DATA STRUCTURES USING C - WEB OLC CONTENT

CURSOR-BASED LINKED LISTS

Pointers are extensively used while implementing different linked list variants such as singly-linked list, doubly-linked list and circular list. But, what happens when pointers are not supported by the programming language chosen for linked list implementation? For instance, LISP and Python programming languages do not support pointers. In such situations, a cursor-based linked list is used. It stores the list elements in an array and uses the array index values for connecting the list elements.

The following points explain how a cursor-based linked list emulates a pointer-based linked list:

1. The node of a cursor-based list comprises of two fields: INFO and NEXT. INFO stores the node value while NEXT stores the index value of the next element in the list.
2. A separate list of available or free nodes is maintained in the same manner to provide a new node at the time of insert operation.

Figure O.1 shows an illustration of cursor-based linked list:

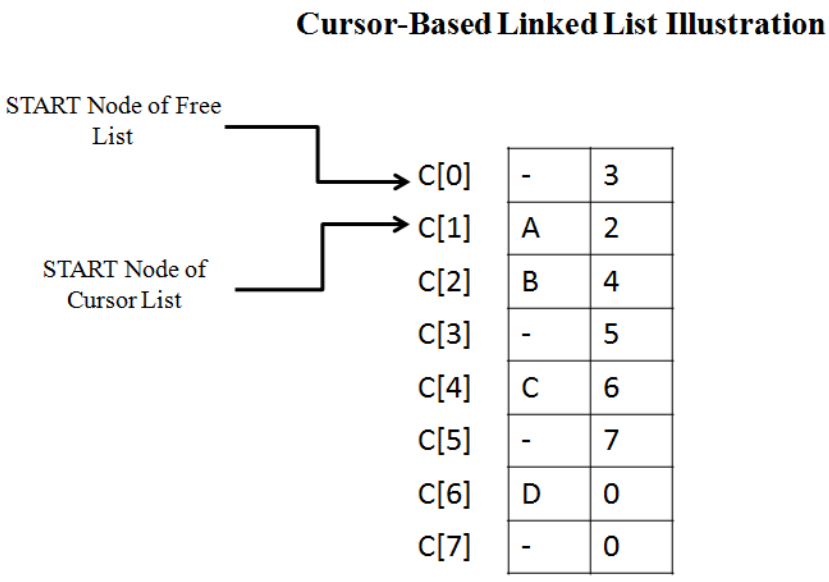


Figure O.1. **Cursor-based list.**

In the above illustration:

- Cursor-based linked list is implemented using array C[8]
- The elements in the list are A-B-C-D
- The list of free nodes available for insert operation are C[0]-C[3] -C[5] -C[7]
- The last elements of both cursor and free list point at index value 0, which is also the start of the free list. This is analogous to a pointer-based linked list where the last node contains the NULL pointer.

Let us now take a look at the implementation details of cursor-based linked list.

Declaration of List Node

The following structure declaration defines a cursor-based linked list node:

```
struct node
{
 int INFO;
 int NEXT;
};
struct node CL[8];
```

Insert Operation

The following code implements the insert operation for cursor-based linked list:

```
void insert(int value, struct node C, int loc)
{
 int k;
 k=aloc(); /*aloc() function returns the location of the next available node in the
free list*/
 C[k].INFO=value;
 C[k].NEXT=C[loc].NEXT;
 C[loc].NEXT=k;
}
```

Delete Operation

The following code implements the delete operation for cursor-based linked list:

```
void delete(int value, struct node C)
{
 int k1, k2;
 k1=search_prev(value,C); /*search_prev() returns the location of node previous to
the node containing value*/


 if(k1!=-1)
 {
  k2=C[k1].NEXT;
  C[k1].NEXT=C[k2].NEXT;
  release(k2);
 }
}
```

LEFT CHILD/RIGHT SIBLING DATA STRUCTURES FOR GENERAL TREES

Left child/right sibling data structure is used to covert a general tree into a binary tree. Unlike binary trees which have a restriction of a maximum of two child nodes, the general trees do not have any such restriction. The nodes of a general tree may contain more than two child nodes. Consider the general tree shown in Figure O.2:

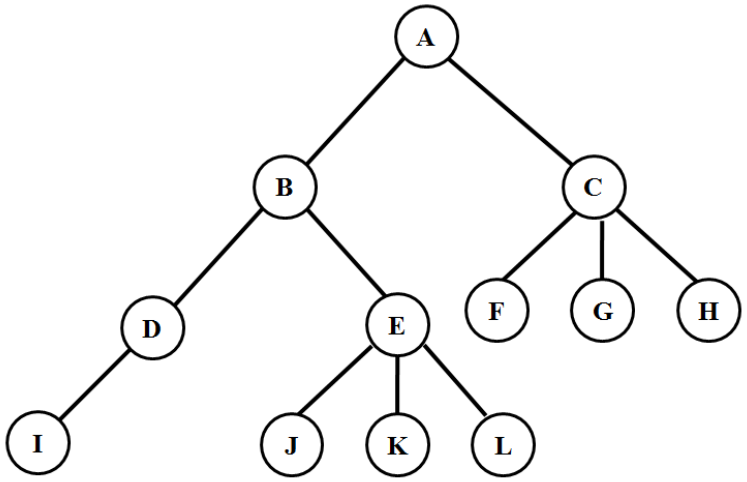


Figure O.2. **General Tree**

To convert this general tree into a binary tree, you need to perform the following steps:

1. Make the root node of general tree as the root node of binary tree.
2. Identify the leftmost child node of the current node in general tree and attach it as the left child of the current node in the binary tree.
3. Identify the right sibling node and attach it as the right child in the binary tree.
4. Repeat the above steps for all the child nodes of the general tree to derive the equivalent binary tree.

Figure O.3 shows how the general tree is transformed into an equivalent binary tree:

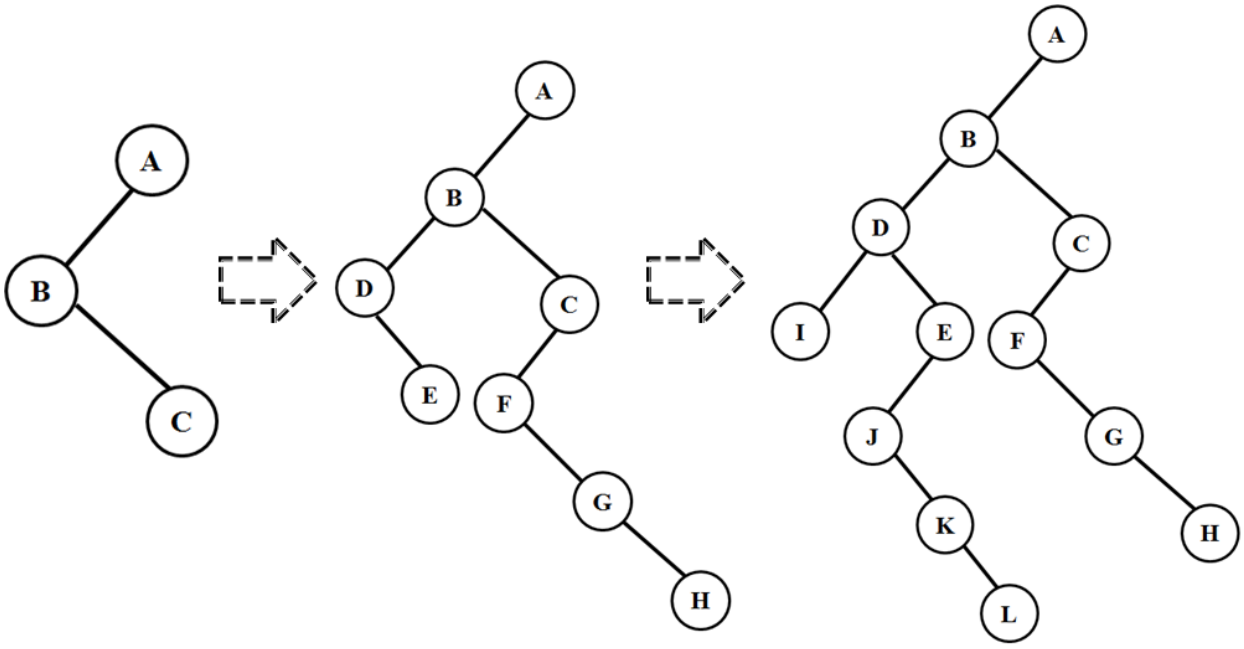


Figure O.3. **Converting general tree into binary tree**

HEAPS

Heap is a tree-based data structure that possesses the heap property. As per the heap property, the key of any node N has a definite order (such as less than or greater than) with respect to the key of its parent node. If the keys of all the parent nodes are greater than or equal to the keys of the child nodes then the heap is referred as max heap. Similarly, if the keys of all the parent nodes are less than or equal to the keys of the child nodes then the heap is referred as min heap.

Figure O.4 shows examples of min and max heaps:

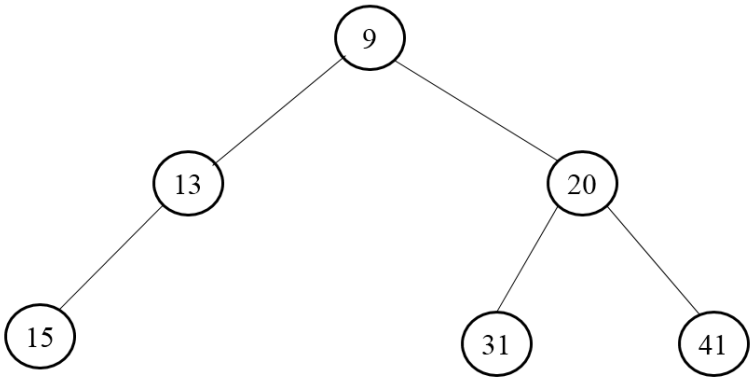


Figure O.4 (a). **Min Heap**

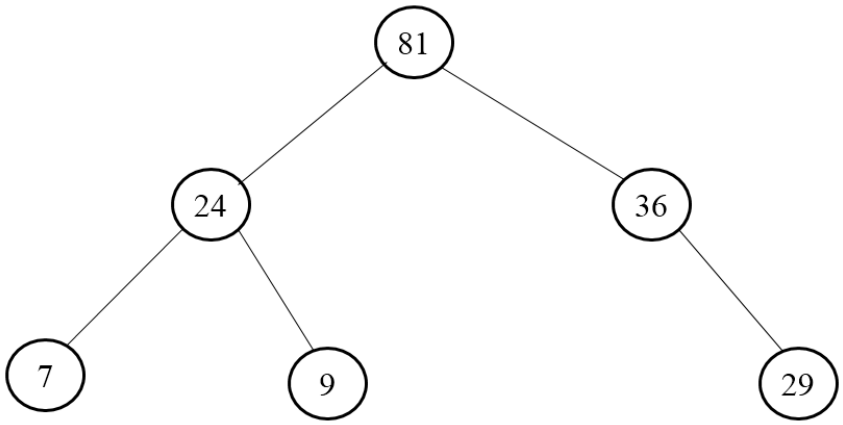


Figure O.4 (b). **Max Heap**

BINARY HEAPS

A binary heap is a heap that in addition to the heap property also satisfies the shape property. As per the shape property, the heap must be a complete binary tree.

Thus, a **binary heap**:

- is a max heap, min heap or both (**heap property**).
- is a complete binary tree. In a complete binary tree of depth d, all the levels from 0 to d-1 are completely filled and all the leaf nodes present at level d are placed towards the left side (**structure property**).

Figure O.5 shows examples of binary min and max heaps:

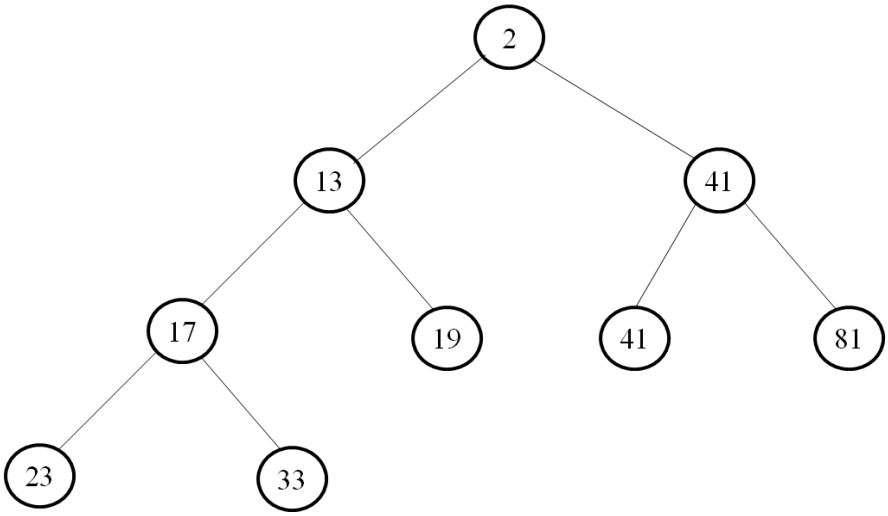


Figure O.5 (a). **Binary Min Heap**

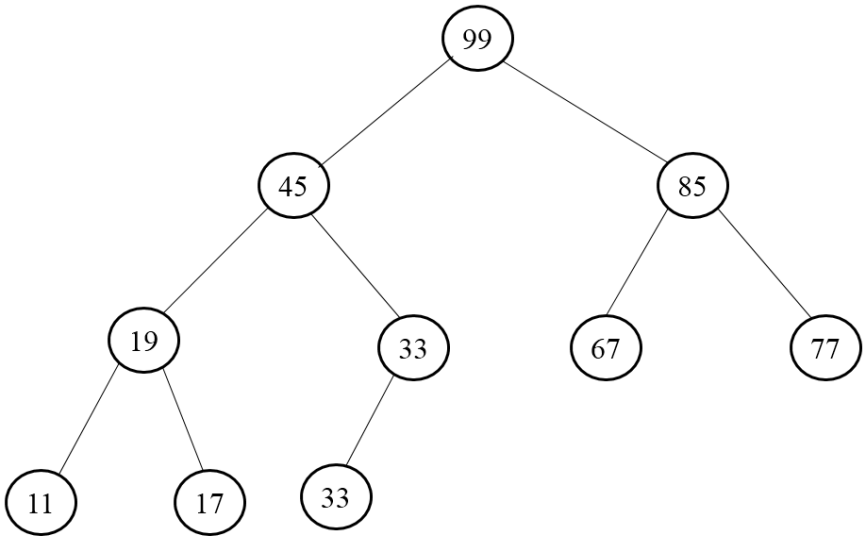


Figure O.5 (b). **Binary Max Heap**

Array Representation of Binary Heap

A simple way of representing a binary heap in memory is through array. Since, a binary heap is a complete binary tree; its elements can be stored in an array without any gaps. We will consider an array representation where heap elements are stored starting at index location 1. Figure O.6 shows the array representation of the min heap shown in Figure O.5 (a).

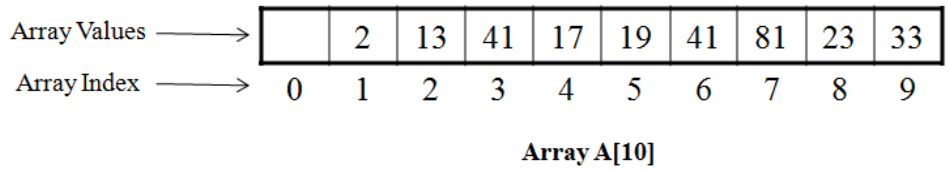


Figure O.6. **Array representation of binary heap**

From the above array representation, we can deduce the following:

- The root element is stored at location A[1].
- The parent of any node A[i] (except root node) is A[i/2].
- The left child of any node A[i] is A[2i].
- The right child of any node A[i] is A[2i+1].

Binary Heap Operations

Insert and Delete are the two main operations performed on a binary heap. Both the operations are performed in such a manner that the heap and structure properties of the binary heap are restored.

Insert

The insert operation is performed by adding the new element at the next available position in the heap and then moving the element in the upward direction until the heap property is restored. The act of moving the element in the upward direction is referred as *percolate-up*.

To insert an element into the binary heap, perform the following steps:

1. Add the element at the next available position in the heap.
2. Compare the element with its parent and swap if it is not in correct order.
3. Repeat the above step until the elements are in order.

Figure O.7 illustrates the addition of element 7 in the binary min heap shown in Figure O.5 (a).

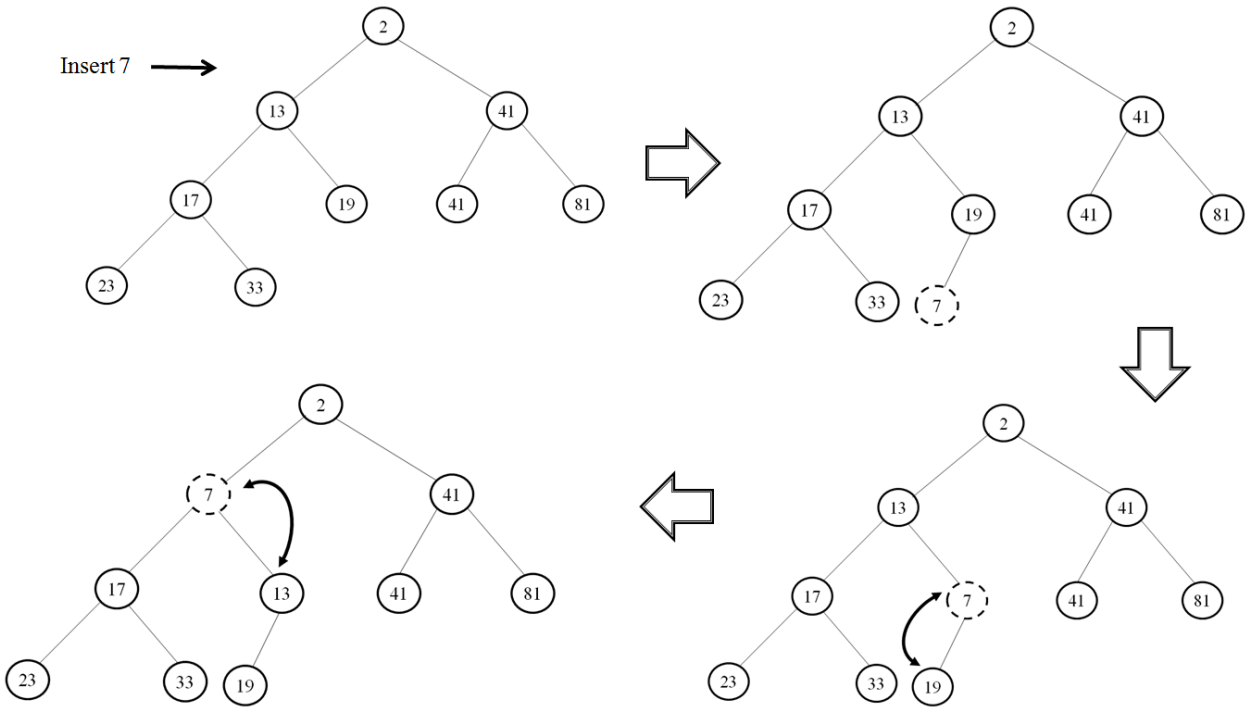


Figure O.7. **Inserting element in binary min heap**

The following code implements the insert operation for a binary min heap:

```
void insert(int value, int *H)
{
  int loc = ++heap_count;

  while(loc>1 && H[loc/2]>value)
  {
   H[loc]=H[loc/2];
      loc=loc/2 ;
   }
   H[loc]=value;
}
```

Delete

The delete operation involves deleting the root element and replacing it with the last element present at the lowest level of the binary heap. The heap property is then restored by moving in the downward direction and performing the necessary swap operations until the heap property is restored. The act of moving in the downward direction is referred as *percolate-down*.

To delete an element from the binary heap, perform the following steps:

1. Remove the root element and replace it with the last element present at the lowest level of the heap.
2. Compare the new element at the root position with its children and swap if it is not in correct order.
3. Repeat the above step until the elements are in order.

Figure O.8 illustrates how the delete operation is performed on the binary max heap shown in Figure O.5 (b).

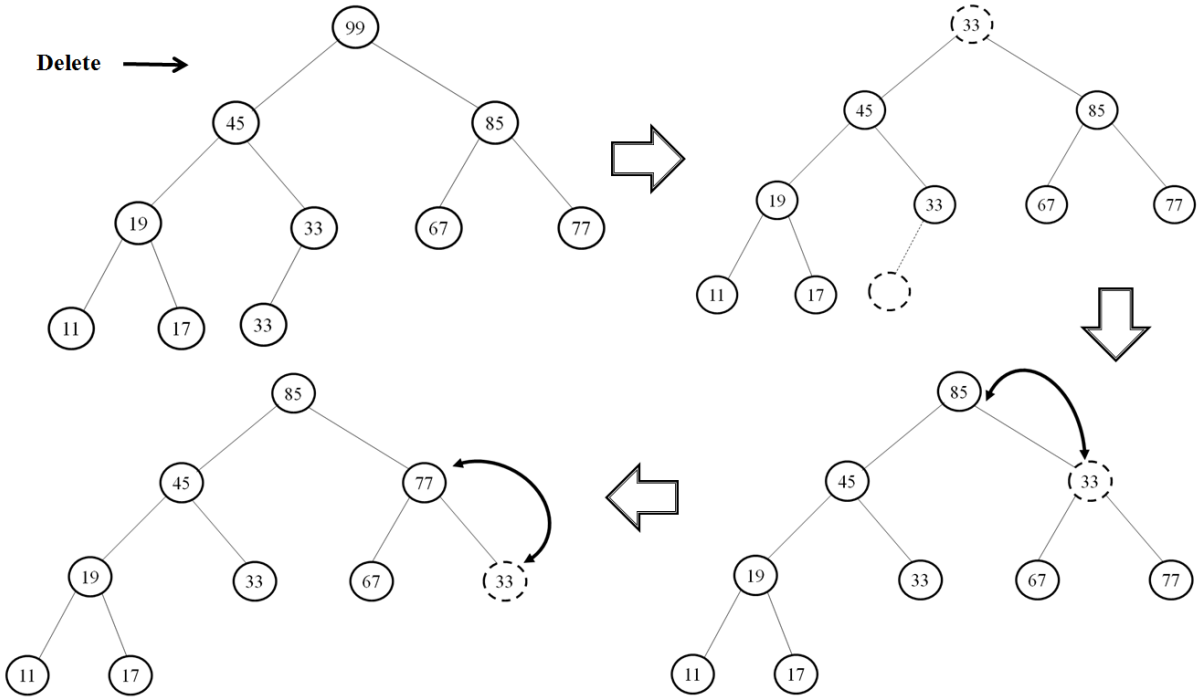


Figure O.8. **Performing delete operation in binary max heap**

The following code implements the delete operation for a binary max heap:

```
void delete(int *H)
{
  int loc = 1;
  int k;
  H[loc]=H[heap_count--];

  while(loc<heap_count)
  {
   k=find_max(loc, 2*loc, 2*loc+1)
```

```
    if(k==-1)
     break;
    else
    {
     swap(H[loc],H[k]);
     loc=k;
    }
   }
}
```

APPLICATIONS OF BINARY HEAPS

The two most important applications of binary heaps are:

- Heap sort
- Priority Queue

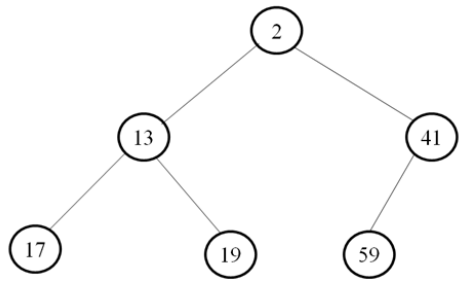
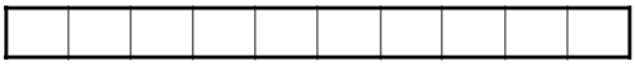
Heap Sort

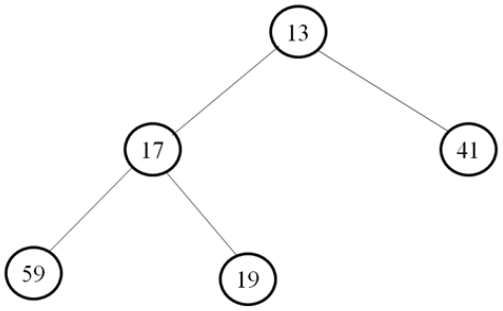
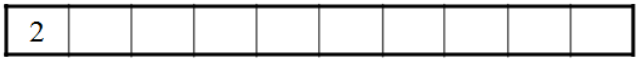
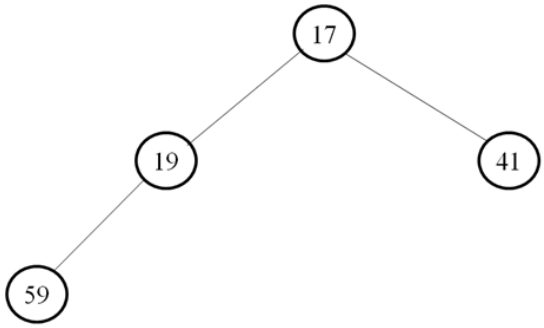
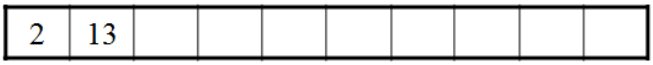
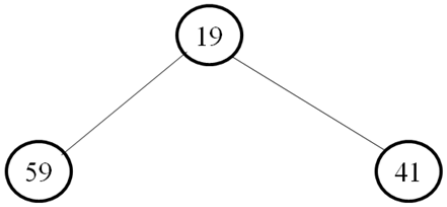
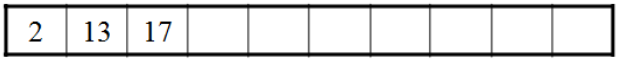
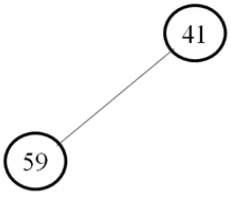
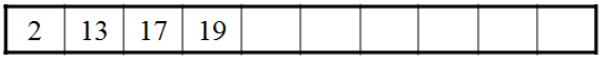
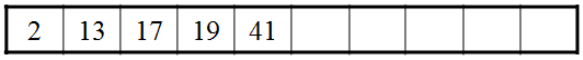
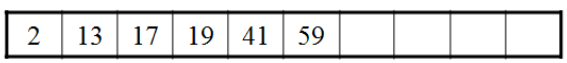
Heap sort technique is used to sort a given list of elements using a binary heap. As part of the heap sort technique, the following tasks are performed:

1. A min or max heap is built from the given list of elements.
2. Delete operation is repeatedly performed on the heap and the retrieved elements are stored in successive locations of an array. Depending on whether the binary heap was a min heap or a max heap, the array forms the sorted output in ascending or descending order.

Table O.1 illustrates how heap sort obtains the sorted output in ascending order by repeatedly performing the delete operation on a binary min heap:

Table O.1. **Illustration of heap sort**

 **Min Heap**	 **Array**

 **Min Heap**	 **Array**
 **Min Heap**	 **Array**
 **Min Heap**	 **Array**
 **Min Heap**	 **Array**
 **Min Heap**	 **Array**
Empty Min Heap	 **Sorted Array**

The algorithm for performing heap sort on an array A containing n elements is given below:

```
heap_sort(A, n)

Step 1: Start

Step 2: Call build_heap(A, H, n) //call the build_heap function to build a min heap
H from given n elements

Step 3: Set i = 1

Step 4: Repeast Step 5-7 while i <= n

Step 5: Call delete(H,element) //call the delete function to retrieve the root
element from the min heap

Step 6: Set A[i]=element

Step 7: Set i = i + 1

Step 8: Stop
```

Priority Queue

A binary heap can also be used for implementing a priority queue. In such cases, the heap order property is maintained as per the priorities of the queue elements. A delete operation removes the highest priority element from the queue and readjusts the heap to restore the order of the queue.

The enqueue and dequeue operations are performed in the same manner as insert and delete operations of binary heaps.

HASHING

Collision is one of the elementary problems associated with hashing. Several collision resolution techniques are used to overcome this problem. These techniques are:

- Separate chaining
- Open addressing
- Rehashing
- Extendible Hashing

Separate Chaining

The separate chaining technique uses linked lists to overcome the problem of collisions. In this technique, all the keys that resolve to the same hash values are maintained in a linked list. This is depicted in Figure O.9:

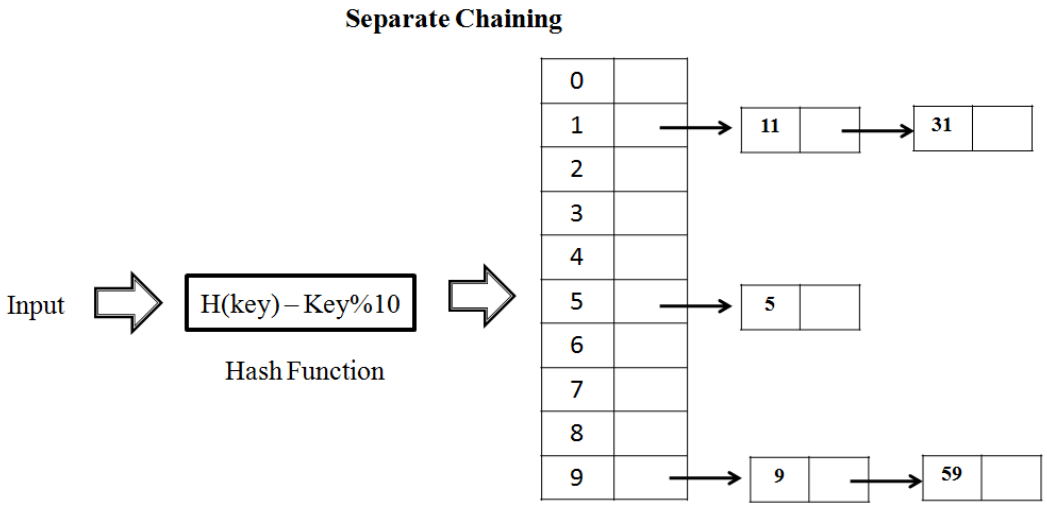


Figure O.9. **Separate chaining**

As shown in the above illustration, whenever a collision occurs, the key value is added at the end of the corresponding linked list. Thus, each position in the hash table acts as a header node for a linked list. To perform the search operation, the list indicated by the hash function is searched sequentially.

The following code snippet shows the type declarations and insert function for separate chaining hashing:

```
.
.
struct node
{
 int element;
 struct node *NEXT;
};
.
.
struct hashT
{
 int size;
 struct node *list_ptr;
};
.
.
void insert(int key, hashT H)
{
 int pos;
 pos=hash(key,H);
```

```
 if(isempty(pos))
  create_list(pos,key);
 else
 insert_list(pos, key);
}
.
.
```

Open Addressing

The disadvantage of using separate chaining hashing is that it adds the overhead of maintaining multiple linked lists. Also, the use of pointers slows the program. To overcome these disadvantages, open addressing hashing is used. In this technique, a block of memory space is reserved for storing the hash keys. In case of a collision, alternative locations are accessed until an empty location is found. For example, H1(x), H2(x), H3(x), etc. are computed one after the other until a vacant slot is found. A generalised representation of this hash function is given below:

```
hi(x) = (hash(x) + F(i)) % n
```

The way F(i) is defined determines how alternative locations are searched for the hash keys.

Linear Probing

In case of linear probing, F(i) is a linear function of i. That means, alternative locations are searched in a sequential manner. Thus,

```
F(i) = i
Or,
hi(x) = (hash(x) + i) % n
```

The various iterations or attempts towards computing the next available location are given below:

```
h0(k) = hash(k) % n
h1(k) = hash(k) + 1) % n
h2(k) = hash(k) + 2) % n
```

Figure O.10 shows an illustration of the linear probing technique:

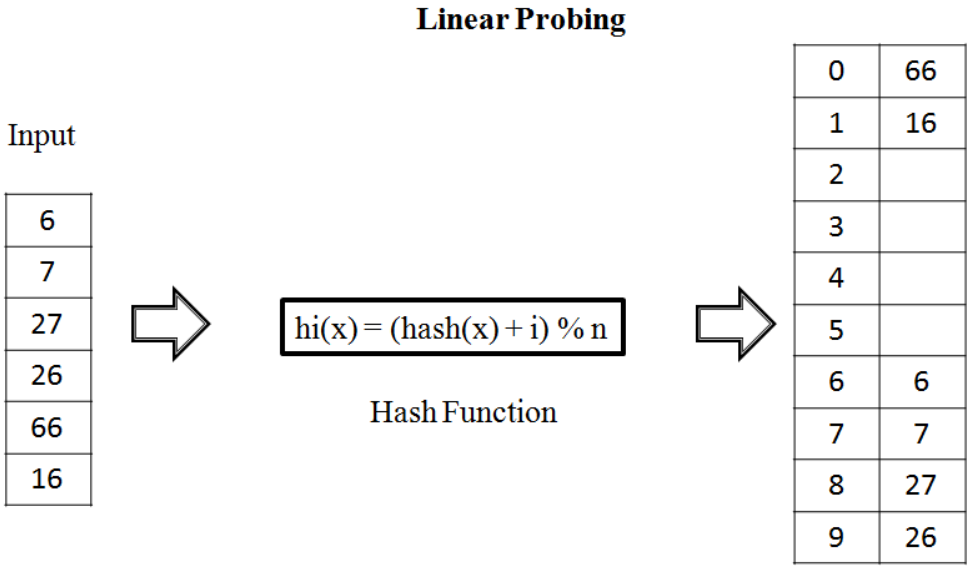


Figure O.10. **Linear Probing**

Quadratic Probing

One of the disadvantages of linear probing is that it leads to clustering of key values at one place. Thus, if a new key value hashes to the cluster then it requires several iterations to find the next vacant slot. To overcome this disadvantage, quadratic probing technique is used. In this technique, F(i) is a quadratic function of i. That means, alternative locations are searched as a quadratic function. Thus,

$F(i) = i^2$

Or,

$hi(x) = (hash(x) + i^2) \% n$

The various iterations or attempts towards computing the next available location are given below:

```
h0(k) = hash(k) % n
h1(k) = hash(k) + 1) % n
h2(k) = hash(k) + 4) % n
```

Figure O.11 shows an illustration of the quadratic probing technique:

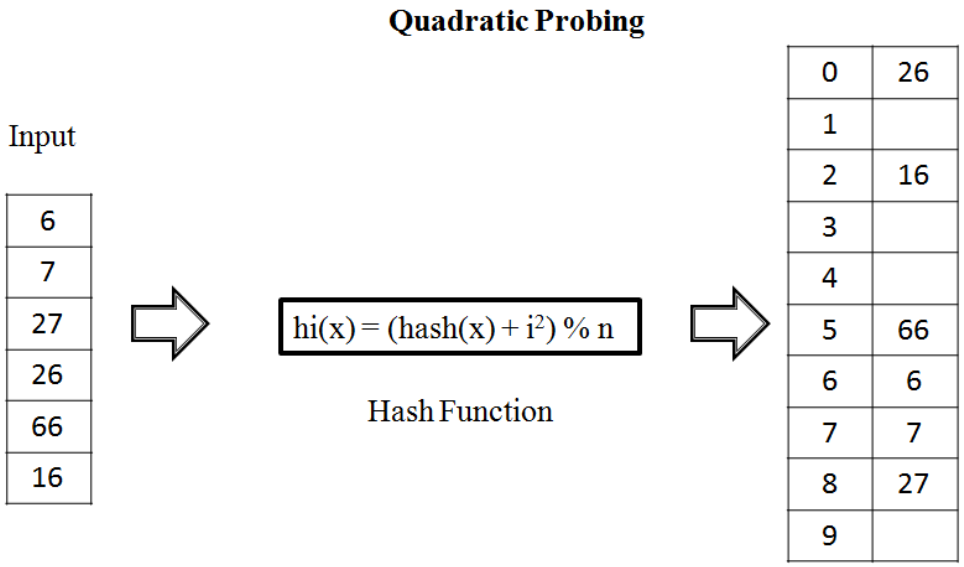


Figure O.11. **Quadratic Probing**

Double Hashing

The use of quadratic probing may as well lead to the formation of secondary clusters. To overcome such situation, double hashing technique is used. It uses two hash functions to arrive at a hash value. Thus,

```
F(i) = i * hash2(x)
Or,
hi(x) = (hash(x) + i * hash2(x)) % n
```

The various iterations or attempts towards computing the next available locations are given below:

```
h0(k) = hash(k) % n
h1(k) = hash(k) + 1 * hash2(k)) % n
h2(k) = hash(k) + 2 * hash2(k)) % n
```

Figure O.12 shows an illustration of the double hashing technique:

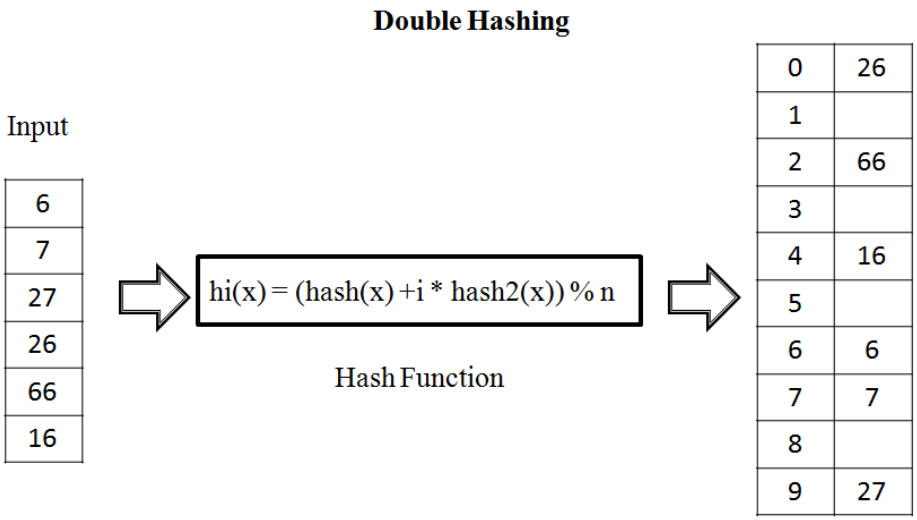


Figure O.12. **Double Hashing**

Rehashing

When the hash table becomes too full in open addressing hashing, the successive insert operations may consume more time to complete. To overcome this situation, the rehashing technique is used. In the rehashing technique, a bigger memory cluster, approximately twice the size, is reserved as soon as the original hash table becomes too full. Then, the elements are rehashed into the new cluster. Thus,

$h_1(x)$ rehashes to $h_2(x)$

If $h_1(x) = x \bmod n$, where n is a prime number

Then, $h_2(x) = x \bmod m$, where m is a prime number approximately twice as large as n

The trigger for initiating rehashing could be any of the following:

- When the insert operation fails
- When the cluster becomes half full
- When the hash performance goes below a pre-defined threshold value

Figure O.13 shows an illustration of the rehashing technique:

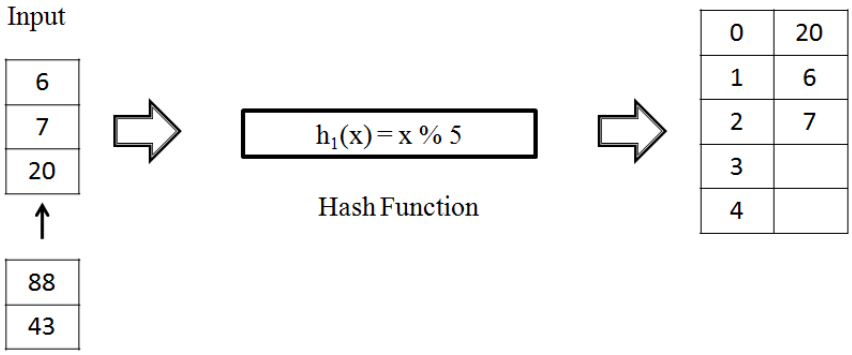


Figure O.13 (a). **Rehashing $h_1(x)$**

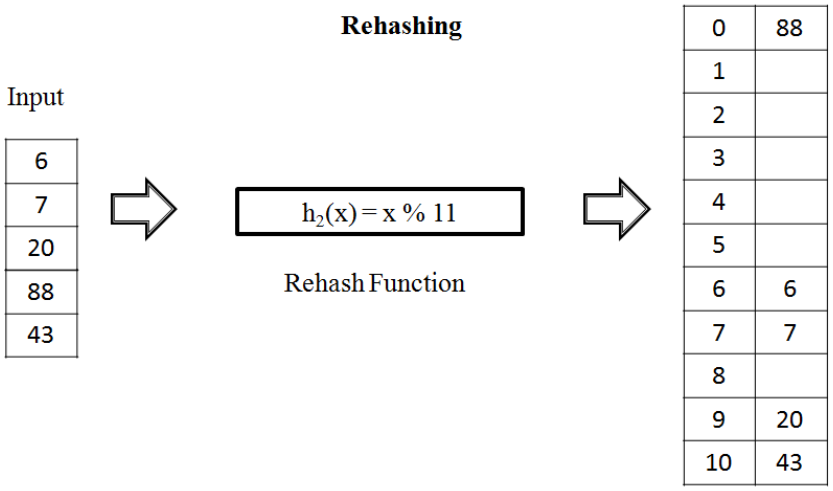


Figure O.13 (b). **Rehashing $h_2(x)$**

Extendible Hashing

In the hashing techniques that we have learned so far, we observed that the static hashing techniques (separate chaining and open addressing) led to performance issues at some point in time. Similarly, dynamic hashing technique (rehashing) led to temporary unavailability of hash table when the rehashing is being done. To avoid these limitations, extendible hashing technique is used.

In extendible hashing, a hash table is stored in the main memory and buckets are stored in the disk. Each value in the hash table is a pointer to a bucket in the secondary memory. The hash function is applied on the input value to generate the bucket pointer and perform the store and retrieve operations.

To understand extendible hashing, consider the illustration shown in Figure O.14.

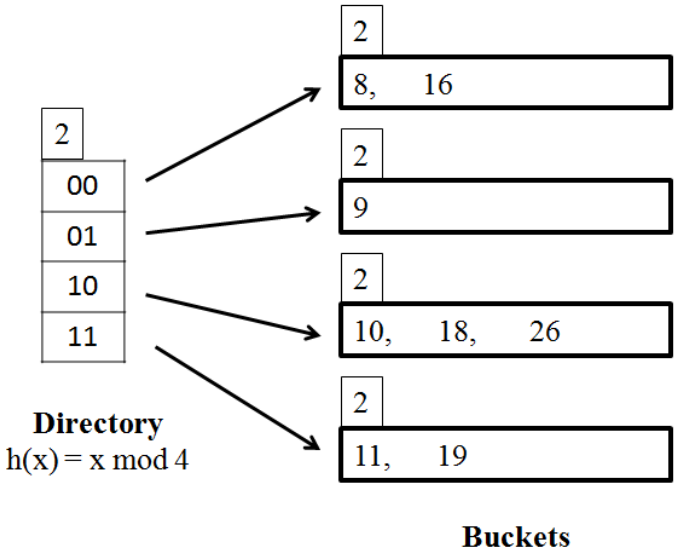


Figure O.14. **Extendible Hashing**

As shown in the above figure,

- A directory of four slots has been used to depict extendible hashing.
- Each of the four directory slots points at the corresponding buckets.
- Each bucket stores a maximum of four integers.
- The hash function that determines the bucket pointer is *h(x) = x mod 4*. That means, it performs the modulus of 4 on all the input values to determine the target bucket. Or, we can say that it considers the two right most binary digits of the input number to determine the bucket in which the number will be stored.
- The value written at the top of the directory is called global depth. It indicates the number of rightmost bits of the input number that are used to determine the target bucket. Similarly, the value written at the top of the buckets is called local depth.

Now, let us insert the values 6 and 30 into the directory. The result is depicted in Figure O.15:

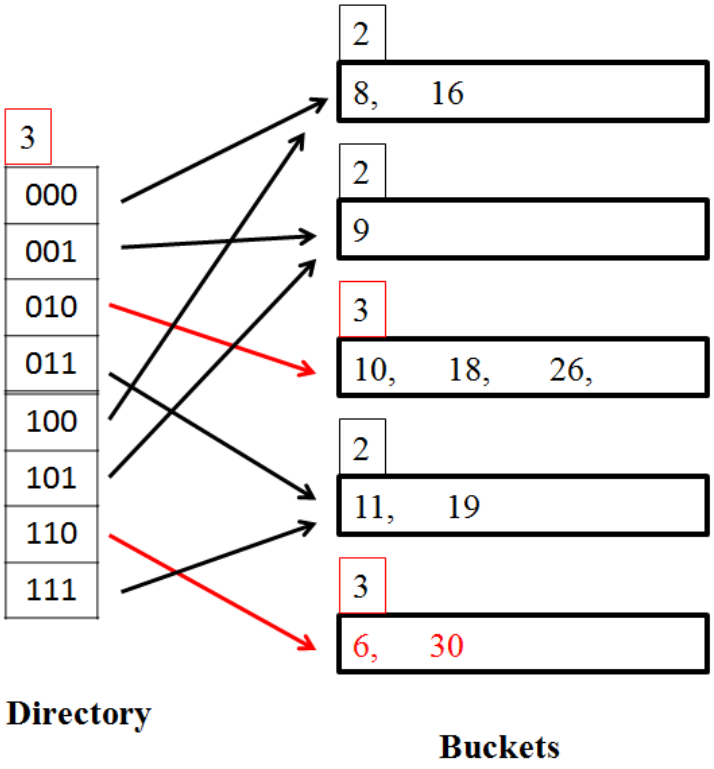


Figure O.15. **Extending the Directory**

The values 6 and 30 would have been inserted in bucket '10' but since there is a restriction of four elements for each bucket the directory is extended to double its size. A new bucket is also added at the bottom and the elements from the overflowing bucket are redistributed by using the hash function *h(x) = x mod 8*. Notice the change in the value of global depth and the local depths of impacted buckets.

So, essentially when is a directory extended? Here are the rules for the same:

1. If the insertion of the new element is causing overflow situation in a bucket then compare its local depth with the global depth.

2. If the local depth is less than the global depth then add a new bucket, adjust the pointers and redistribute the elements.
3. If the local depth is equal to the global depth then expand the directory as depicted in Figure O.15.

Extendible hashing technique is about expanding and rehashing, so how is it different from the Rehashing technique? The difference lies in the manner in which rehashing is done. In case of rehashing technique, the entire hash table is rehashed into a new table in one go. However, in case of extendible hashing, splitting and rehashing is performed only on a specific portion of the memory cluster thus reducing the impact on the performance and availability of the overall hashing scheme.

GRAPHS

Biconnectivity

Biconnectivity is established when an undirected, connected graph stays connected after the removal of any one of its vertices. Such a graph is termed as biconnected graph. Biconnectivity implies that there is always an alternate graph traversal route available after the failure of a node.

Figure O.16 shows a biconnected graph:

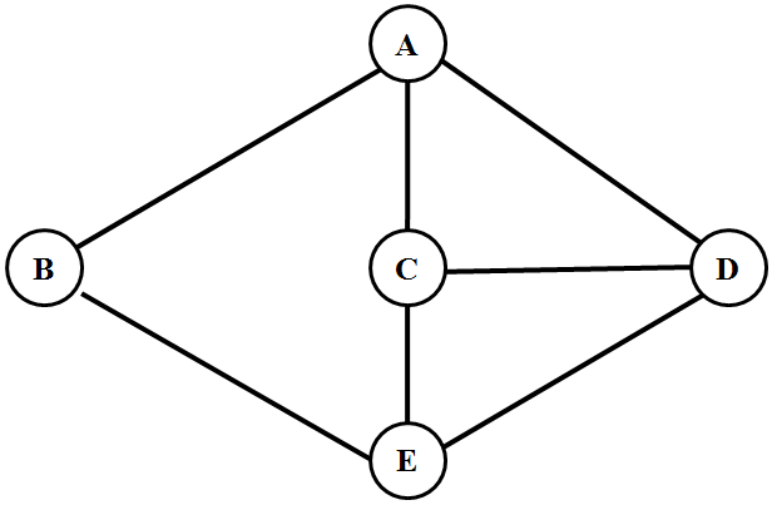


Figure O.16. **Biconnected Graph**

As shown in the above figure, the graph stays connected even after the removal of any one of its nodes.

Now, consider the connected graph shown in Figure O.17:

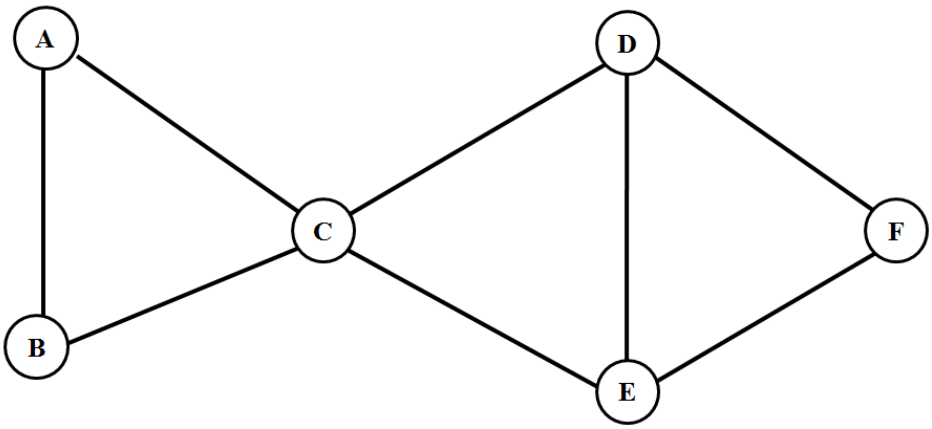


Figure O.17. **Connected Graph**

The above connected graph is not biconnected because the removal of node C makes the graph disconnected. Such disconnecting vertices are called as articulation points or separation vertices. Separation vertices and edges hold great significance in a network system. They help to identify the nodes which are critical to the smooth functioning of the network.

In a connected graph, articulation points can be ascertained by applying the Depth First Search technique.

Euler Circuit

Euler path is a continuous traversal path in a graph that traverses every edge exactly once; however it may traverse the edges multiple times. Euler circuit is a Euler path that begins and ends at the same node. Consider the graph shown in Figure O.18:

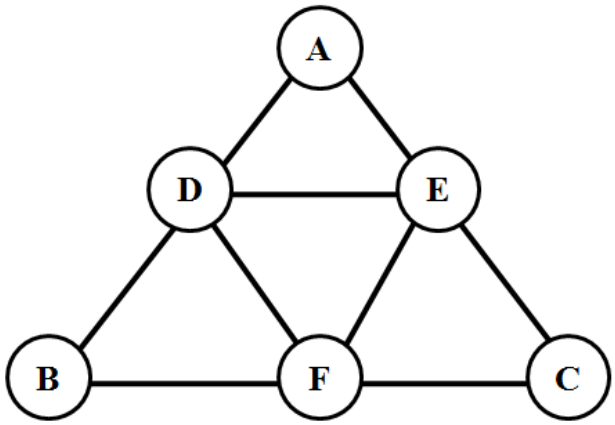


Figure O.18. **Euler Path**

In the above graph, the Euler circuit is:

```
A-D-E-F-D-B-F-C-E-A
```

Similarly, we can deduce other Euler circuits as well.

So, do all the graphs have Euler paths or Euler circuits? The answer is No. Then, what are the typical properties of a graph containing Euler path or Euler circuit? The answer is given by Euler's theorems.

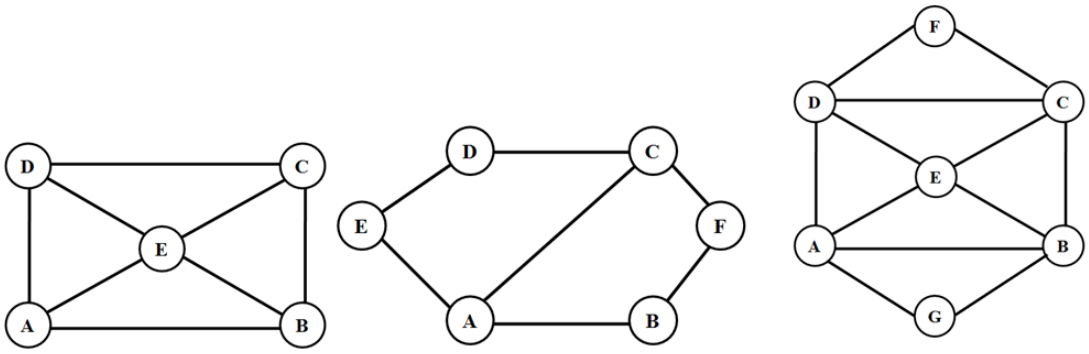
Euler's 1st Theorem

- If any of the vertices in a graph has an odd degree then the graph cannot have any Euler circuit.
- If all the graph vertices have an even degree then the graph must have at least one Euler circuit.

Euler's 2nd Theorem

- If two vertices in a graph have odd degree then the graph must have at least one Euler path.
- If more than two vertices in a graph have odd degree then the graph cannot have any Euler path.

Example O.1 Consider the graphs shown below:



Graph 1 **Graph 2** **Graph 3**

Apply Euler's theorem and find out whether these graphs have Euler Paths and Euler Circuits.

Solution:

Graph	Number of Vertices with Odd Degree	Number of Vertices with Even Degree	Euler Path/Euler Circuit?
Graph1	4	1	None
Graph 2	2	4	Euler path
Graph3	0	7	Euler Circuit

DISJOINT SET ADT

Disjoint Set ADT is used to represent a group of non-empty sets. That is,

$S = S_1, S_2, S_3, ..., S_n$

Here,

- All the sets are stored as trees
- None of the sets have common elements.
- Each set is identified by its root element called *representative* element

The operations performed on Disjoint Set ADT are:

- *find(x)*: Searches the set in which element x is present. A successful search is signified by the return of the representative element of that set.
- *union(x, y)*: Searches the sets in which x and y are present and combines the elements of the two sets to form a new set.

Disjoint Set algorithms are also referred as *union-find* algorithms. The applications of Disjoint Set ADT are:

- It helps in maintaining a set of connected components.
- It helps in constructing a minimum spanning tree (Kruskal's algorithm).

Dynamic Equivalence Problem

To understand the dynamic equivalence problem, you need to first understand an equivalence relation.

A relation R defined on a set S is an equivalence relation if:

- R is reflexive, that is **a R a**, for all $a \in S$
- R is symmetric, that is **a R b if and only if b R a**, where **(**a, b)$\in$S
- R is transitive, that is **a R b and b R c implies a R c**, where **(**a, b, c)$\in$S

An equivalence class of an element $a \in S$ is defined as a subset of S that contains all the elements related to a. Thus, equivalence classes are nothing but various subsets of S such that the elements of S appear only once in one equivalence class. To determine if two elements are related to each other, we need to simply find out whether the elements are present in the same equivalence class or not.

This forms the basis for solving the dynamic equivalence problem. It involves applying a series of find and union operations. Since the union operation alters the disjoint set compositions, the algorithm is considered as dynamic in nature.

The normal union operation involves combining the second tree (disjoint set) with the first one. A variation to the normal union operation is the *smart union algorithm* which involves the following:

- It makes the smaller tree a subtree of the large one
- It keeps a check on the height or size of the resultant tree

To further enhance the performance of the union-find algorithm, a variation called *path compression* is added to the find(x) algorithm. It involves changing the root of every element on the path from x to the root to the root element itself. This speeds up the running time of the successive find operations on the impacted nodes.