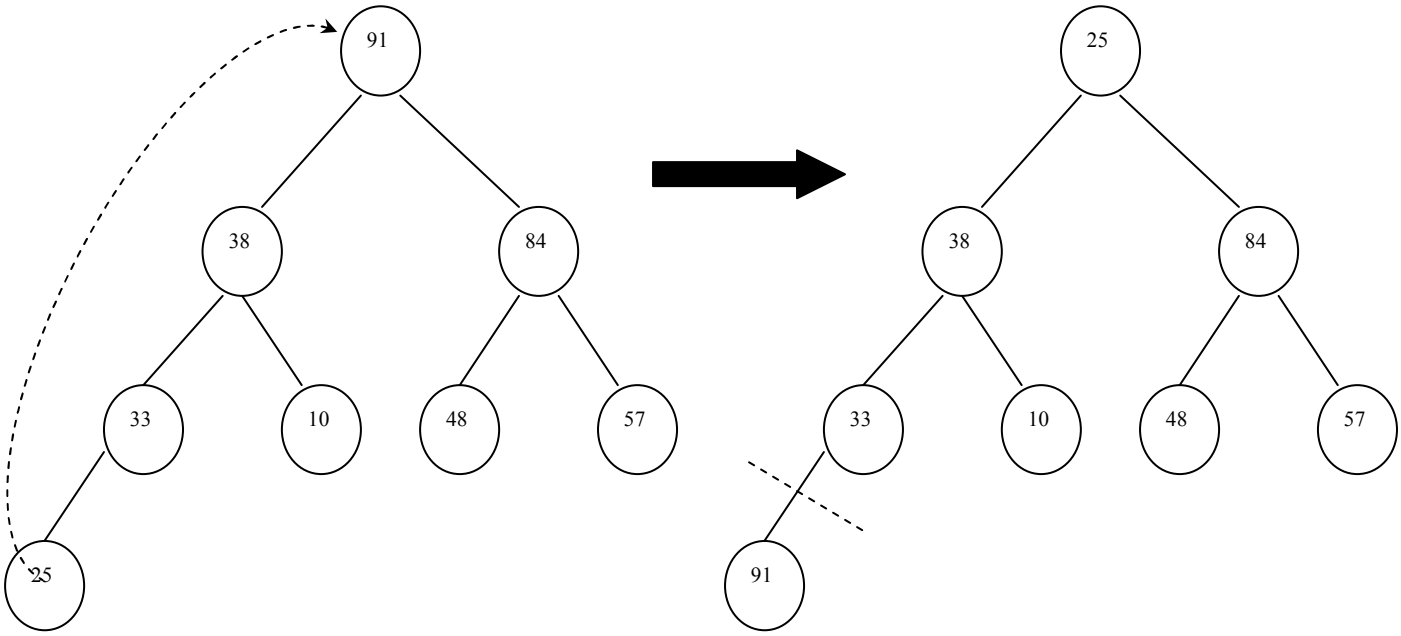

The next element is 84, which 91>84>57 the middle element. So 84 will be the parent of 57. For making the complete binary tree 57 will be attached as right of 84.

Insert 33

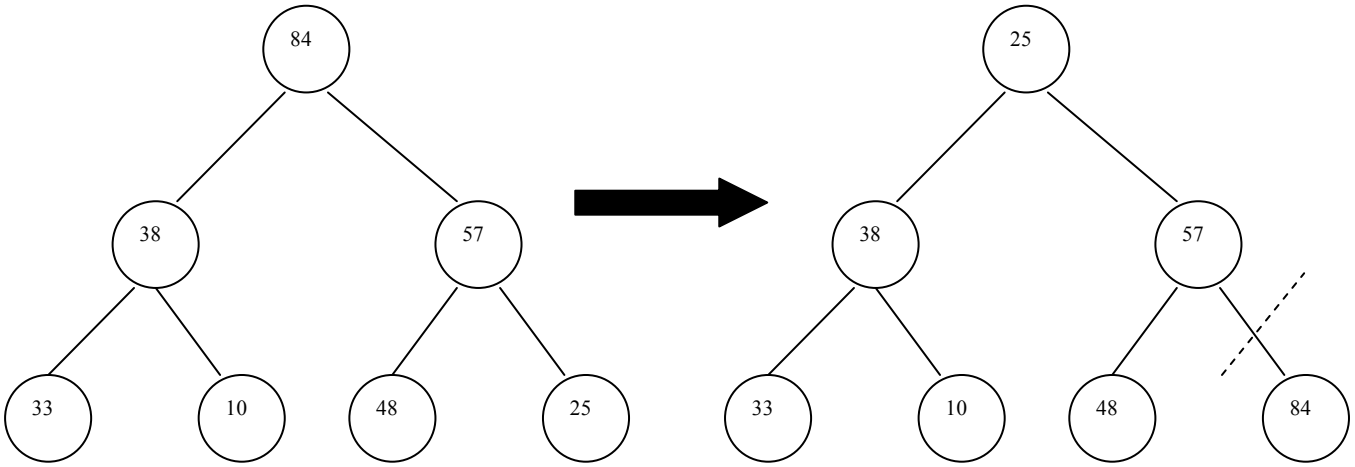



The next element is 33. The 38>33>25
So make 33 as a parent of 25 but child of 38.

Now the heap is formed. Let us sort it. For sorting the heap remember two main things the first thing is that the binary tree form of the heap should not be distributed at all. For the complete sorting binary tree should be remained.
And the second thing is that we will start sorting the higher elements at the end of array in sorted manner
i.e.. A[7]=91, A[6]=84 and so on..
Step 1:- Exchange A[0] with A[7]

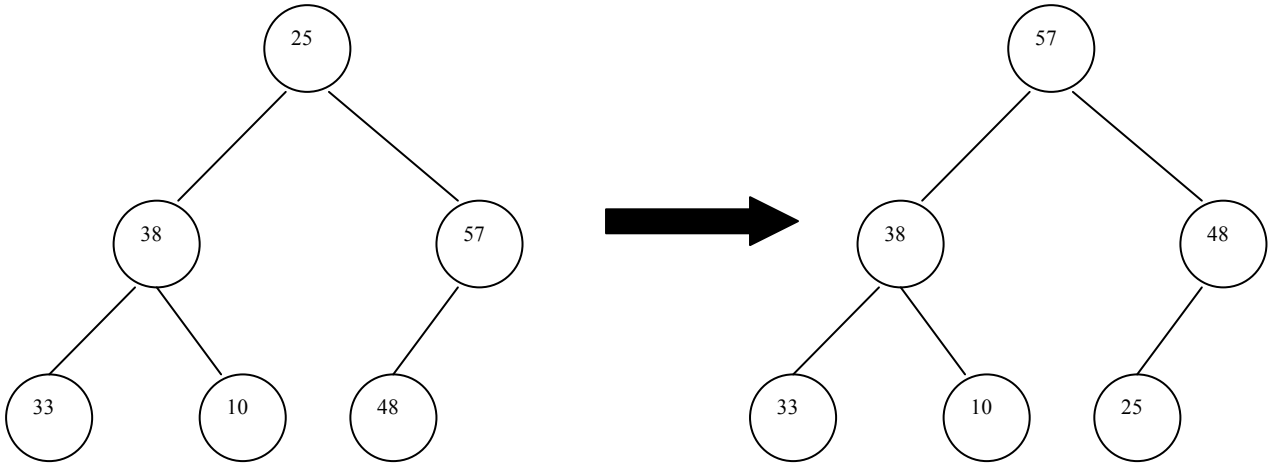


Step 2:-Exchane A[0] with A[6]

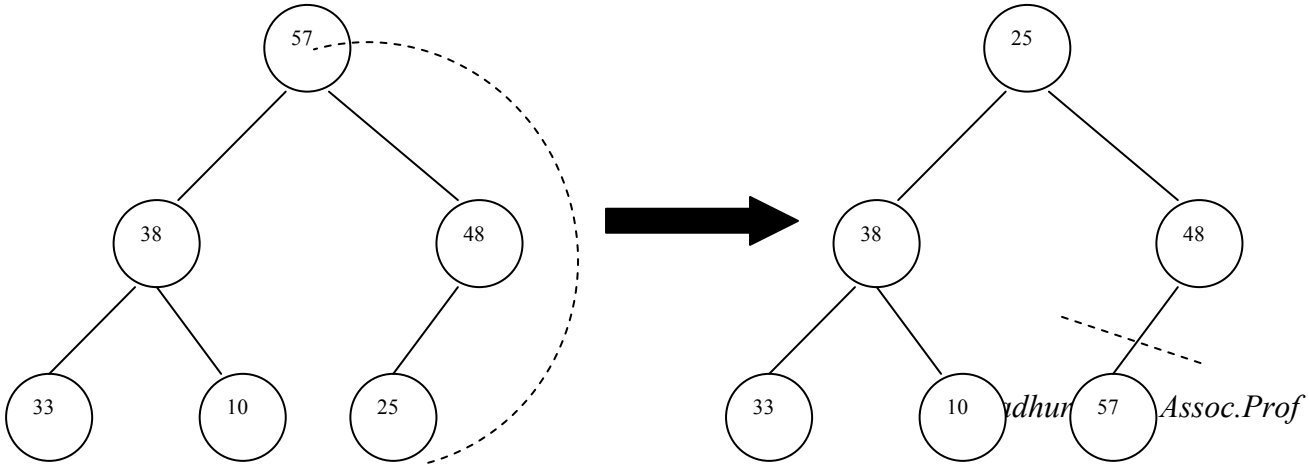



Queue

| | | | | | | 84 | 91 |
|---|---|---|---|---|---|---|---|

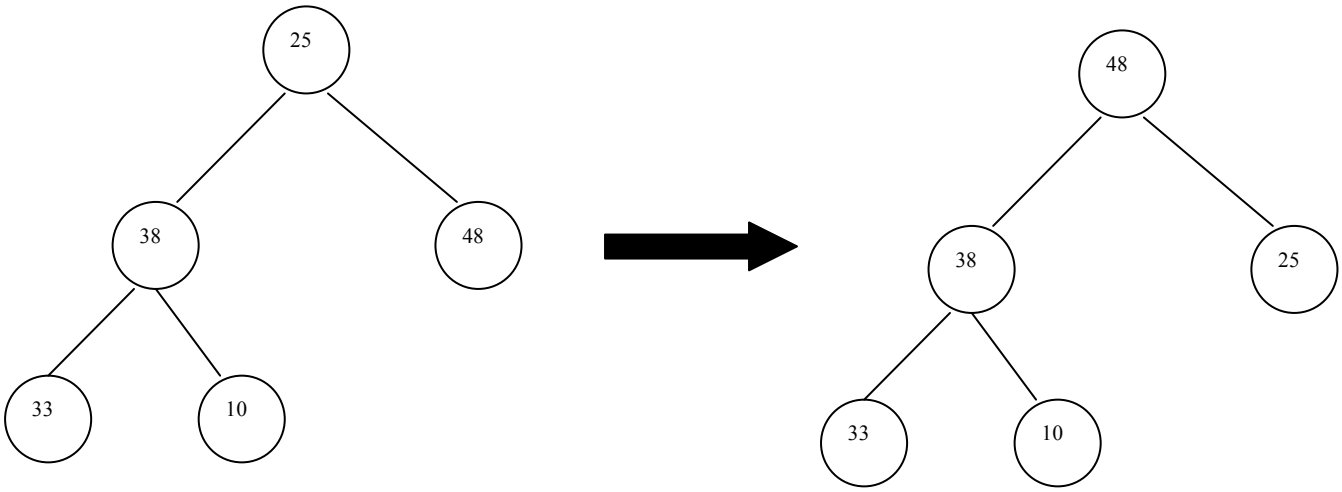



Step 3:-Exchane A[0] with A[5]

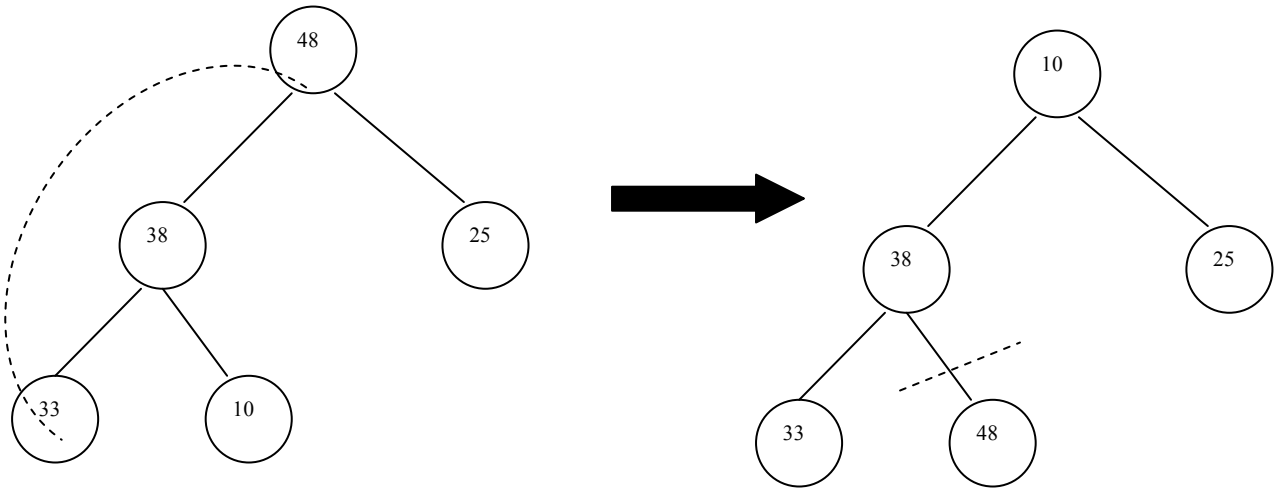


Queue

| | | | | | 57 | 84 | 91 |
|---|---|---|---|---|---|---|---|

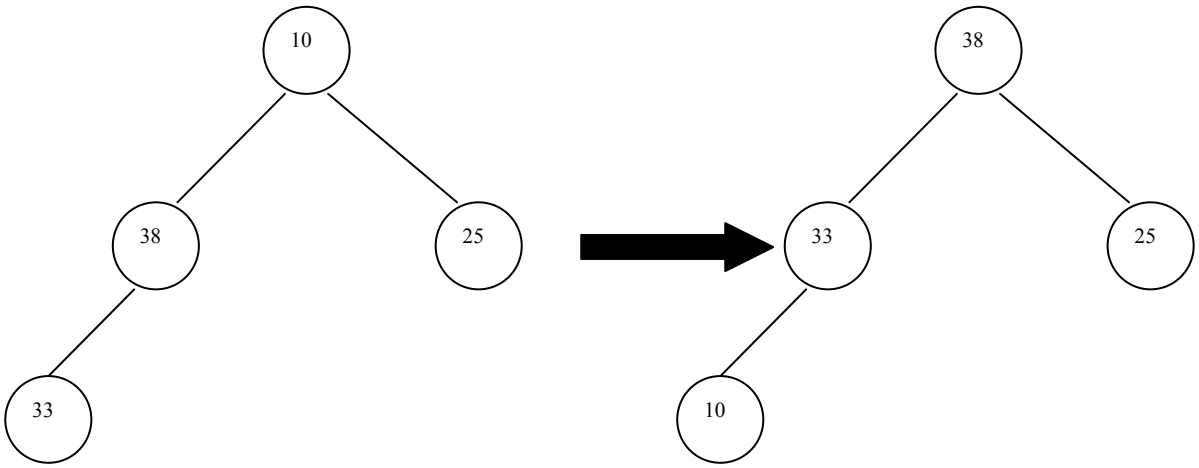



Step 4:-Exchane A[0] with A[4]

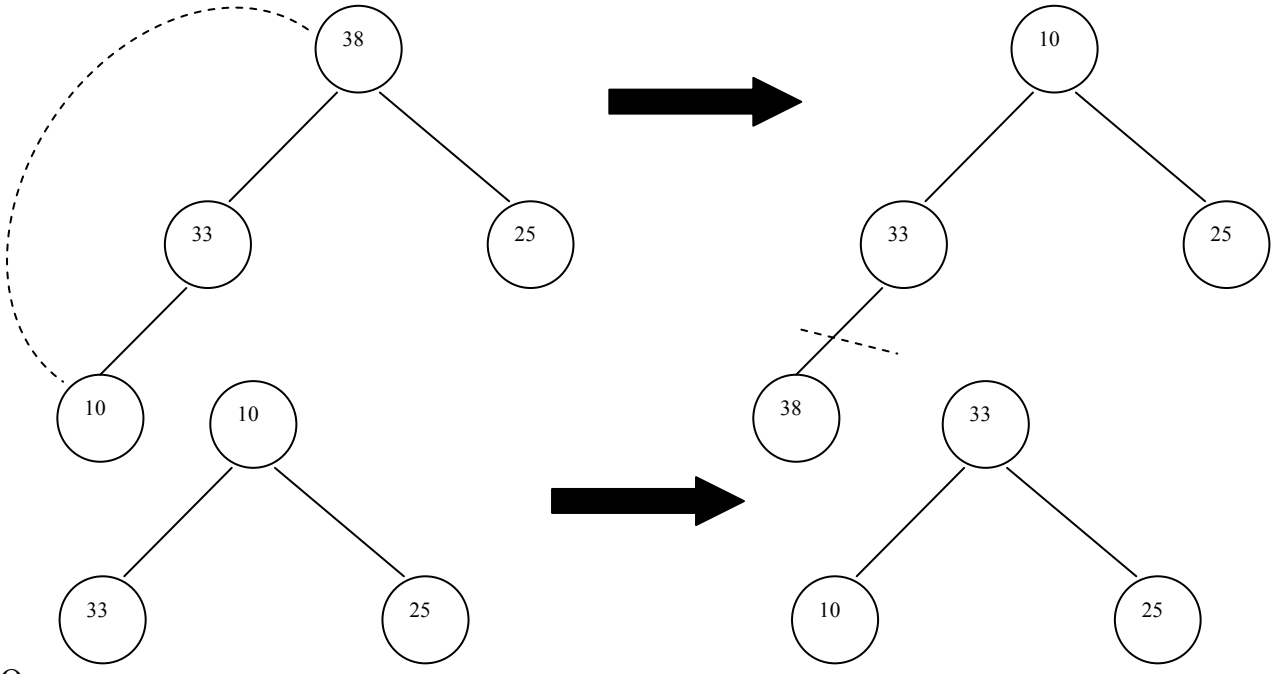



Queue

| | | | | 48 | 57 | 84 | 91 |
|---|---|---|---|---|---|---|---|



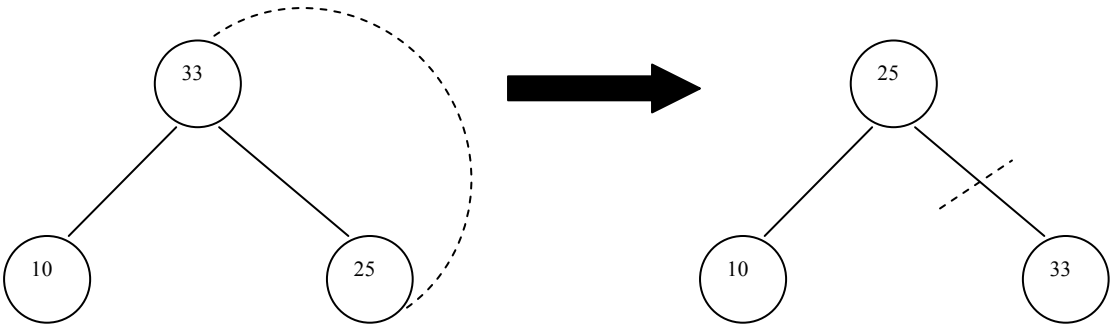

Step 5:-Exchane A[0] with A[3]




Queue

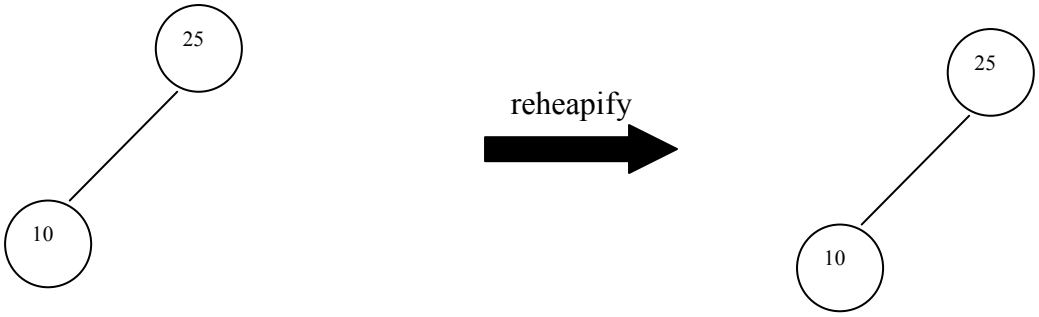
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | 38 | 48 | 57 | 84 | 91 |

Step 6:-Exchane A[0] with A[2]

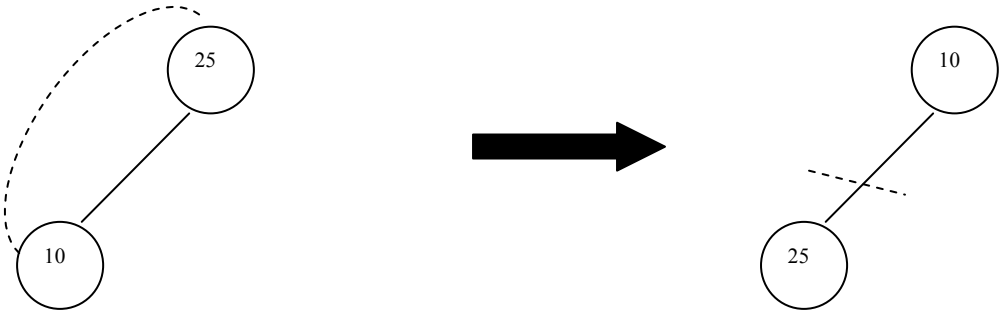



Queue

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 33 | 38 | 48 | 57 | 84 | 91 |




Step 7:-Exchane A[0] with A[1]




Queue

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 25 | 33 | 38 | 48 | 57 | 84 | 91 |


Step 8: The remaining element 10 has already occupied its proper position because only one position is empty so insert 10 also in the queue.

| 10 | 25 | 33 | 38 | 48 | 57 | 84 | 91 |
|---|---|---|---|---|---|---|---|

## RADIX SORT

Radix sort is a technique, which is usually used when large lists of names are to be sorted alphabetically. Using this sorting technique one can classify the list of names into 26 groups. The list is first sorted on the first letter of each name, i.e. the names are arranged in 26 classes, where the first class consists of those names that begin with alphabet A, the second class consists of those names that begin with alphabet B and so on.. During the second pass each class is alphabetized according to the second letter of the name. and so on.. For example if no name contains more than 20 characters then the sort requires 20 passes.

The radix sort is a technique, which is used by a card sorter. A card sorter contains 13 receiving pockets labeled as

9 8 7 6 5 4 3 2 1 10 11 12 R

Here R represents reject, if any card comes in this pocket.

For example, suppose a card sorter is given a collection of cards where each card contains a four-digit number punched in columns 1-4, The cards are first sorted according to the unit's digit. In the second pass, the cards are sorted according to the tens digit. In the third pass the cards are sorted according to the hundreds digit. In the fourth and last pass, the cards are sorted according to the thousands digit. For example, suppose the cards are..

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 4348 | 2143 | 4361 | 5423 | 4538 | 2128 | 5438 |

Sorting Rules:

i. Pass 1 => The unit digits are sorted into pockets. The cards are collected pocket by pocket from pocket 9 to 0. The cards are now re-input to the sorter.
ii. Pass 2 => The tens digits are sorted into pockets. Again cards are collected pocket by pocket and re-input to the sorter.
iii. Pass 3 => The $100^{th}$ digits are sorted into pockets. Again cards are collected pocket by pocket and re-input to the sorter.
iv. Pass 4 => The 1000's digits are sorted into pocket. Again cards are sorted pocket by pocket.

When cards are collected after fourth pass, the numbers are in the sorted order.

| Input | Pocket 0 | Pocket 1 | Pocket 2 | Pocket 3 | Pocket 4 | Pocket 5 | Pocket 6 | Pocket 7 | Pocket 8 | Pocket 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 4348 | | | | | | | | | 4348 | |
| 2143 | | | | 2143 | | | | | | |
| 4361 | | 4361 | | | | | | | | |
| 5423 | | | | 5423 | | | | | | |
| 4538 | | | | | | | | | 4538 | |
| 2128 | | | | | | | | | 2128 | |
| 5438 | | | | | | | | | 5438 | |

i. Pass 1

| Input | Pocket 0 | Pocket 1 | Pocket 2 | Pocket 3 | Pocket 4 | Pocket 5 | Pocket 6 | Pocket 7 | Pocket 8 | Pocket 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 4348 | | | | | 4348 | | | | | |
| 4538 | | | | 4538 | | | | | | |
| 2128 | | | 2128 | | | | | | | |
| 5438 | | | | 5438 | | | | | | |
| 2143 | | | | | 2143 | | | | | |
| 5423 | | | 5423 | | | | | | | |
| 4361 | | | | | | | 4361 | | | |

ii. Pass 2

| Input | Pocket 0 | Pocket 1 | Pocket 2 | Pocket 3 | Pocket 4 | Pocket 5 | Pocket 6 | Pocket 7 | Pocket 8 | Pocket 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 4361 | | | | 4361 | | | | | | |
| 4348 | | | | 4348 | | | | | | |
| 2143 | | 2143 | | | | | | | | |
| 4538 | | | | | | 4538 | | | | |
| 5438 | | | | | 5438 | | | | | |
| 2128 | | 2128 | | | | | | | | |
| 5423 | | | | | 5423 | | | | | |

iii. Pass 3

| Input | Pocket 0 | Pocket 1 | Pocket 2 | Pocket 3 | Pocket 4 | Pocket 5 | Pocket 6 | Pocket 7 | Pocket 8 | Pocket 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 4538 | | | | | 4538 | | | | | |
| 5438 | | | | | | 5438 | | | | |
| 5423 | | | | | | 5423 | | | | |
| 4361 | | | | | 4361 | | | | | |
| 4348 | | | | | 4348 | | | | | |
| 2143 | | | 2143 | | | | | | | |
| 2128 | | | 2128 | | | | | | | |

iv. Pass 4

Now collect the values of pockets from $9^{th}$ to $0^{th}$ then we get the list of array as

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 5438 | 5423 | 4538 | 4361 | 4348 | 2143 | 2128 |

**Complexity:**

It runs in O(d(n + N)) time, where [0, N-1] is the range of integer keys ( and d=1) Thus if d(n + N) is "below" n log n (formally d(n + N) is o(n log n) using the little o notation, then this sorting method should run faster than even quick sort or heap sort.