Searching:

Searching is the technique of finding desired data items that has been stored within some data structure. Data structures can include linked lists, arrays, search trees, hash tables, or various other storage methods.

The appropriate search algorithm often depends on the data structure being searched.

Search algorithms can be classified based on their mechanism of searching.

Searching Types:

- **Linear searching**
- **Binary searching**

Linear Search/Sequential Search

Linear Search is the most natural searching method and It is very simple but very poor in performance at times

In this method, the searching begins with searching every element of the list till the required record is found. The elements in the list may be in any order. i.e. sorted or unsorted.

We begin search by comparing the first element of the list with the target element. If it matches, the search ends and position of the element is returned. Otherwise, we will move to next element and compare. In this way, the target element is compared with all the elements until a match occurs. If the match do not occur and there are no more elements to be compared, we conclude that target element is absent in the list by returning position as -1.

Example:

55 95 75 85 11 25 65 45

Suppose we want to search for element 11(i.e. Target element = 11). We first compare the target element with first element in list i.e. 55. Since both are not matching we move on the next elements in the list and compare. Finally we will find the match after 5 comparisons at position 4 starting from position 0. Linear search can be implemented in two ways.i)Non recursive ii)recursive

Algorithm

Linear_Search (A[ ], N, val , pos)

Step 1 : Set pos = -1 and k = 0

Step 2 : Repeat while k < N

Begin

Step 3 : if A[k] = val Set pos = k print pos Goto step 5

End while

Step 4 : print ―Value is not present‖

Step 5 : Exit

Linear Search Program

```
#include<iostream> using namespace std;

int main()

{

int arr[10], i, num, index; cout<<"Enter 10 Numbers: ";

for(i=0; i<10; i++)

 cin>>arr[i];

cout<<"\nEnter a Number to Search: ";

cin>>num;

for(i=0; i<10; i++)

{
```

```
if(arr[i]==num)
{
index = i;
break;
}
}
cout<<"\nFound at Index No."<<index; cout<<endl; return 0;
}
```

Binary Search

Binary search is a fast search algorithm with run-time complexity of O(log n).

This search algorithm works on the principle of divide and conquer.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the subarray to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Before applying binary searching, the list of items should be sorted in ascending or descending order.

Best case time complexity is O(1)

Worst case time complexity is 0(logn)

Algorithm

Binary_Search (A [ ], U_bound, VAL)

Step 1 : set BEG = 0 , END = U_bound , POS = -1

Step 2 : Repeat while (BEG <= END)

Step 3 : set MID = (BEG + END) / 2

Step 4 : if A [MID] == VAL then

POS = MID

print VAL — is available at —, POS GoTo

Step 6 End if if A [MID] > VAL then set END = MID – 1 Else

set BEG = MID + 1

End if

End while

Step 5 : if POS = -1 then print VAL — is not present — End if

Step 6 : EXIT

Binary Search Program

```
#include <iostream> using namespace std;
int binarySearch(int array[], int x, int low, int high) {
// Repeat until the pointers low and high meet each other
while (low <= high) { int mid = low + (high - low) / 2;
if (array[mid] == x) return mid;
if (array[mid] < x) low = mid + 1;
else
high = mid - 1;
} return -1;
}
int main(void) {
int array[] = {3, 4, 5, 6, 7, 8, 9}; int x = 4;
int n = sizeof(array) / sizeof(array[0]);
 int result = binarySearch(array, x, 0, n - 1); if (result == -1)
cout<<"Not found";
else
```

cout<<"Element is found at index<< result; }

Hashing

Hashing is a technique using which we can map a large amount of data to a smaller table using a "hash function".

Using the hashing technique, we can search the data more quickly and efficiently when compared to other searching techniques like linear and binary search.

Hash Function

We already mentioned that the efficiency of mapping depends on the efficiency of the hash function that we use.

A hash function basically should fulfill the following requirements:

Easy to Compute: A hash function, should be easy to compute the unique keys.

Less Collision: When elements equate to the same key values, there occurs a collision. There should be minimum collisions as far as possible in the hash function that is used. As collisions are bound to occur, we have to use appropriate collision resolution techniques to take care of the collisions.

Uniform Distribution: Hash function should result in a uniform distribution of data across the hash table and thereby prevent clustering.

Hash Table

Hash table or a hash map is a data structure that stores pointers to the elements of the original data array.

In our library example, the hash table for the library will contain pointers to each of the books in the library.

Having entries in the hash table makes it easier to search for a particular element in the array.

the hash table uses a hash function to compute the index into the array of buckets or slots using which the desired value can be found.

Implement Hash Tables:

A hash table is a data structure which is used to store key-value pairs. Hash function is used by hash table to compute an index into an array in which an element will be inserted or searched.

This is a C++ program to Implement Hash Tables.

Algorithm

Begin

Initialize the table size T_S to some integer value.

Create a structure hashTableEntry to declare key k and value v.

Create a class hashMapTable:

Create a constructor hashMapTable to create the table.

Create a hashFunc() function which return key mod T_S.

Create a function Insert() to insert element at a key.

Create a function SearchKey() to search element at a key.

Create a function Remove() to remove element at a key.

Call a destructor hashMapTable to destroy the objects created by the constructor.

In main, perform switch operation and enter input as per choice.

To insert key and values, call insert().

To search element, call SearchKey().

To remove element, call Remove().

End.

Example Code

```
#include<iostream>
#include<cstdlib>
#include<string>
#include<cstdio>
using namespace std;
const int T_S = 200;
class HashTableEntry {
```

```
  public:
    int k;
    int v;
    HashTableEntry(int k, int v) {
      this->k= k;
      this->v = v;
    }
};
class HashMapTable {
  private:
    HashTableEntry **t;
  public:
    HashMapTable() {
      t = new HashTableEntry * [T_S];
      for (int i = 0; i< T_S; i++) {
        t[i] = NULL;
      }
    }
    int HashFunc(int k) {
      return k % T_S;
    }
    void Insert(int k, int v) {
      int h = HashFunc(k);
      while (t[h] != NULL && t[h]->k != k) {
        h = HashFunc(h + 1);
```

```
    }
    if (t[h] != NULL)
      delete t[h];
    t[h] = new HashTableEntry(k, v);
  }
  int SearchKey(int k) {
    int h = HashFunc(k);
    while (t[h] != NULL && t[h]->k != k) {
      h = HashFunc(h + 1);
    }
    if (t[h] == NULL)
      return -1;
    else
      return t[h]->v;
  }
  void Remove(int k) {
    int h = HashFunc(k);
    while (t[h] != NULL) {
      if (t[h]->k == k)
        break;
      h = HashFunc(h + 1);
    }
    if (t[h] == NULL) {
      cout<<"No Element found at key "<<k<<endl;
      return;
```

```
    } else {
      delete t[h];
    }
    cout<<"Element Deleted"<<endl;
  }
  ~HashMapTable() {
    for (int i = 0; i < T_S; i++) {
      if (t[i] != NULL)
        delete t[i];
        delete[] t;
    }
  }
};
int main() {
  HashMapTable hash;
  int k, v;
  int c;
  while (1) {
    cout<<"1.Insert element into the table"<<endl;
    cout<<"2.Search element from the key"<<endl;
    cout<<"3.Delete element at a key"<<endl;
    cout<<"4.Exit"<<endl;
    cout<<"Enter your choice: ";
    cin>>c;
    switch(c) {
```

```
case 1:
   cout<<"Enter element to be inserted: ";
   cin>>v;
   cout<<"Enter key at which element to be inserted: ";
   cin>>k;
   hash.Insert(k, v);
break;
case 2:
   cout<<"Enter key of the element to be searched: ";
   cin>>k;
   if (hash.SearchKey(k) == -1) {
      cout<<"No element found at key "<<k<<endl;
      continue;
   } else {
      cout<<"Element at key "<<k<<" : ";
      cout<<hash.SearchKey(k)<<endl;
   }
break;
case 3:
   cout<<"Enter key of the element to be deleted: ";
   cin>>k;
   hash.Remove(k);
break;
case 4:
   exit(1);
```

```
        default:
            cout<<"\nEnter correct option\n";
      }
   }
   return 0;
}
```

Output

```
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 1
Enter element to be inserted: 1
Enter key at which element to be inserted: 1
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 1
Enter element to be inserted: 2
Enter key at which element to be inserted: 2
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
```

```
Enter your choice: 1
Enter element to be inserted: 4
Enter key at which element to be inserted: 5
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 1
Enter element to be inserted: 7
Enter key at which element to be inserted: 6
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 2
Enter key of the element to be searched: 7
No element found at key 7
1.Insert element into the table
2.Search element from the key
3.Delete element at a key
4.Exit
Enter your choice: 2
Enter key of the element to be searched: 6
Element at key 6 : 7
1.Insert element into the table
```

2.Search element from the key

3.Delete element at a key

4.Exit

Enter your choice: 3

Enter key of the element to be deleted: 1

Element Deleted

1.Insert element into the table

2.Search element from the key

3.Delete element at a key

4.Exit

Enter your choice: 4

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-

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

54%10=4

72%10=2

89%10=9

37%10=7

Hash Function

Hash Function is used to index the original value or key and then used later each time the data associated with the value or key is to be retrieved. Thus, hashing is always a one-way operation. There is no need to "reverse engineer" the hash function by analyzing the hashed values.

Characteristics of Good Hash Function:

1. The hash value is fully determined by the data being hashed.
2. The hash Function uses all the input data.
3. The hash function "uniformly" distributes the data across the entire set of possible hash values.
4. The hash function generates complicated hash values for similar strings.

Some Popular Hash Function is:

1. Division Method:

Choose a number m smaller than the number of n of keys in k (The number m is usually chosen to be a prime number or a number without small divisors, since this frequently a minimum number of collisions).

The hash function is:

$h(k) = k \bmod m$	$h(k) = k \bmod m + 1$

For Example: if the hash table has size m = 12 and the key is k = 100, then h (k) = 4. Since it requires only a single division operation, hashing by division is quite fast.

2. Multiplication Method:

The multiplication method for creating hash functions operates in two steps. First, we multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA. Then, we increase this value by m and take the floor of the result.

The hash function is:

$$h(k) = \lfloor m(k A \bmod 1) \rfloor$$

Where "k A mod 1" means the fractional part of k A, that is, $k A - \lfloor k A \rfloor$.

3. Mid Square Method:

The key k is squared. Then function H is defined by

1. H (k) = L

Where L is obtained by deleting digits from both ends of k^2. We emphasize that the same position of k^2 must be used for all of the keys.

4. Folding Method:

The key k is partitioned into a number of parts k_1, k_2.... k_n where each part except possibly the last, has the same number of digits as the required address.

Then the parts are added together, ignoring the last carry.

$H(k) = k^1 + k^2 + ... + k^n$

Example: Company has 68 employees, and each is assigned a unique four- digit employee number. Suppose L consist of 2- digit addresses: 00, 01, and 02....99. We apply the above hash functions to each of the following employee numbers:

1. 3205, 7148, 2345

(a) Division Method: Choose a Prime number m close to 99, such as m =97, Then

1. H (3205) = 4, H (7148) = 67, H (2345) = 17.

That is dividing 3205 by 17 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, dividing 2345 by 97 gives a remainder of 17.

(b) Mid-Square Method:

k = 3205 7148 2345

k^2= 10272025 51093904 5499025

h (k) = 72 93 99

Observe that fourth & fifth digits, counting from right are chosen for hash address.

(c) Folding Method: Divide the key k into 2 parts and adding yields the following hash address:

Learn more

1. H (3205) = 32 + 50 = 82 H (7148) = 71 + 84 = 55
2. H (2345) = 23 + 45 = 68

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

131%10=1

44%10=4

43%10=3

78%10=8

19%10=9

36%10=6

57%10=7

77%10=7

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 toplace 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 atechnique is called collision handling technique.

- Chaining
- Open addressing (linear probing)
- Quadratic probing
- Double hashing
- Double hashing
- Rehashing

CHAINING

In collision handling method chaining is a concept which introduces an additional field withdata 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 bucketsare 131, 3, 4, 21, 61, 7, 97, 8, 9
then we will apply a hash function as H(key) = key % DWhere 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 table131, 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 % tablesizeH(key) = key % 10

For instance the element 131 can be placed atH(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.

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.

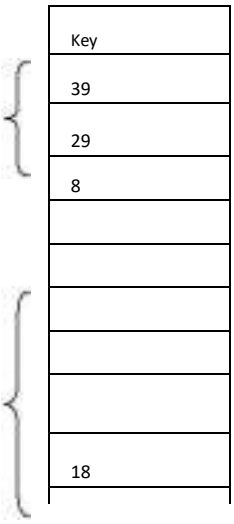
19%10 = 9 cluster is formed

18%10 = 8

39%10 = 9

29%10 = 9

8%10 = 8



rest of the table is emptythis 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 9

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

Now to place 87 we will use quadratic probing.

(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 in the hash table the size of divisor (m) should be twice as large as total number of elements.

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

$H_1(key) = key \bmod tablesize$

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

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

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

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

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

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

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

Key
90
22
45
37
49

Now if 17 to be inserted then

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

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

Key
90
17
22
45
37
49

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

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

Now to insert number 55

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

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

$= 7 - 6 = 1$

Key
90
17
22
45
55
37
49

That means we have to take one jump from index 5 to place 55.

Comparison of Quadratic Probing & Double Hashing

The double hashing requires another hash function whose probing efficiency is same assome 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.

Priority Queues:

The queue which is implemented as FIFO where insertions are done at one end (rear) and deletions are done from another end (front). The first element that entered is deleted first.

Queue operations are

EnQueue (int data): Insertion at rear end

int DeQueue(): Deletion from front end

But a priority queue doesn't follow First-In-First-Out, but rather than each element has a priority based on the basis of urgency.

Items with the same priority are processed on First-In-First-Out service basis.

An item with higher priority is processed before other items with lower priority.

Algorithm

Begin

class Priority_Queue has following functions:

function insert() to insert items at priority queue with their priorities:

1) If queue is empty insert data from the left end of the queue.

2) If queue is having some nodes then insert the new node at the end of those nodes having priority same with the new node and also before all the nodes having priority lesser than the current priority of the new node.

function del() to delete items from queue.

If queue is completely empty, print underflow otherwise delete the front element and update front.

End

Proram

```
#include <iostream>
```

```
#include <cstdio>
#include <cstring>
#include <cstdlib>
using namespace std;
struct n  {
   int p;
   int info;
   struct n *l;
};
class Priority_Queue {
   private:
      //Declare a front pointer f and initialize it to NULL.
      n *f;
   public:
```

```
Priority_Queue() {
   f = NULL;
}
void insert(int i, int p) {
   n *t, *q;
   t = new n;
   t->info = i;
   t->p = p;
   if (f == NULL || p < f->p) {
      t->l= f;
      f = t;
   } else {
      q = f;
      while (q->l != NULL && q->l->p <= p)
```

```
        q = q->l;
        t->l = q->l;
        q->l = t;
    }
}
void del() {
    n *t;
    if(f == NULL) //if queue is null
        cout<<"Queue Underflow\n";
    else {
        t = f;
        cout<<"Deleted item is: "<<t->info<<endl;
        f = f->l;
        free(t);
```

```
    }
}
void show() //print queue {
    n *ptr;
    ptr = f;
    if (f == NULL)
        cout<<"Queue is empty\n";
    else {
        cout<<"Queue is :\n";
        cout<<"Priority Item\n";
        while(ptr != NULL) {
            cout<<ptr->p<<" "<<ptr->info<<endl;
            ptr = ptr->l;
        }
```

```
        }
      }
};
int main() {
   int c, i, p;
   Priority_Queue pq;
   Do//perform switch opeartion {
      cout<<"1.Insert\n";
      cout<<"2.Delete\n";
      cout<<"3.Display\n";
      cout<<"4.Exit\n";
      cout<<"Enter your choice : ";
      cin>>c;
      switch(c) {
```

```
case 1:
   cout<<"Input the item value to be added in the queue : ";
   cin>>i;
   cout<<"Enter its priority : ";
   cin>>p;
   pq.insert(i, p);
   break;
case 2:
   pq.del();
   break;
case 3:
   pq.show();
   break;
case 4:
```

```
            break;
        default:
        cout<<"Wrong choice\n";
      }
   }
   while(c != 4);
   return 0;
}
```

Output

```
1.Insert
2.Delete
3.Display
4.Exit
Enter your choice : 1
```

Input the item value to be added in the queue : 7

Enter its priority : 2

1.Insert

2.Delete

3.Display

4.Exit

Enter your choice : 1

Input the item value to be added in the queue : 6

Enter its priority : 1

1.Insert

2.Delete

3.Display

4.Exit

Enter your choice : 1

Input the item value to be added in the queue : 3

Enter its priority : 3

1.Insert

2.Delete

3.Display

4.Exit

Enter your choice : 1

Input the item value to be added in the queue : 4

Enter its priority : 3

1.Insert

2.Delete

3.Display

4.Exit

Enter your choice : 3

Queue is :

Priority Item

1 6

2 7

3 3

3 4

1.Insert

2.Delete

3.Display

4.Exit

Enter your choice : 4

Max Priority Queue:

In a max priority queue, elements are inserted in the order in which they arrive the queue and the maximum value is always removed first from the queue. For example, assume that we insert in the order 8, 3, 2 & 5 and they are removed in the order 8, 5, 3, 2.

Priority Queue in C++ Standard Template Library (STL)

A C++ priority queue is a type of container adapter, specifically designed such that the first element of the queue is either the greatest or the smallest of all elements in the queue, and elements are in non-increasing or non-decreasing order (hence we can see that each element of the queue has a priority {fixed order}).

In C++ STL, the top element is always the greatest by default. We can also change it to the smallest element at the top. Priority queues are built on the top of the max heap and use an array or vector as an internal structure. In simple terms, STL Priority Queue is the implementation of Heap Data Structure.

Syntax:

std::priority_queue<int> pq;

```
// C++ program to demonstrate the use of priority_queue
#include <iostream>
#include <queue>
using namespace std;

// driver code
int main()
{
        int arr[6] = { 10, 2, 4, 8, 6, 9 };

        // defining priority queue
        priority_queue<int> pq;

        // printing array
```

```
cout << "Array: ";
for (auto i : arr) {
        cout << i << ' ';
}
cout << endl;
// pushing array sequentially one by one
for (int i = 0; i < 6; i++) {
        pq.push(arr[i]);
}

// printing priority queue
cout << "Priority Queue: ";
while (!pq.empty()) {
        cout << pq.top() << ' ';
```

```
            pq.pop();
        }


        return 0;
}
```

Max Priority Queue

In a max priority queue, elements are inserted in the order in which they arrive the queue and the maximum value is always removed first from the queue. For example, assume that we insert in the order 8, 3, 2 & 5 and they are removed in the order 8, 5, 3, 2.

Another Way of operations performed in a Max priority queue.

isEmpty() - Check whether queue is Empty.

insert() - Inserts a new value into the queue.

findMax() - Find maximum value in the queue.

remove() - Delete maximum value from the queue.

Heap:

A heap is a tree based data structure in which tree should be almost complete. It is of two types i.e. max and min heap.

Max heap: In max heap, if p is the parent and c is its child, then for every parent p the value of it is greater than or equal to the value of c

Min heap: In min heap, if p is the parent and c is its child, then for every parent p value is less than or equal to the value of c.

Heap is also used as priority queue. In which highest(in case of max heap) or lowest(in case of min heap) element is present at the root.

Heap is used in problems where we need to remove the highest or lowest priority element. A common implementation of heap is binary heap.

Implementation

Although heap can be implemented as a tree, but lots of storage will go waste for storing pointers. Due to the property of heap being a complete binary tree, it can be easily stored in an array.

Where root element is stored at first index and its child index can be calculated as following.

Left child index = $2 \times r$ where r is index of the root and array starting index is 1 .

Right child index = 2×r+1.

And parent index can be calculated as floor (i/2) where i is the index of its left or right child.

Deletion in Heaps

The standard deletion operation on Heap is to delete the element present at the root node of the Heap. That is if it is a Max Heap, the standard deletion operation will delete the maximum element and if it is a Min heap, it will delete the minimum element.

Process of Deletion:

Since deleting an element at any intermediary position in the heap can be costly, so we can simply replace the element to be deleted by the last element and delete the last element of the Heap.

Replace the root or element to be deleted by the last element.

Delete the last element from the Heap.

Since, the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, heapify the last node placed at the position of root.

```
#include <iostream>
 using namespace std;
 // To heapify a subtree rooted with node i which is
// an index of arr[] and n is the size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;
```

```
    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
```

```
// Function to delete the root from Heap
void deleteRoot(int arr[], int& n)
{
    // Get the last element
    int lastElement = arr[n - 1];

    // Replace root with last element
    arr[0] = lastElement;

    // Decrease size of heap by 1
    n = n - 1;

    // heapify the root node
    heapify(arr, n, 0);
```

```
}
 /* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}
 // Driver Code
int main()
{
    int arr[] = { 10, 5, 3, 2, 4 };
     int n = sizeof(arr) / sizeof(arr[0]);
```

```
    deleteRoot(arr, n);

    printArray(arr, n);

    return 0;

}
```

Time complexity: O(logn) where n is no of elements in the heap

Auxiliary Space: O(n)

Insertion in Heaps

The insertion operation is also similar to that of the deletion process.

Given a Binary Heap and a new element to be added to this Heap. The task is to insert the new element to the Heap maintaining the properties of Heap.

Process of Insertion: Elements can be inserted to the heap following a similar approach as discussed for deletion. The idea is to:

First increase the heap size by 1, so that it can store the new element.

Insert the new element at the end of the Heap.

This newly inserted element may distort the properties of Heap for its parents. So, in order to keep the properties of Heap, heapify this newly inserted element following a bottom-up approach.

```
#include <iostream>

using namespace std;

 #define MAX 1000 // Max size of Heap

// Function to heapify ith node in a Heap

// of size n following a Bottom-up approach

void heapify(int arr[], int n, int i)

{

    // Find parent

    int parent = (i - 1) / 2;


    if (arr[parent] > 0) {

        // For Max-Heap
```

```
        // If current node is greater than its parent
        // Swap both of them and call heapify again
        // for the parent
        if (arr[i] > arr[parent]) {
            swap(arr[i], arr[parent]);
            // Recursively heapify the parent node
            heapify(arr, n, parent);
        }
    }
}

// Function to insert a new node to the Heap
void insertNode(int arr[], int  n, int Key)
{
```

```
    // Increase the size of Heap by 1
    n = n + 1;
    // Insert the element at end of Heap
    arr[n - 1] = Key;
   // Heapify the new node following a
    // Bottom-up approach
    heapify(arr, n, n - 1);
}
 // A utility function to print array of size n
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
```

```
    cout << "\n";
}
 // Driver Code
int main()
{
    int arr[MAX] = { 10, 5, 3, 2, 4 };
   int n = 5;
    int key = 15;
    insertNode(arr, n, key);
    printArray(arr, n);
       return 0;
}
```

Time Complexity: O(log(n)) (where n is no of elements in the heap)

Auxiliary Space: O(n)

Heap Sort:

Heap Sort is based on the binary heap data structure. In the binary heap the child nodes of a parent node are smaller than or equal to it in the case of a max heap, and the child nodes of a parent node are greater than or equal to it in the case of a min heap.

An example that explains all the steps in Heap Sort is as follows.

The original array with 10 elements before sorting is −

20 7 1 54 10 15 90 23 77 25

This array is built into a binary max heap using max-heapify. This max heap represented as an array is given as follows.

90 77 20 54 25 15 1 23 7 10

The root element of the max heap is extracted and placed at the end of the array. Then max heapify is called to convert the rest of the elements into a max heap. This is done until finally the sorted array is obtained which is given as follows −

1 7 10 15 20 23 25 54 77 90

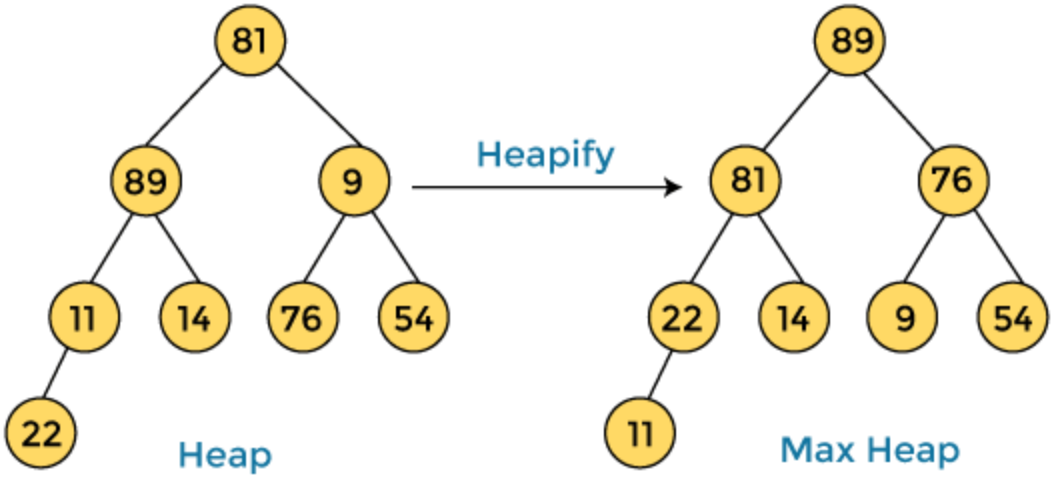
The program to sort an array of 10 elements using the heap sort algorithm is given as follows.
Working of Heap sort Algorithm

In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

The first step includes the creation of a heap by adjusting the elements of the array.

After the creation of heap, now remove the root element of the heap repeatedly by shifting it to the end of the array, and then store the heap structure with the remaining elements.

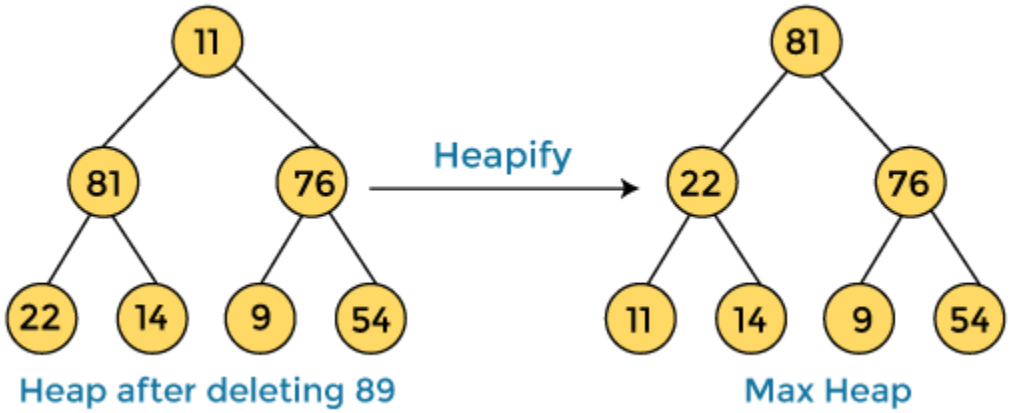
First, we have to construct a heap from the given array and convert it into max heap.



After converting the given heap into max heap, the array elements are -

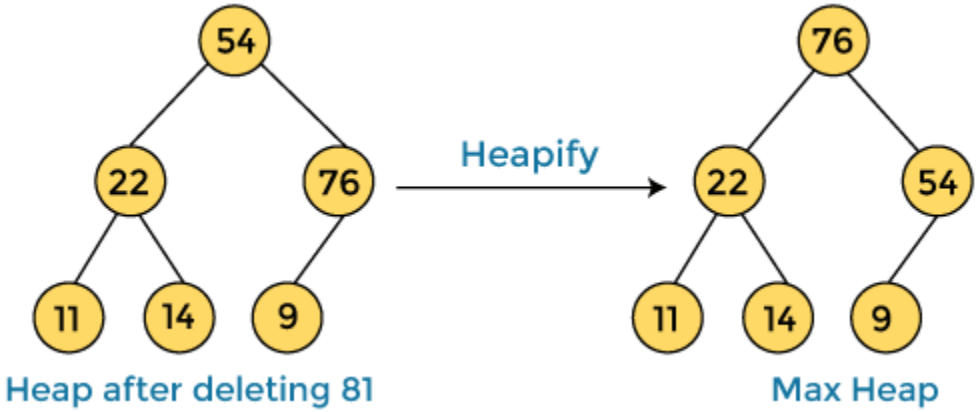
89	81	76	22	14	9	54	11

Next, we have to delete the root element (89) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.



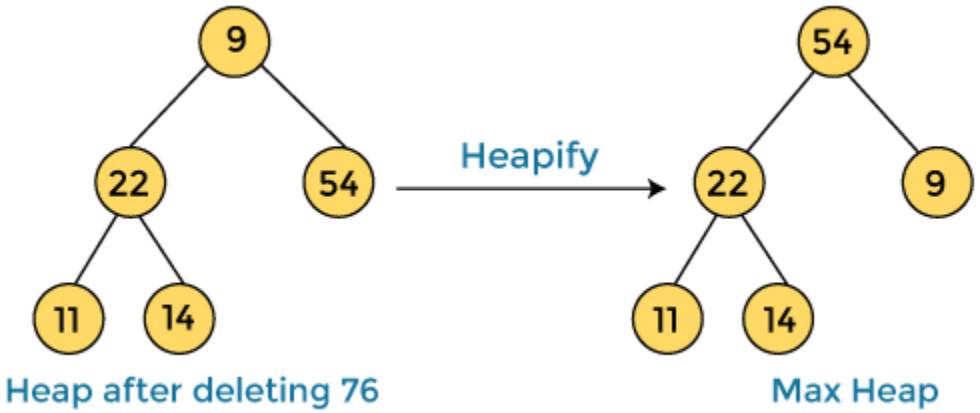
After swapping the array element 89 with 11, and converting the heap into max-heap, the elements of array are -

In the next step, again, we have to delete the root element (81) from the max heap. To delete this node, we have to swap it with the last node, i.e. (54). After deleting the root element, we again have to heapify it to convert it into max heap.



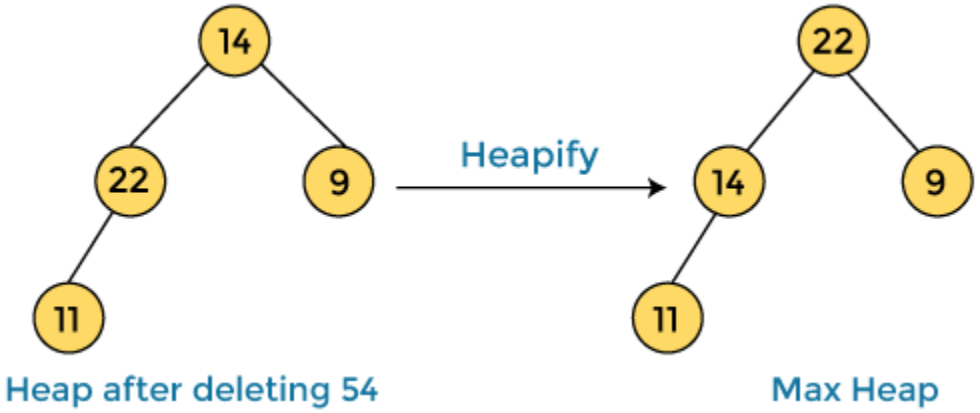
After swapping the array element 81 with 54 and converting the heap into max-heap, the elements of array are -

In the next step, we have to delete the root element (76) from the max heap again. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.

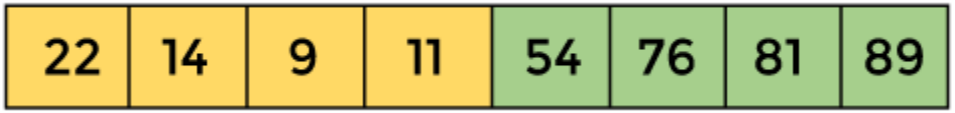


After swapping the array element 76 with 9 and converting the heap into max-heap, the elements of array are -

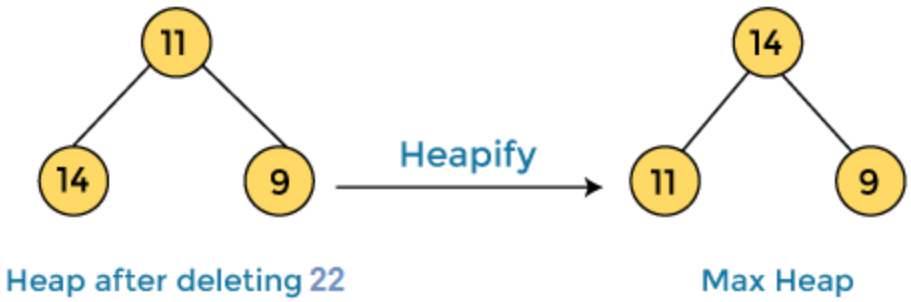
In the next step, again we have to delete the root element (54) from the max heap. To delete this node, we have to swap it with the last node, i.e. (14). After deleting the root element, we again have to heapify it to convert it into max heap.



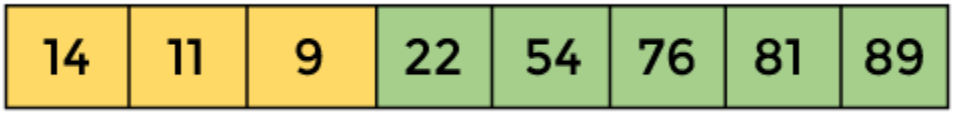
After swapping the array element 54 with 14 and converting the heap into max-heap, the elements of array are -



In the next step, again we have to delete the root element (22) from the max heap. To delete this node, we have to swap it with the last node, i.e. (11). After deleting the root element, we again have to heapify it to convert it into max heap.

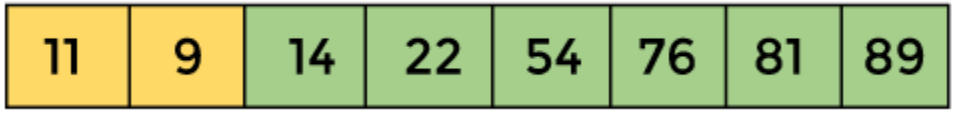


After swapping the array element 22 with 11 and converting the heap into max-heap, the elements of array are -



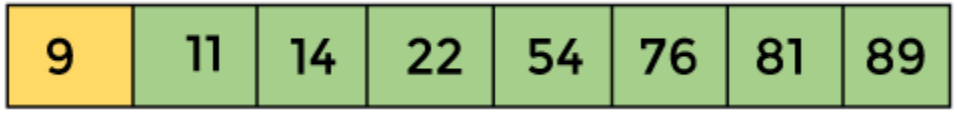
In the next step, again we have to delete the root element (14) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element 14 with 9 and converting the heap into max-heap, the elements of array are -



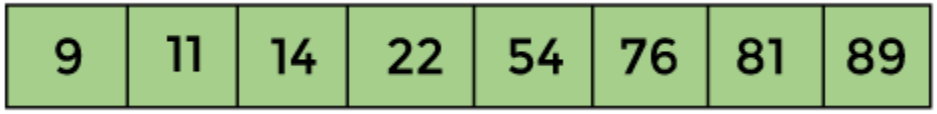
In the next step, again we have to delete the root element (11) from the max heap. To delete this node, we have to swap it with the last node, i.e. (9). After deleting the root element, we again have to heapify it to convert it into max heap.

After swapping the array element 11 with 9, the elements of array are -



Now, heap has only one element left. After deleting it, heap will be empty.

After completion of sorting, the array elements are -



Now, the array is completely sorted.

Heap sort complexity

Now, let's see the time complexity of Heap sort in the best case, average case, and worst case. We will also see the space complexity of Heapsort.

1. Time Complexity

Case	Time Complexity
Best Case	O(n logn)
Average Case	O(n log n)
Worst Case	O(n log n)

The space complexity of Heap sort is O(1).

Example:-01

```
#include<iostream>
using namespace std;
void heapify(int arr[], int n, int i) {
   int temp;
   int largest = i;
   int l = 2 * i ;
   int r = 2 * i+1;
   if (l <= n && arr[l] > arr[largest])
      largest = l;
   if (r < =n && arr[r] > arr[largest])
      largest = r;
   if (largest != i) {
      temp = arr[i];//swap(arr[largest],arr[i])
      arr[i] = arr[largest];
```

```
      arr[largest] = temp;
      heapify(arr, n, largest);
    }
  }
  void heapSort(int arr[], int n) {
    int temp;
    for (int i = n / 2 ; i >= 1; i--)
      heapify(arr, n, i);
    for (int i = n ; i >= 1; i--) {
      temp = arr[1];//swap(arr[1],arr[i])
      arr[1] = arr[i];
      arr[i] = temp;
      heapify(arr, n, 1);
    }
```

```
}
int main() {
   int arr[] = { 20, 7, 1, 54, 10, 15, 90, 23, 77, 25};
   int n = 10;
int i;
   cout<<"Given array is: "<<endl;
   for (i = 0; i < n; i++)
   cout<<arr[i]<<" ";
   cout<<endl;
   heapSort(arr, n);
   printf("\nSorted array is: \n");
   for (i = 0; i < n; ++i)
   cout<<arr[i]<<" ";
}
```

Output

Given array is:

20 7 1 54 10 15 90 23 77 25

Sorted array is:

1 7 10 15 20 23 25 54 77 90

Example:-02

```
#include<iostream>
using namespace std;
void heapify(int arr[], int n, int i) {
   int temp;
   int largest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;
   if (l < n && arr[l] > arr[largest])
```

```
    largest = l;
  if (r < n && arr[r] > arr[largest])
    largest = r;
  if (largest != i) {
    temp = arr[i];
    arr[i] = arr[largest];
    arr[largest] = temp;
    heapify(arr, n, largest);
  }
}
void heapSort(int arr[], int n) {
  int temp;
  for (int i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);
```

```
   for (int i = n - 1; i >= 0; i--) {
      temp = arr[0];
      arr[0] = arr[i];
      arr[i] = temp;
      heapify(arr, i, 0);
   }
}
int main() {
   int arr[] = { 20, 7, 1, 54, 10, 15, 90, 23, 77, 25};
   int n = 10;
int i;
   cout<<"Given array is: "<<endl;
   for (i = 0; i *lt; n; i++)
   cout<<arr[i]<<" ";
```

```
    cout<<endl;
    heapSort(arr, n);
    printf("\nSorted array is: \n");
    for (i = 0; i < n; ++i)
    cout<<arr[i]<<" ";
}
```

Output

```
Given array is:
20 7 1 54 10 15 90 23 77 25
Sorted array is:
1 7 10 15 20 23 25 54 77 90
```

Counting Sort Algorithm.

This sorting technique doesn't perform sorting by comparing elements. It performs sorting by counting objects having distinct key values like hashing. After that, it performs some arithmetic operations to calculate each object's index position in the output sequence. Counting sort is not used as a general-purpose sorting algorithm.

Counting sort is effective when range is not greater than number of objects to be sorted. It can be used to sort the negative input values.

Algorithm

countingSort(array, n) // 'n' is the size of array

max = find maximum element in the given array

create count array with size maximum + 1

Initialize count array with all 0's

for i = 0 to n

find the count of every unique element and

store that count at ith position in the count array

for j = 1 to max

Now, find the cumulative sum and store it in count array

for i = n to 1

Restore the array elements

Decrease the count of every restored element by 1

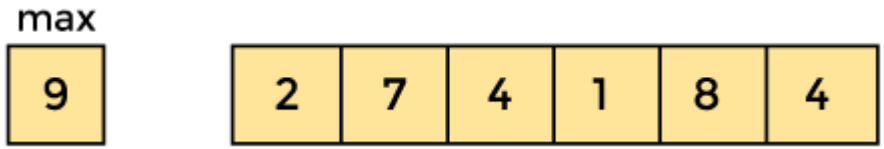
end countingSort

Working of counting sort Algorithm

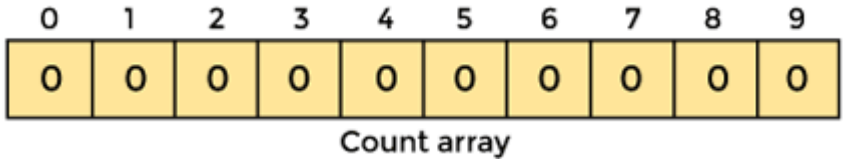
The counting sort via an example.

Let the elements of array are -

1. Find the maximum element from the given array. Let max be the maximum element.

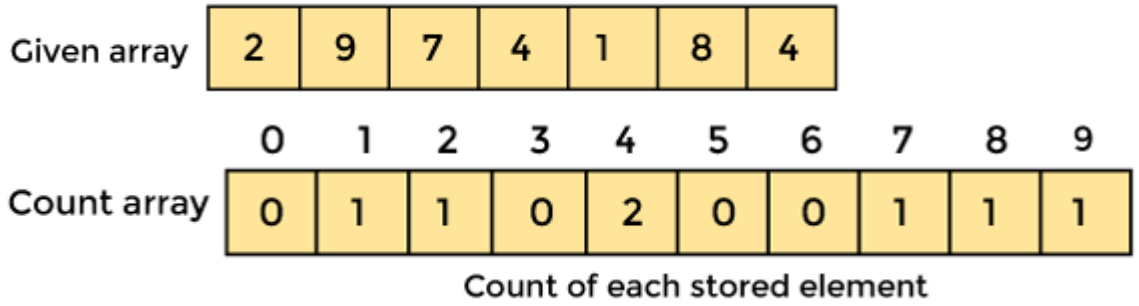


2. Now, initialize array of length max + 1 having all 0 elements. This array will be used to store the count of the elements in the given array.

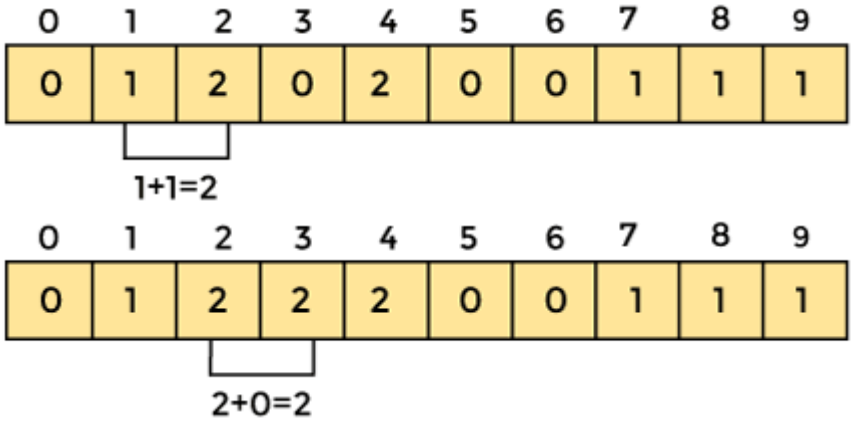


3. Now, we have to store the count of each array element at their corresponding index in the count array.

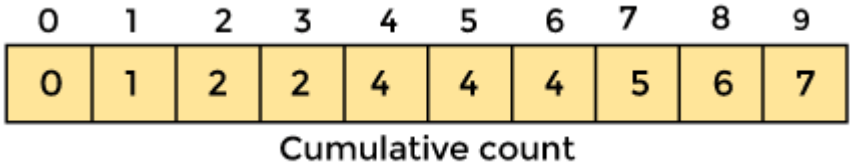
The count of an element will be stored as - Suppose array element '4' is appeared two times, so the count of element 4 is 2. Hence, 2 is stored at the 4^{th} position of the count array. If any element is not present in the array, place 0, i.e. suppose element '3' is not present in the array, so, 0 will be stored at 3^{rd} position.



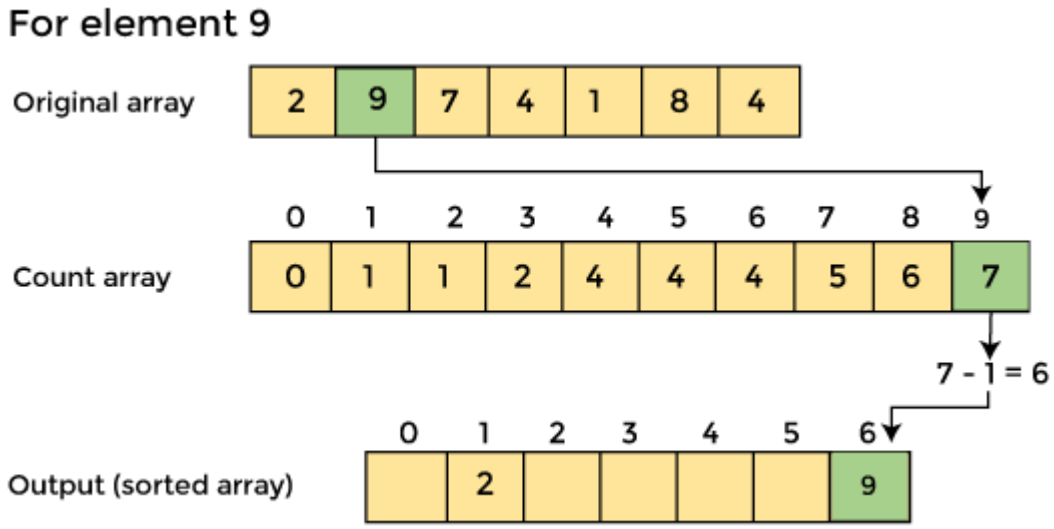
Now, store the cumulative sum of count array elements. It will help to place the elements at the correct index of the sorted array.



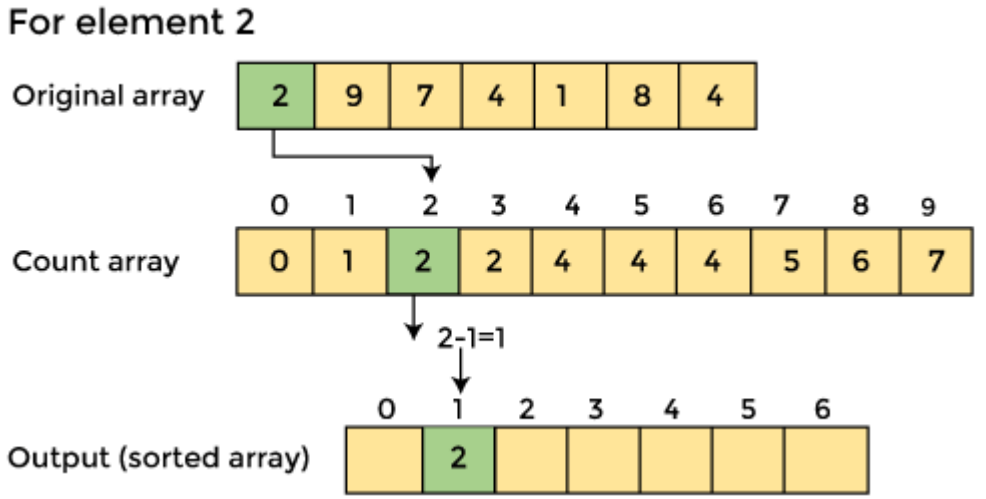
Similarly, the cumulative count of the count array is -



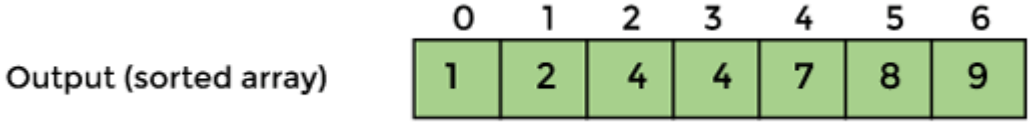
4. Now, find the index of each element of the original array



After placing element at its place, decrease its count by one. Before placing element 2, its count was 2, but after placing it at its correct position, the new count for element 2 is 1.



Similarly, after sorting, the array elements are -



Now, the array is completely sorted.

Counting sort complexity

1. Time Complexity

Case	Time	Complexity

Best Case	O(n + k)	
Average Case	O(n + k)	
Worst Case	O(n + k)	

The space complexity of counting sort is O(max). The larger the range of elements, the larger the space complexity.

Program: Write a program to implement counting sort in C++.

```
#include <iostream>

 using namespace std;

 int getMax(int a[], int n) {

  int max = a[0];

  for(int i = 1; i<n; i++) {

    if(a[i] > max)

       max = a[i];

  }
```

```
   return max; //maximum element from the array
}
void countSort(int a[], int n) // function to perform counting sort
{
   int output[n+1];
   int max = getMax(a, n);
   int count[max+1]; //create count array with size [max+1]
    for (int i = 0; i <= max; ++i)
  {
    count[i] = 0; // Initialize count array with all zeros
  }
    for (int i = 0; i < n; i++) // Store the count of each element
  {
    count[a[i]]++;
  }
```

```
   for(int i = 1; i<=max; i++)
      count[i] += count[i-1]; //find cumulative frequency
  /* This loop will find the index of each element of the original array in count array, and
   place the elements in output array*/
  for (int i = n - 1; i >= 0; i--) {
    output[count[a[i]] - 1] = a[i];
    count[a[i]]--; // decrease count for same numbers
}
   for(int i = 0; i<n; i++) {
      a[i] = output[i]; //store the sorted elements into main array
   }
}
 void printArr(int a[], int n) /* function to print the array */
{
```

```
    int i;
    for (i = 0; i < n; i++)
        cout<<a[i]<<" ";
}
 int main() {
    int a[] = { 31, 11, 42, 7, 30, 11 };
    int n = sizeof(a)/sizeof(a[0]);
    cout<<"Before sorting array elements are - \n";
    printArr(a, n);
    countSort(a, n);
    cout<<"\nAfter sorting array elements are - \n";
    printArr(a, n);
    return 0;
}
```

Output

After the execution of above code, the output will be -

```
Before sorting array elements are -
31 11 42 7 30 11
After sorting array elements are -
7 11 11 30 31 42
```

Radix Sort Algorithm

Radix sort is the linear sorting algorithm that is used for integers. In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

The process of radix sort works similar to the sorting of students names, according to the alphabetical order. In this case, there are 26 radix formed due to the 26 alphabets in English. In the first pass, the names of students are grouped according to the ascending order of the first letter of their names. After that, in the second pass, their names are grouped according to the ascending order of the second letter of their name. And the process continues until we find the sorted list.

Algorithm

radixSort(arr)

max = largest element in the given array

d = number of digits in the largest element (or, max)

Now, create d buckets of size 0 - 9

for i -> 0 to d

sort the array elements using counting sort (or any stable sort) according to the digits at

the ith place

Working of Radix sort Algorithm

The steps used in the sorting of radix sort are listed as follows -

First, we have to find the largest element (suppose max) from the given array. Suppose 'x' be the number of digits in max. The 'x' is calculated because we need to go through the significant places of all elements.

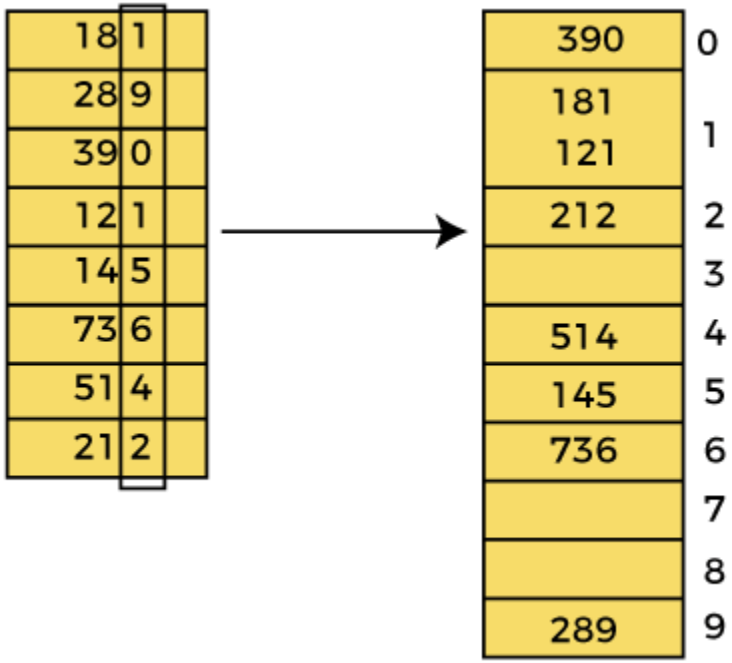
After that, go through one by one each significant place. Here, we have to use any stable sorting algorithm to sort the digits of each significant place.

In the given array, the largest element is 736 that have 3 digits in it. So, the loop will run up to three times (i.e., to the hundreds place). That means three passes are required to sort the array.

Now, first sort the elements on the basis of unit place digits (i.e., x = 0). Here, we are using the counting sort algorithm to sort the elements.

Pass 1:

In the first pass, the list is sorted on the basis of the digits at 0's place.

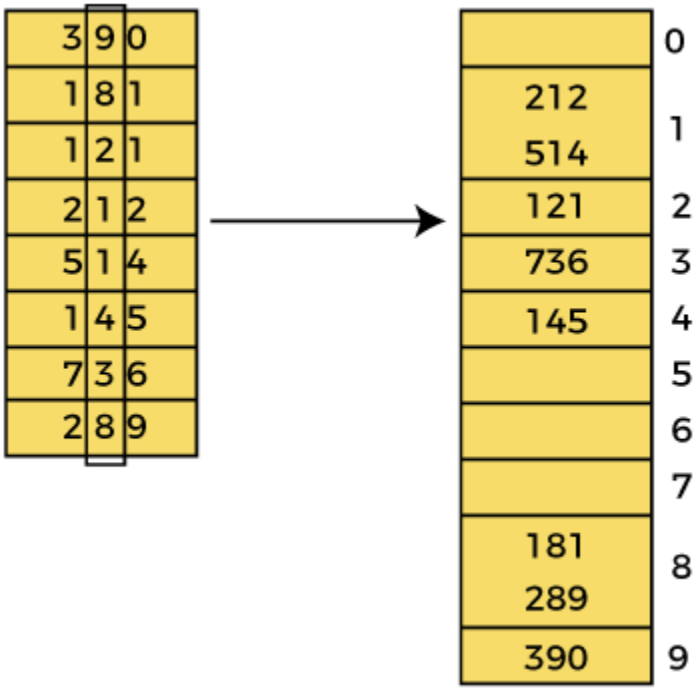


After the first pass, the array elements are -

390	181	121	212	514	145	736	289

Pass 2:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 10th place).

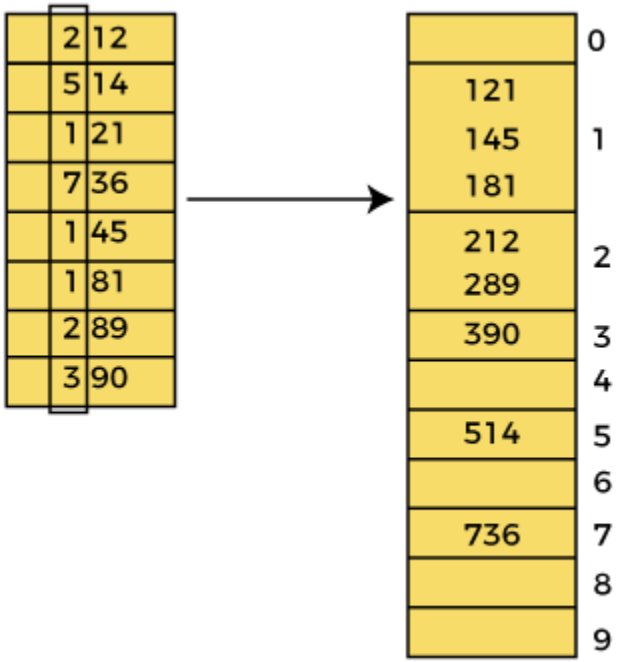


After the second pass, the array elements are -

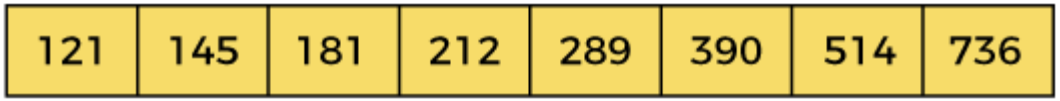
212	514	121	736	145	181	289	390

Pass 3:

In this pass, the list is sorted on the basis of the next significant digits (i.e., digits at 100th place).



After the third pass, the array elements are -



Now, the array is sorted in ascending order.

Radix sort complexity

Now, let's see the time complexity of Radix sort in best case, average case, and worst case. We will also see the space complexity of Radix sort.

1. Time Complexity

Case	Time Complexity
Best Case	Ω(n+k)
Average Case	θ(nk)
Worst Case	O(nk)

Radix sort is a non-comparative sorting algorithm that is better than the comparative sorting algorithms. It has linear time complexity that is better than the comparative algorithms with complexity O(n logn).

The space complexity of Radix sort is O(n + k)

Program: Write a program to implement Radix sort in C++.

```
#include <iostream>
```

```
using namespace std;
  int getMax(int a[], int n) {
   int max = a[0];
   for(int i = 1; i<n; i++) {
      if(a[i] > max)
         max = a[i];
   }
   return max; //maximum element from the array
}
  void countingSort(int a[], int n, int place) // function to implement counting sort
{
  int output[n + 1];
  int count[10] = {0};

  // Calculate count of elements
```

```
for (int i = 0; i < n; i++)
  count[(a[i] / place) % 10]++;

// Calculate cumulative frequency
for (int i = 1; i < 10; i++)
  count[i] += count[i - 1];

// Place the elements in sorted order
for (int i = n - 1; i >= 0; i--) {
  output[count[(a[i] / place) % 10] - 1] = a[i];
  count[(a[i] / place) % 10]--;
}

for (int i = 0; i < n; i++)
  a[i] = output[i];
```

```
}

// function to implement radix sort
void radixsort(int a[], int n) {

  // get maximum element from array
  int max = getMax(a, n);

  // Apply counting sort to sort elements based on place value
  for (int place = 1; max / place > 0; place *= 10)
    countingSort(a, n, place);
}

// function to print array elements
void printArray(int a[], int n) {
```

```
  for (int i = 0; i < n; ++i)
    cout<<a[i]<<" ";
}

int main() {
  int a[] = {171, 279, 380, 111, 135, 726, 504, 878, 112};
  int n = sizeof(a) / sizeof(a[0]);
  cout<<"Before sorting array elements are - \n";
  printArray(a,n);
  radixsort(a, n);
  cout<<"\n\nAfter applying Radix sort, the array elements are - \n";
  printArray(a, n);
  return 0;
}
```

Output:

```
Before sorting array elements are -
171 279 380 111 135 726 504 878 112

After applying Radix sort, the array elements are -
111 112 135 171 279 380 504 726 878
```

Sorting

Arranging the elements in a list either in ascending or descending order.

various sorting algorithms are:

- Quick sort
- Merge sort
- Heap sort
- Radix Sort
- Counting Sort

Quick sort:

Quick sort: It is a divide and conquer algorithm. Developed by Tony Hoare in 1959.

Quick sort first divides a large array into two smaller sub-arrays: the low elements and the high elements.

Quick sort can then recursively sort the sub-arrays.

ALGORITHM:

Step 1: Pick an element, called a pivot, from the array.

Step 2: Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

Step 3: Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2), respectively.

There are many different versions of quickSort that pick pivot in different ways.

Always pick the first element as a pivot

Always pick the last element as the pivot.

Pick a random element as a pivot.

Program

```
#include <iostream>
using namespace std;
int partition(int arr[], int start, int end)
{
    int pivot = arr[start];
    int count = 0;
    for (int i = start + 1; i <= end; i++) {
        if (arr[i] <= pivot)
            count++;
    }
    // Giving pivot element its correct position
    int pivotIndex = start + count;
    swap(arr[pivotIndex], arr[start]);
    // Sorting left and right parts of the pivot element
```

```
    int i = start, j = end;
     while (i < pivotIndex && j > pivotIndex) {
        while (arr[i] <= pivot) {
            i++;
        }
 while (arr[j] > pivot) {
            j--;
        }
        if (i < pivotIndex && j > pivotIndex) {
            swap(arr[i++], arr[j--]);
        }
    }
    return pivotIndex;
}
void quickSort(int arr[], int start, int end)
```

```
{
    // base case
    if (start >= end)
        return;
    // partitioning the array
    int p = partition(arr, start, end);
    // Sorting the left part
    quickSort(arr, start, p - 1);
    // Sorting the right part
    quickSort(arr, p + 1, end);
}
int main()
{    int arr[] = { 9, 3, 4, 2, 1, 8 };
    int n = 6;
    quickSort(arr, 0, n - 1);
```

```
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Example of Quick Sort:

44 33 11 55 77 90 40 60 99 22 88

Let 44 be the Pivot element and scanning done from right to left

Comparing 44 to the right-side elements, and if right-side elements are smaller than 44, then swap it. As 22 is smaller than 44 so swap them.

22 33 11 55 77 90 40 60 99 44 88

Now comparing 44 to the left side element and the element must be greater than 44 then swap them. As 55 are greater than 44 so swap them.

22 33 11 44 77 90 40 60 99 55 88

Recursively, repeating steps 1 & steps 2 until we get two lists one left from pivot element 44 & one right from pivot element.

22 33 11 40 77 90 44 60 99 55 88

Swap with 77:

22 33 11 40 44 90 77 60 99 55 88

Now, the element on the right side and left side are greater than and smaller than 44 respectively.

22 33 11 40 **44** 90 77 66 99 55 88

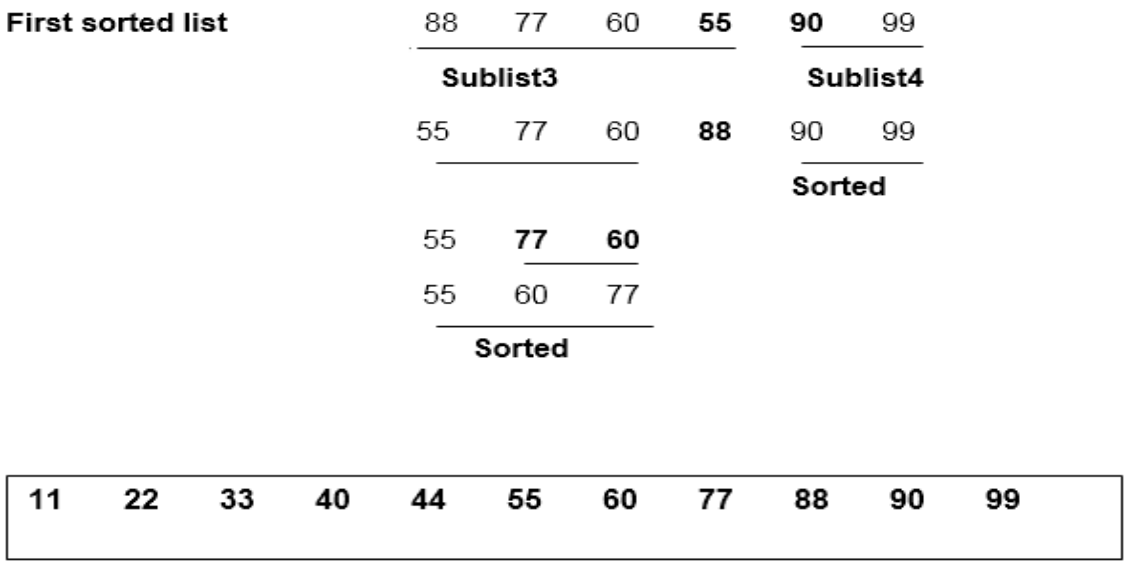
Sublist1 **Sublist2**

these sublists are sorted under the same process as above done.

22 33 11 40 **44** **90** 77 60 99 55 88

11 33 **22** 40 **44** 88 77 60 99 55 **90**

11 **22** 33 40 **44** 88 77 60 **90** 55 **99**



Method 2:

This method's space complexity is O(n). As we will take an extra array in partition function

Algorithm explanation and steps of partition function:

Make a new array of size equal to given array.

push all the smaller elements than pivotElement to the new array.

Push pivotElement to new array now.

finally, push all the greater elements than pivotElement to the new array.

Now, copy the new array to the original array.

Store the index of the pivotElement from the original array. Return this index.

After this, all the elements in the original array are in the order : smaller than pivotElement -> pivotElement -> greater than pivotElement.

Time Complexity : θ(nlogn).

Space Complexity : O(n).

Program

```
#include <iostream>

using namespace std;

int partition(int* arr, int start, int end)

{

    // assuming last element as pivotElement

   int index = 0, pivotElement = arr[end], pivotIndex;

   int* temp = new int[end - start + 1]; // making an array whose size is equal to current partition range...

   for (int i = start; i <= end; i++) // pushing all the elements in temp which are smaller than pivotElement
```

```
    {
        if(arr[i] < pivotElement)        {
            temp[index] = arr[i];
            index++;        }
    }
temp[index] = pivotElement; // pushing pivotElement in temp
    index++;
    for (int i = start; i < end; i++) // pushing all the elements in temp which are greater than pivotElement
    {
        if(arr[i] > pivotElement)
        {
            temp[index] = arr[i];
            index++;
        }    }
  // all the elements now in temp array are order :
```

```
  // leftmost elements are lesser than pivotElement and rightmost elements are greater than pivotElement
index = 0;
    for (int i = start; i <= end; i++) // copying all the elements to original array i.e arr     {
        if(arr[i] == pivotElement) {
            // for getting pivot index in the original array.
            // we need the pivotIndex value in the original and not in the temp array
           pivotIndex = i; }
        arr[i] = temp[index];
        index++;
    }
    return pivotIndex; // returning pivotIndex
}
void quickSort(int* arr, int start, int end)
{
    if(start < end)
```

```
    {
        int partitionIndex = partition(arr, start, end); // for getting partition
        quickSort(arr, start, partitionIndex - 1); // sorting left side array
        quickSort(arr, partitionIndex + 1, end); // sorting right side array
    }
    return;
}
int main() {
    int size = 9;
    int arr[size] = {5, 12, 7, 1, 13, 2 ,23, 11, 18};
      cout << "Unsorted array : ";
    for (int i = 0; i < size; i++)    {
        cout << arr[i] << " ";        }
    printf("\n");
    quickSort(arr, 0, size - 1);
```

```
    cout << "Sorted array : ";
   for (int i = 0; i < size; i++)    {
     cout << arr[i] << " ";        }
    return 0;
}
```

Quick Sort:

Example: 6 3 7 2 4 5 -----List of Elements

Quick sort

It is a divide and conquer algorithm.

Step 1 − Pick an element from an array, call it as pivot element.

Step 2 − Divide an unsorted array element into two arrays.

Step 3 − If the value less than pivot element come under first sub array, the remaining elements with value greater than pivot come in second sub array.

Consider an example given below, wherein

P is the pivot element.

L /Low is the left pointer.

R / High is the right pointer.

```
void quicksort(int a[25],int first,int last){
   int i, j, pivot, temp;
   if(first<last){
      pivot=first;
      i=first;
      j=last;
      while(i<j){
         while(a[i]<=a[pivot]&&i<last)
         i++;
         while(a[j]>a[pivot])
         j--;
         if(i<j){
            temp=a[i];
            a[i]=a[j];
```

```
            a[j]=temp;
          }
       }
       temp=a[pivot];
       a[pivot]=a[j];
       a[j]=temp;
       quicksort(a,first,j-1);
       quicksort(a,j+1,last);
    }
}
```

Case:-2

It is also called partition-exchange sort. This algorithm divides the list into three main parts:

1.Elements less than the Pivot element

2.Pivot element(Central element)

3.Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element.

For example: In the array {9,7,5,11,12,2,14,3,10,6}, we take 6 as pivot. So after the first pass, the list will be changed pivot will be at Correct Position

{2,3,5,6,12,7,14,9,10,11}

After selecting an element as pivot, which is the last index of the array in our case, we divide the array for the first time.

In quick sort, we call this partitioning. It is not simple breaking down of array into 2 subarrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the pivot will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.

And the pivot element will be at its final sorted position.

The elements to the left and right, may not be sorted.

Then we pick subarrays, elements on the left of pivot and elements on the right of pivot, and we perform partitioning on them by choosing a pivot in the subarrays.

```
A[]-- List  beg=0 end=n-1
int partition(int a[], int beg, int end)
{
    int left, right, temp, loc, flag;
    loc = left = beg;
    right = end;
    flag = 0;
    while(flag != 1)
    {
        while((a[loc] <= a[right]) && (loc!=right))
        right--;
        if(loc==right)
        flag =1;
        else if(a[loc]>a[right])
        {
```

```
    temp = a[loc];
    a[loc] = a[right];
    a[right] = temp;
    loc = right;
}
if(flag!=1)
{
    while((a[loc] >= a[left]) && (loc!=left))
    left++;
    if(loc==left)
    flag =1;
    else if(a[loc] < a[left])
    {
        temp = a[loc];
        a[loc] = a[left];
```

```
            a[left] = temp;
            loc = left;
        }
    }
    return loc;
}
void quickSort(int a[], int beg, int end)
{
    int loc;
    if(beg<end)
    {
        loc = partition(a, beg, end);
        quickSort(a, beg, loc-1);
        quickSort(a, loc+1, end);
```

```
    }
}
```

Merge sort

Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

Merge Sort Working Process

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Follow the steps below to solve the problem:

MergeSort(arr[], l, r)

If r > l

Find the middle point to divide the array into two halves:

middle m = l + (r – l)/2

Call mergeSort for first half:

Call mergeSort(arr, l, m)

Call mergeSort for second half:

Call mergeSort(arr, m + 1, r)

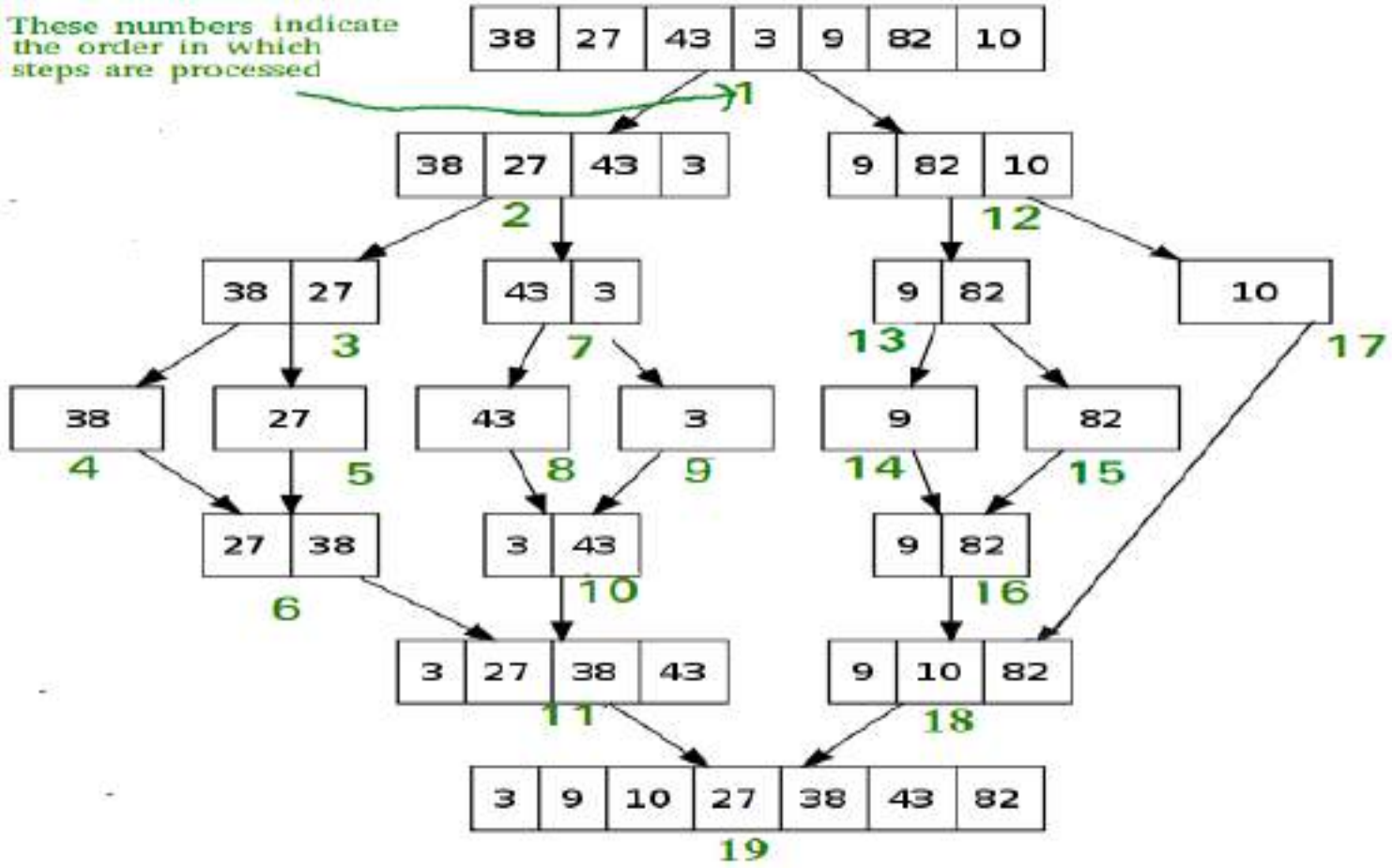
Merge the two halves sorted in steps 2 and 3:

Call merge(arr, l, m, r)

merge sort

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
2
9 82 10
12
38 27
3
43 3
7
9 82
13
10
17
38
4
27
5
43
8
3
9
9
14
82
15
27 38
6
3 43
10
9 82
16
3 27 38 43
11
9 10 82
18
3 9 10 27 38 43 82
19

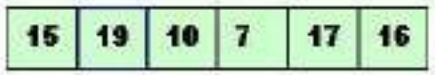
Heap Sort

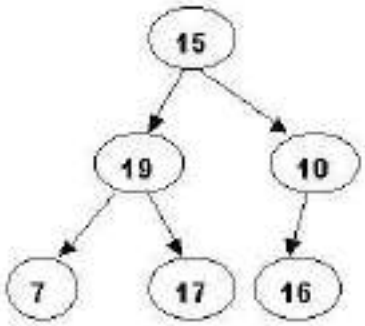
Heap:

A heap is a specialized tree-based data structure that satisfies the heap property:

If P is a parent node of C, then the key (the value) of P is either greater than or equal to (max heap) or less than or equal to (min heap) the key of C. The node at the "top" of the heap (with no parents) is called the root node.

Heap Sort

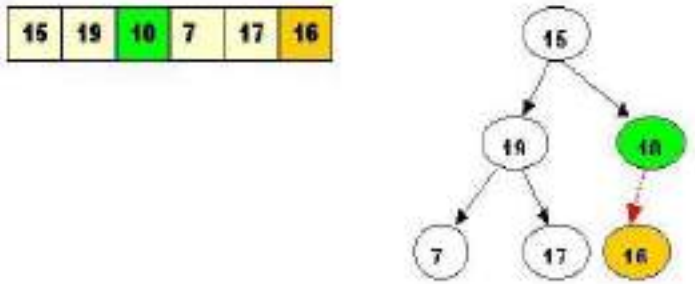




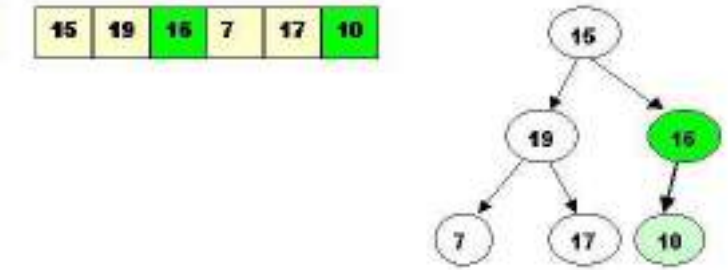
The array represented as a tree, complete but not ordered

Start with the rightmost node at height 1 - the node at position 3 = Size/2.

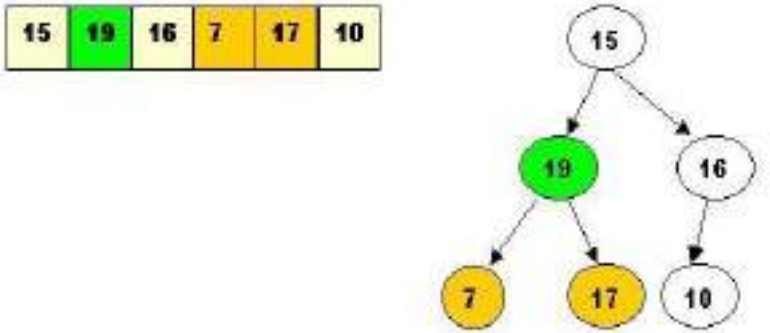
It has one greater child and has to be percolated down:



After processing array[3] the situation is:

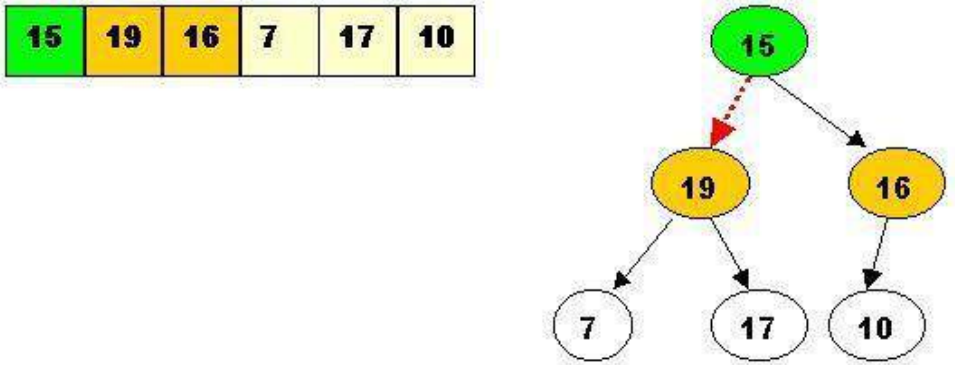


Next comes array[2]. Its children are smaller, so no percolation is needed.

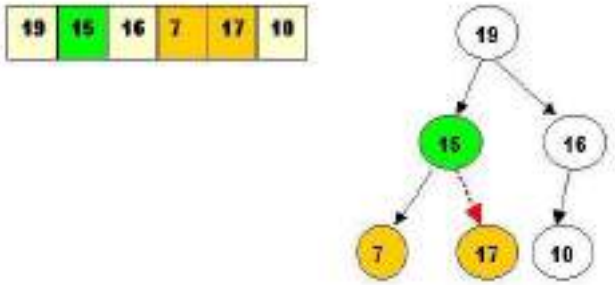


The last node to be processed is array[1]. Its left child is the greater of the children.

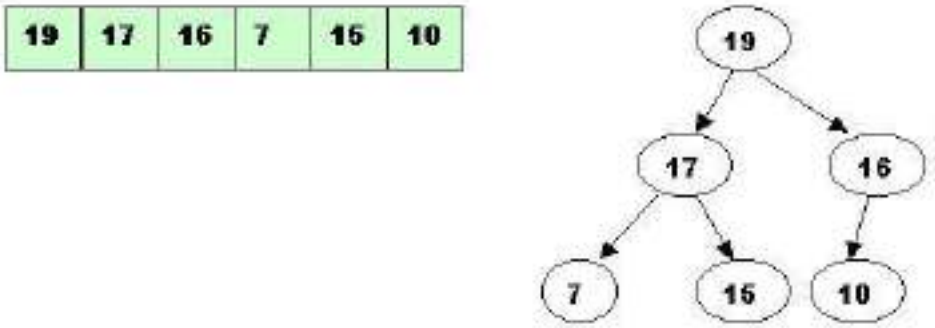
The item at array[1] has to be percolated down to the left, swapped with array[2].



As a result the situation is:



The children of array[2] are greater, and item 15 has to be moved down further, swapped with array[5].

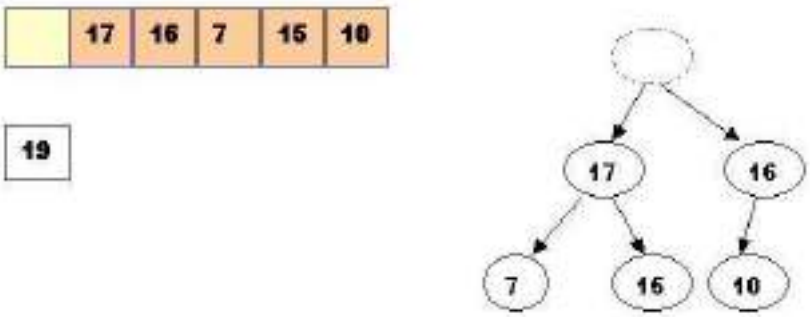


Now the tree is ordered, and the binary heap is built.

Sorting - performing deleteMax operations:

Delete the top element 19.

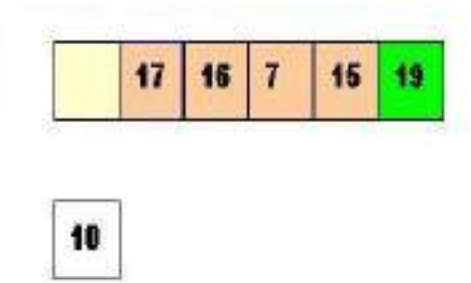
Store 19 in a temporary place. A hole is created at the top



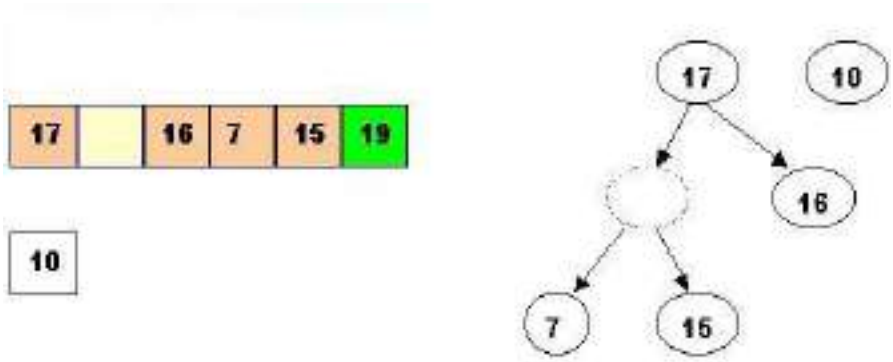
Swap 19 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.

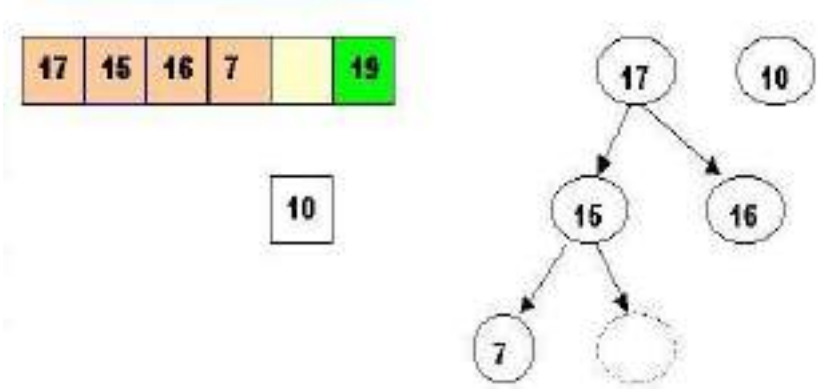
Instead it becomes a cell from the sorted array



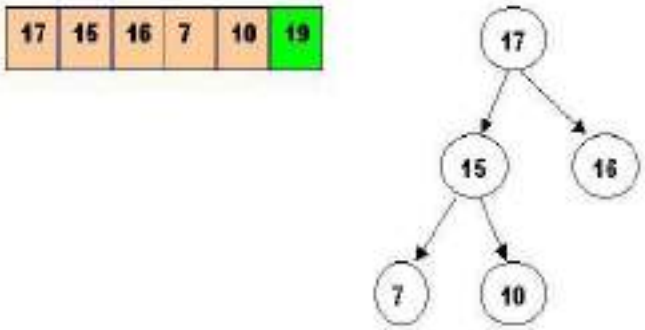
Percolate down the hole



Percolate once more (10 is less that 15, so it cannot be inserted in the previous hole)

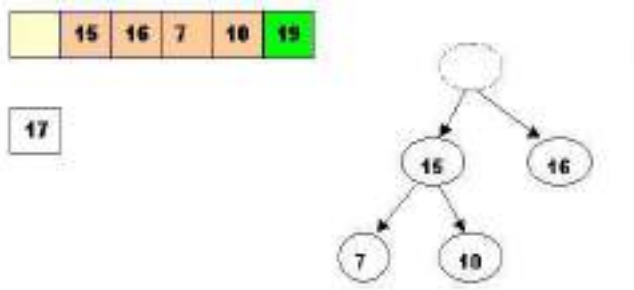


Now 10 can be inserted in the hole



DeleteMax the top element 17

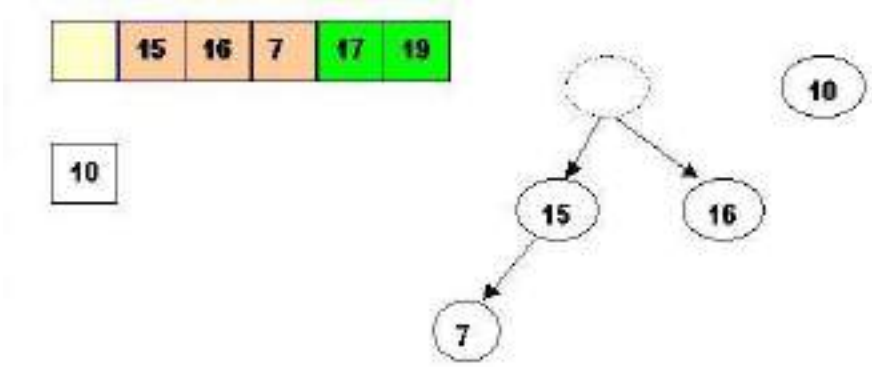
Store 17 in a temporary place. A hole is created at the top



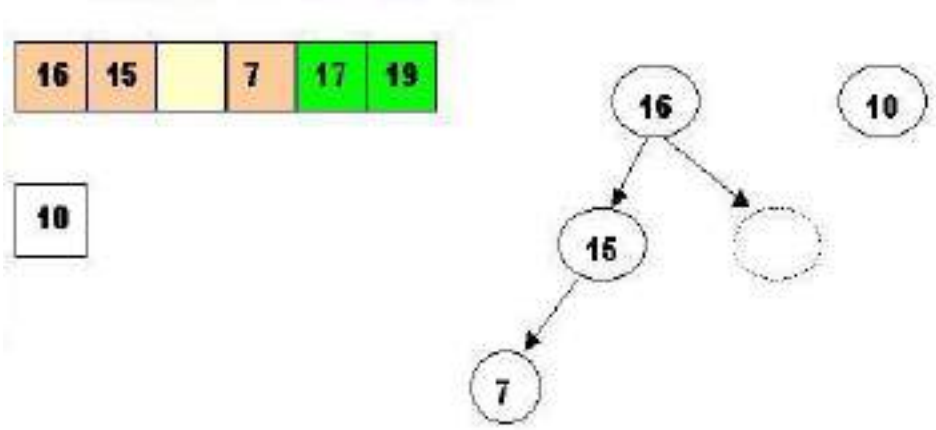
Swap 17 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.

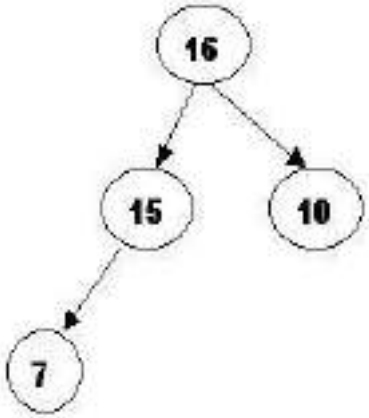
Instead it becomes a cell from the sorted array



The element 10 is less than the children of the hole, and we percolate the hole down:

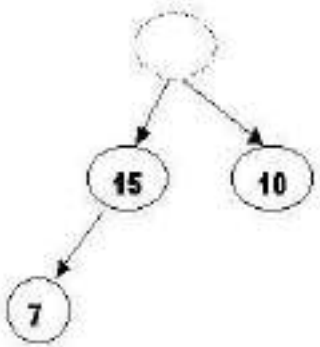


Insert 10 in the hole



DeleteMax 16

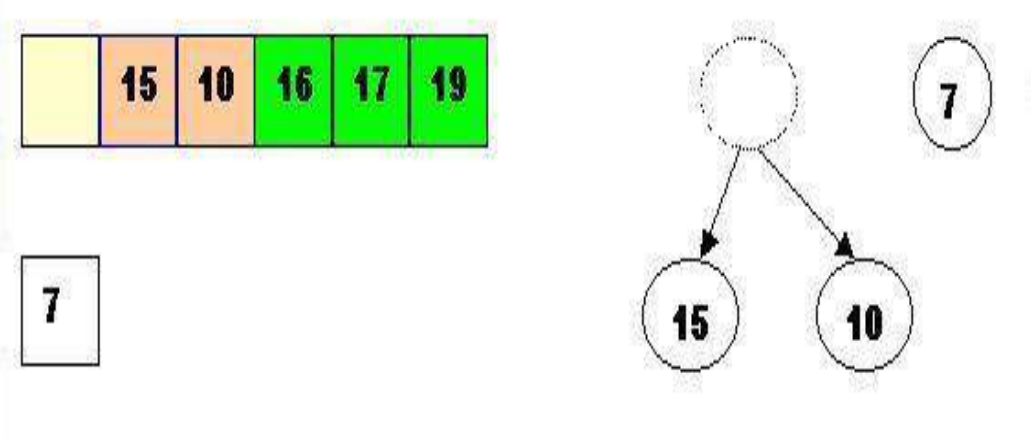
Store 16 in a temporary place. A hole is created at the top



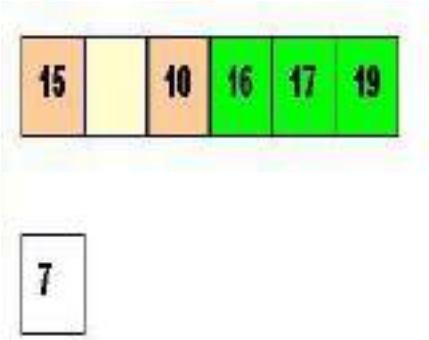
Swap 16 with the last element of the heap.

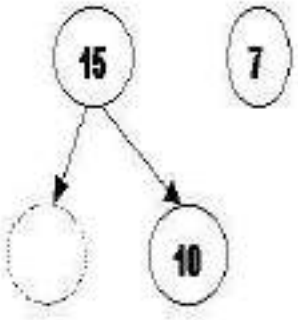
As 7 will be adjusted in the heap, its cell will no longer be a part of the heap.

Instead it becomes a cell from the sorted array

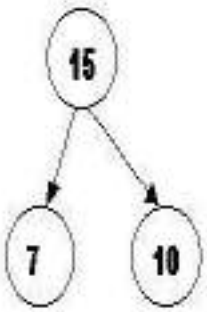


Percolate the hole down (7 cannot be inserted there it is less than the children of the hole)



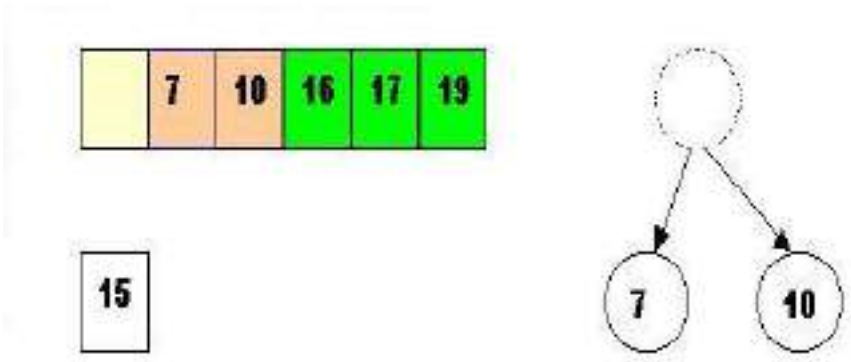


Insert 7 in the hole



DeleteMax the top element 15

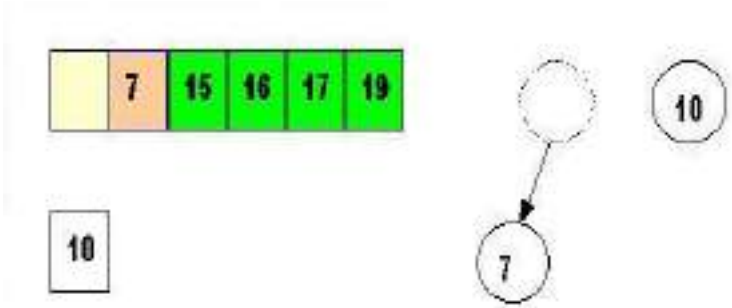
Store 15 in a temporary location. A hole is created.



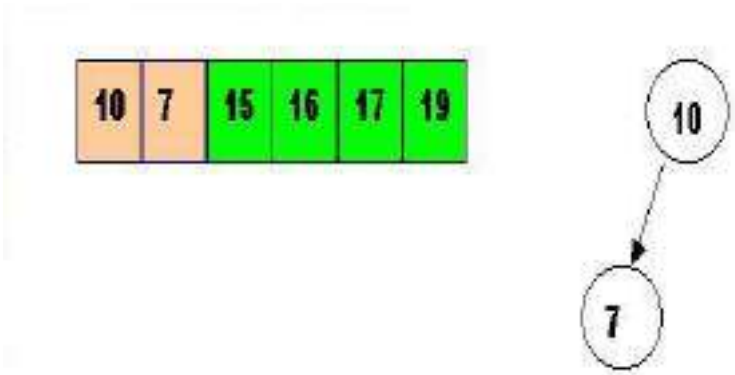
Swap 15 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.

Instead it becomes a position from the sorted array

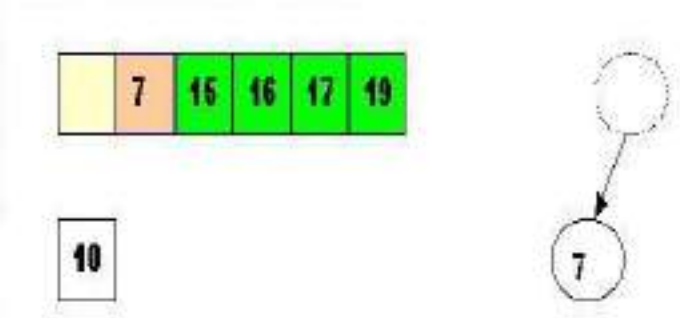


Store 10 in the hole (10 is greater than the children of the hole)



DeleteMax the top element 10.

Remove 10 from the heap and store it into a temporary location.



Swap 10 with the last element of the heap.

As 7 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array

Store 7 in the hole (as the only remaining element in the heap

7 is the last element from the heap, so now the array is sorted

Heap Sort Program

```
#include <iostream> using namespace std; const int MAX = 10 ;
class array
{ private : int arr[MAX] ; int count ;
public :
array( ) ;
 void add ( int num ) ;
void makeheap(int ) ;
void heapsort( ) ;
void display( ) ;
} ;
array :: array( )
{ count = 0 ;
for ( int i = 0 ; i < MAX ; i++ ) arr[MAX] = 0 ;
}
void array :: add ( int num )
{ if ( count < MAX )
{
arr[count] = num ;
count++ ;
}
else cout << "\nArray is full" << endl ;
```

```
}
void array :: makeheap(int c)
{
for ( int i = 1 ; i < c ; i++ )
{ int val = arr[i] ; int s = i ; int f = ( s - 1 ) / 2 ;
while ( s > 0 && arr[f] < val )
{ arr[s] = arr[f] ; s = f ; f = ( s - 1 ) / 2 ;
} arr[s] = val ;
}
}
void array :: heapsort( )
{ for ( int i = count - 1 ; i > 0 ; i-- )
{ int ivalue = arr[i] ; arr[i] = arr[0] ; arr[0]=ivalue; makeheap(i);
}
}
void array :: display( )
{ for ( int i = 0 ; i < count ; i++ ) cout << arr[i] << "\t" ;
cout << endl ;
}
int main( )
{ array a ;
                a.add ( 11 ) ;          a.add ( 2 ) ;
                a.add ( 9 ) ;           a.add ( 13 ) ;
```

```
                a.add ( 57 ) ;          a.add ( 25 ) ;
                a.add ( 17 ) ;          a.add ( 1 ) ;
                a.add ( 90 ) ;          a.add ( 3  ) ;
a.makeheap(10) ;
cout << "\nHeap Sort.\n" ;
cout << "\nBefore Sorting:\n"  ;
a.display( ) ;
a.heapsort( ) ;
cout << "\nAfter Sorting:\n" ; a.display( ) ;
}
```

Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	n * log n	n * log n
Quicksort	n^2	n * log n
Radix sort	n	n
Treesort	n^2	n * log n
Heapsort	n * log n	n * log n