

What is a Trie data structure?

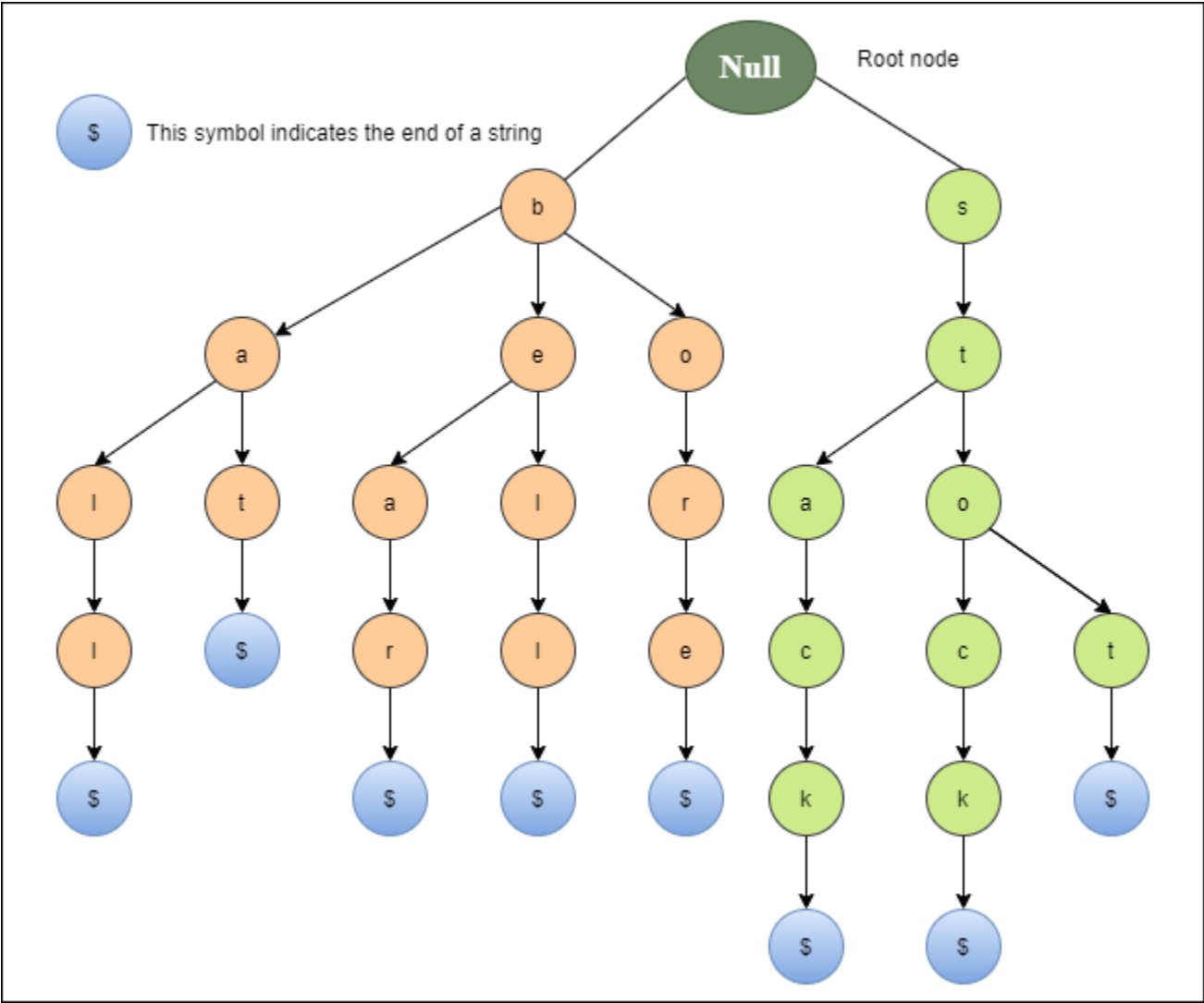
The word "Trie" is an excerpt from the word "retrieval". Trie is a sorted tree-based data-structure that stores the set of strings. It has the number of pointers equal to the number of characters of the alphabet in each node. It can search a word in the dictionary with the help of the word's prefix. For example, if we assume that all strings are formed from the letters 'a' to 'z' in the English alphabet, each trie node can have a maximum of 26 pointers.

Trie is also known as the digital tree or prefix tree. The position of a node in the Trie determines the key with which that node is connected.

Properties of the Trie for a set of the string:

- 1. The root node of the trie always represents the null node.
- 2. Each child of nodes is sorted alphabetically.
- 3. Each node can have a maximum of 26 children (A to Z).
- 4. Each node (except the root) can store one letter of the alphabet.

The diagram below depicts a trie representation for the bell, bear, bore, bat, ball, stop, stock, and stack.



Basic operations of Trie

There are three operations in the Trie:

- 1. Insertion of a node
- 2. Searching a node
- 3. Deletion of a node

Insert of a node in the Trie

The first operation is to insert a new node into the trie. Before we start the implementation, it is important to understand some points:

- 1. Every letter of the input key (word) is inserted as an individual in the Trie_node. Note that children point to the next level of Trie nodes.
- 2. The key character array acts as an index of children.
- 3. If the present node already has a reference to the present letter, set the present node to that referenced node. Otherwise, create a new node, set the letter to be equal to the present letter, and even start the present node with this new node.
- 4. The character length determines the depth of the trie.

Implementation of insert a new node in the Trie

```
1. public class Data_Trie {
2.     private Node_Trie root;
3.     public Data_Trie(){
4.         this.root = new Node_Trie();
5.     }
6.     public void insert(String word){
7.         Node_Trie current = root;
8.         int length = word.length();
9.         for (int x = 0; x < length; x++){
10.            char L = word.charAt(x);
11.            Node_Trie node = current.getNode().get(L);
12.            if (node == null){
13.                node = new Node_Trie ();
14.                current.getNode().put(L, node);
15.            }
16.            current = node;
17.        }
18.        current.setWord(true);
19.    }
20. }
```

Searching a node in Trie

The second operation is to search for a node in a Trie. The searching operation is similar to the insertion operation. The search operation is used to search a key in the trie. The implementation of the searching operation is shown below.

Implementation of search a node in the Trie

```
1. class Search_Trie {
2.
3.     private Node_Trie Prefix_Search(String W) {
4.         Node_Trie node = R;
5.         for (int x = 0; x < W.length(); x++) {
6.             char curLetter = W.charAt(x);
7.             if (node.containsKey(curLetter))
8.                 {
9.                     node = node.get(curLetter);
10.                }
11.             else {
12.                 return null;
13.             }
14.         }
15.         return node;
16.     }
17.
18.     public boolean search(String W) {
19.         Node_Trie node = Prefix_Search(W);
20.         return node != null && node.isEnd();
21.     }
22. }
```

Deletion of a node in the Trie

The Third operation is the deletion of a node in the Trie. Before we begin the implementation, it is important to understand some points:

1. If the key is not found in the trie, the delete operation will stop and exit it.
2. If the key is found in the trie, delete it from the trie.

Implementation of delete a node in the Trie

```

1. public void Node_delete(String W)
2. {
3.     Node_delete(R, W, 0);
4. }
5.
6. private boolean Node_delete(Node_Trie current, String W, int Node_index) {
7.     if (Node_index == W.length()) {
8.         if (!current.isEndOfWord()) {
9.             return false;
10.        }
11.        current.setEndOfWord(false);
12.        return current.getChildren().isEmpty();
13.    }
14.    char A = W.charAt(Node_index);
15.    Node_Trie node = current.getChildren().get(A);
16.    if (node == null) {
17.        return false;
18.    }
19.    boolean Current_Node_Delete = Node_delete(node, W, Node_index + 1) && !node.isEndOfWord();
20.
21.    if (Current_Node_Delete) {
22.        current.getChildren().remove(A);
23.        return current.getChildren().isEmpty();
24.    }
25.    return false;
26. }

```

Applications of Trie

1. Spell Checker

Spell checking is a three-step process. First, look for that word in a dictionary, generate possible suggestions, and then sort the suggestion words with the desired word at the top.

Trie is used to store the word in dictionaries. The spell checker can easily be applied in the most efficient way by searching for words on a data structure. Using trie not only makes it easy to see the word in the dictionary, but it is also simple to build an algorithm to include a collection of relevant words or suggestions.

2. Auto-complete

Auto-complete functionality is widely used on text editors, mobile applications, and the Internet. It provides a simple way to find an alternative word to complete the word for the following reasons.

- It provides an alphabetical filter of entries by the key of the node.
- We trace pointers only to get the node that represents the string entered by the user.
- As soon as you start typing, it tries to complete your input.

3. Browser history

It is also used to complete the URL in the browser. The browser keeps a history of the URLs of the websites you've visited.

Advantages of Trie

1. It can be insert faster and search the string than hash tables and binary search trees.
2. It provides an alphabetical filter of entries by the key of the node.

Disadvantages of Trie

1. It requires more memory to store the strings.
2. It is slower than the hash table.

Complete program in C++

```

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>
4. #define N 26

```

```

5.
6. typedef struct TrieNode TrieNode;
7.
8. struct TrieNode {
9.     char info;
10.    TrieNode* child[N];
11.    int data;
12. };
13.
14. TrieNode* trie_make(char info) {
15.    TrieNode* node = (TrieNode*) calloc (1, sizeof(TrieNode));
16.    for (int i = 0; i < N; i++)
17.        node → child[i] = NULL;
18.    node → data = 0;
19.    node → info = info;
20.    return node;
21. }
22.
23. void free_trienode(TrieNode* node) {
24.    for(int i = 0; i < N; i++) {
25.        if (node → child[i] != NULL) {
26.            free_trienode(node → child[i]);
27.        }
28.        else {
29.            continue;
30.        }
31.    }
32.    free(node);
33. }
34.
35. // Trie node loop start
36. TrieNode* trie_insert(TrieNode* flag, char* word) {
37.    TrieNode* temp = flag;
38.    for (int i = 0; word[i] != '\0'; i++) {
39.        int idx = (int) word[i] - 'a';
40.        if (temp → child[idx] == NULL) {
41.            temp → child[idx] = trie_make(word[i]);
42.        }
43.        else {
44.        }
45.        temp = temp → child[idx];
46.    }trie
47.    temp → data = 1;
48.    return flag;
49. }
50.
51. int search_trie(TrieNode* flag, char* word)
52. {
53.    TrieNode* temp = flag;
54.
55.    for(int i = 0; word[i] != '\0'; i++)
56.    {
57.        int position = word[i] - 'a';
58.        if (temp → child[position] == NULL)
59.            return 0;
60.        temp = temp → child[position];
61.    }

```

```

62.     if (temp != NULL && temp → data == 1)
63.         return 1;
64.     return 0;
65. }
66.
67. int check_divergence(TrieNode* flag, char* word) {
68.     TrieNode* temp = flag;
69.     int len = strlen(word);
70.     if (len == 0)
71.         return 0;
72.     int last_index = 0;
73.     for (int i = 0; i < len; i++) {
74.         int position = word[i] - 'a';
75.         if (temp → child[position]) {
76.             for (int j = 0; j < N; j++) {
77.                 if (j != position && temp → child[j]) {
78.                     last_index = i + 1;
79.                     break;
80.                 }
81.             }
82.             temp = temp → child[position];
83.         }
84.     }
85.     return last_index;
86. }
87.
88. char* find_longest_prefix(TrieNode* flag, char* word) {
89.     if (!word || word[0] == '\0')
90.         return NULL;
91.     int len = strlen(word);
92.
93.     char* longest_prefix = (char*) calloc (len + 1, sizeof(char));
94.     for (int i = 0; word[i] != '\0'; i++)
95.         longest_prefix[i] = word[i];
96.     longest_prefix[len] = '\0';
97.
98.     int branch_idx = check_divergence(flag, longest_prefix) - 1;
99.     if (branch_idx >= 0) {
100.         longest_prefix[branch_idx] = '\0';
101.         longest_prefix = (char*) realloc (longest_prefix, (branch_idx + 1) * sizeof(char));
102.     }
103.
104.     return longest_prefix;
105. }
106.
107. int data_node(TrieNode* flag, char* word) {
108.     TrieNode* temp = flag;
109.     for (int i = 0; word[i]; i++) {
110.         int position = (int) word[i] - 'a';
111.         if (temp → child[position]) {
112.             temp = temp → child[position];
113.         }
114.     }
115.     return temp → data;
116. }
117.
118. TrieNode* trie_delete(TrieNode* flag, char* word) {

```

```

119.     if (!flag)
120.         return NULL;
121.     if (!word || word[0] == '\0')
122.         return flag;
123.     if (!data_node(flag, word)) {
124.         return flag;
125.     }
126.     TrieNode* temp = flag;
127.     char* longest_prefix = find_longest_prefix(flag, word);
128.     if (longest_prefix[0] == '\0') {
129.         free(longest_prefix);
130.         return flag;
131.     }
132.     int i;
133.     for (i = 0; longest_prefix[i] != '\0'; i++) {
134.         int position = (int) longest_prefix[i] - 'a';
135.         if (temp → child[position] != NULL) {
136.             temp = temp → child[position];
137.         }
138.         else {
139.             free(longest_prefix);
140.             return flag;
141.         }
142.     }
143.     int len = strlen(word);
144.     for (; i < len; i++) {
145.         int position = (int) word[i] - 'a';
146.         if (temp → child[position]) {
147.             TrieNode* rm_node = temp → child[position];
148.             temp → child[position] = NULL;
149.             free_trienode(rm_node);
150.         }
151.     }
152.     free(longest_prefix);
153.     return flag;
154. }

155.
156. void print_trie(TrieNode* flag) {
157.     if (!flag)
158.         return;
159.     TrieNode* temp = flag;
160.     printf("%c → ", temp → info);
161.     for (int i = 0; i < N; i++) {
162.         print_trie(temp → child[i]);
163.     }
164. }

165.
166. void search(TrieNode* flag, char* word) {
167.     printf("Search the word %s: ", word);
168.     if (search_trie(flag, word) == 0)
169.         printf("Not Found\n");
170.     else
171.         printf("Found!\n");
172. }

173.
174. int main() {
175.     TrieNode* flag = trie_make('\0');

```

```
176.     flag = trie_insert(flag, "oh");
177.     flag = trie_insert(flag, "way");
178.     flag = trie_insert(flag, "bag");
179.     flag = trie_insert(flag, "can");
180.     search(flag, "ohh");
181.     search(flag, "bag");
182.     search(flag, "can");
183.     search(flag, "ways");
184.     search(flag, "way");
185.     print_trie(flag);
186.     printf("\n");
187.     flag = trie_delete(flag, "oh");
188.     printf("deleting the word 'hello'...\n");
189.     print_trie(flag);
190.     printf("\n");
191.     flag = trie_delete(flag, "can");
192.     printf("deleting the word 'can'...\n");
193.     print_trie(flag);
194.     printf("\n");
195.     free_trienode(flag);
196.     return 0;
197. }
```

Output

```
Search the word ohh: Not Found
Search the word bag: Found!
Search the word can: Found!
Search the word ways: Not Found
Search the word way: Found!
→ h → e → l → l → o → w → a → y → i → t → e → a → b → a → g → c → a → n
deleting the word 'hello'...
→ w → a → y → h → i → t → e → a → b → a → g → c → a → n
deleting the word 'can'...
→ w → a → y → h → i → t → e → a → b → a → g
```

Heap Data Structure

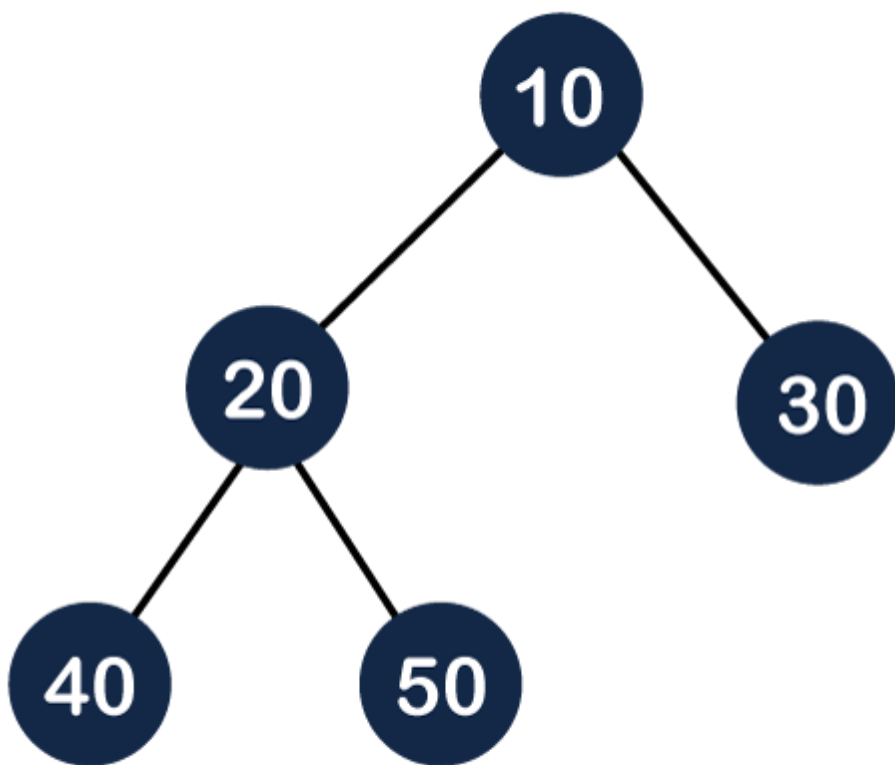
What is Heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap *data structure*, we should know about the complete binary tree.

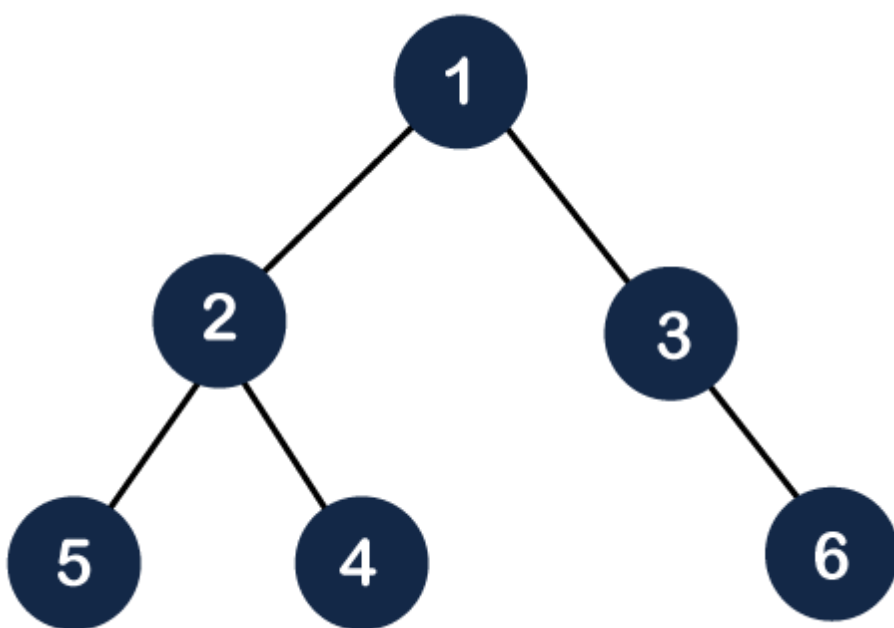
What is a complete binary tree?

A complete binary tree is a *binary tree* in which all the levels except the last level, i.e., leaf node should be completely filled, and all the nodes should be left-justified.

Let's understand through an example.



In the above figure, we can observe that all the internal nodes are completely filled except the leaf node; therefore, we can say that the above tree is a complete binary tree.



The above figure shows that all the internal nodes are completely filled except the leaf node, but the leaf nodes are added at the right part; therefore, the above tree is not a complete binary tree.

Note: The heap tree is a special balanced binary tree data structure where the root node is compared with its children and arrange accordingly.

How can we arrange the nodes in the Tree?

There are two types of the heap:

- Min Heap
- Max heap

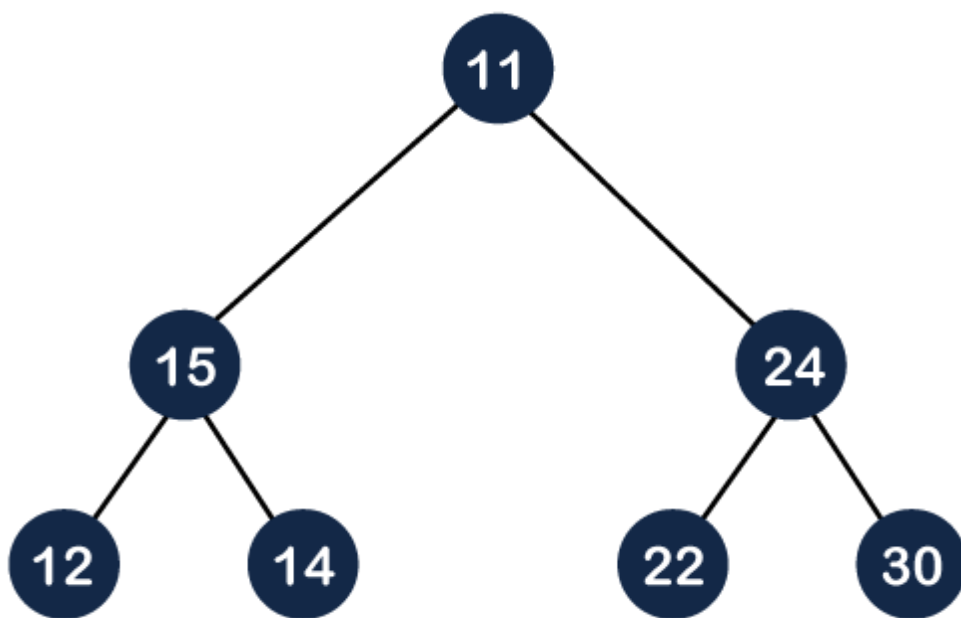
Min Heap: The value of the parent node should be less than or equal to either of its children.

Or

In other words, the min-heap can be defined as, for every node i , the value of node i is greater than or equal to its parent value except the root node. Mathematically, it can be defined as:

$$A[\text{Parent}(i)] \leq A[i]$$

Let's understand the min-heap through an example.



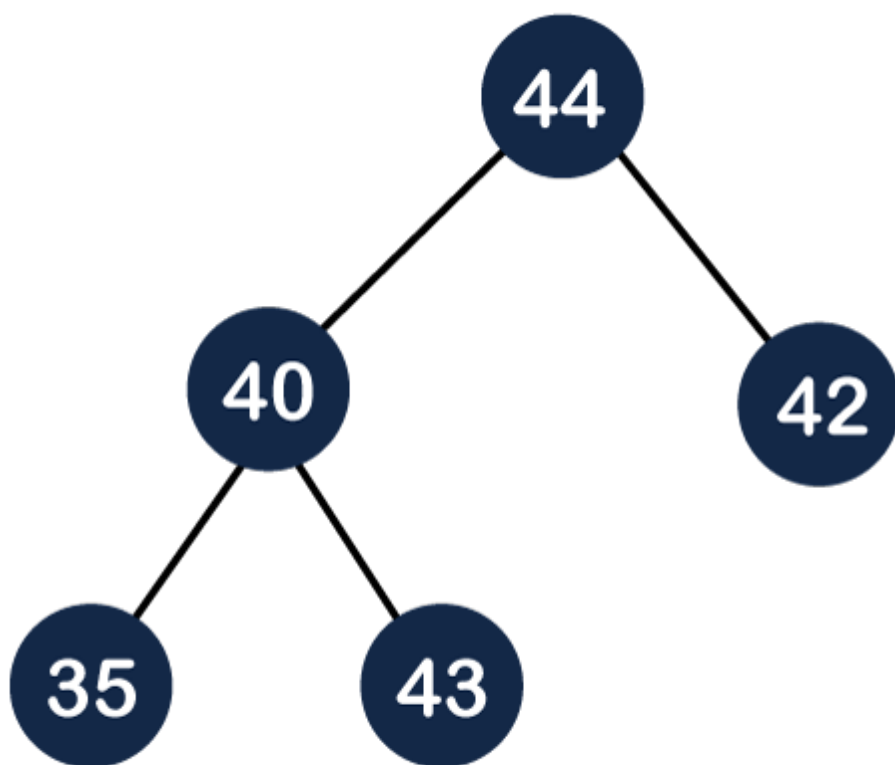
In the above figure, 11 is the root node, and the value of the root node is less than the value of all the other nodes (left child or a right child).

Max Heap: The value of the parent node is greater than or equal to its children.

Or

In other words, the max heap can be defined as for every node i ; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

$$A[\text{Parent}(i)] \geq A[i]$$



The above tree is a max heap tree as it satisfies the property of the max heap. Now, let's see the array representation of the max heap.

Time complexity in Max Heap

The total number of comparisons required in the max heap is according to the height of the tree. The height of the complete binary tree is always $\log n$; therefore, the time complexity would also be $O(\log n)$.

Algorithm of insert operation in the max heap.

1. // algorithm to insert an element in the max heap.
2. insertHeap(A, n, value)
3. {
4. $n = n + 1$; // n is incremented to insert the new element
5. $A[n] = \text{value}$; // assign new value at the nth position
6. $i = n$; // assign the value of n to i
7. // loop will be executed until i becomes 1.
8. while($i > 1$)
9. {
10. parent = floor value of $i/2$; // Calculating the floor value of $i/2$
11. // Condition to check whether the value of parent is less than the given node or not
12. if($A[\text{parent}] < A[i]$)

```
13. {  
14. swap(A[parent], A[i]);  
15. i = parent;  
16. }  
17. else  
18. {  
19. return;  
20. }  
21. }  
22. }
```

Let's understand the max heap through an example.

In the above figure, 55 is the parent node and it is greater than both of its child, and 11 is the parent of 9 and 8, so 11 is also greater than from both of its child. Therefore, we can say that the above tree is a max heap tree.

Insertion in the Heap tree

44, 33, 77, 11, 55, 88, 66

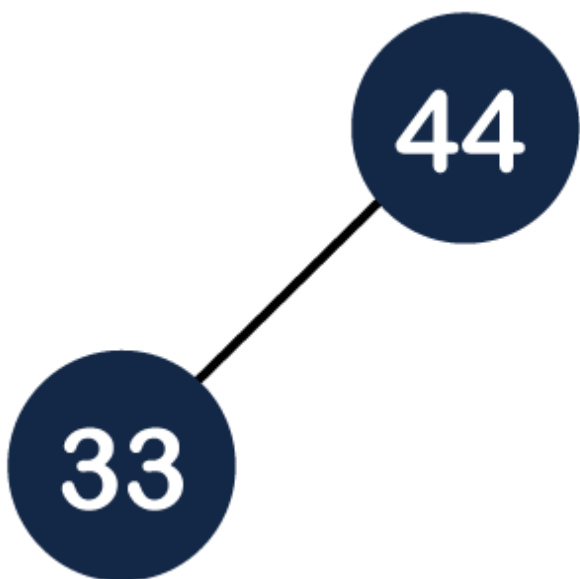
Suppose we want to create the max heap tree. To create the max heap tree, we need to consider the following two cases:

- First, we have to insert the element in such a way that the property of the complete binary tree must be maintained.
- Secondly, the value of the parent node should be greater than the either of its child.

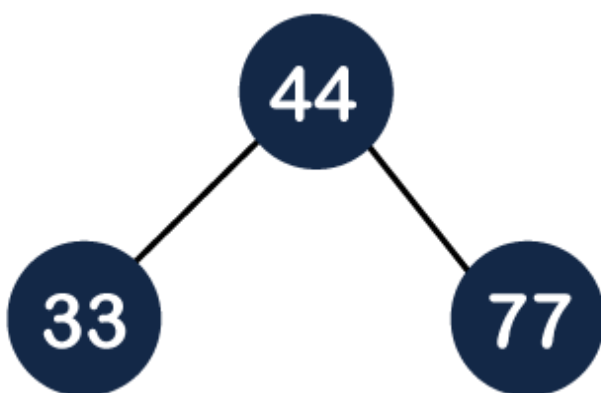
Step 1: First we add the 44 element in the tree as shown below:



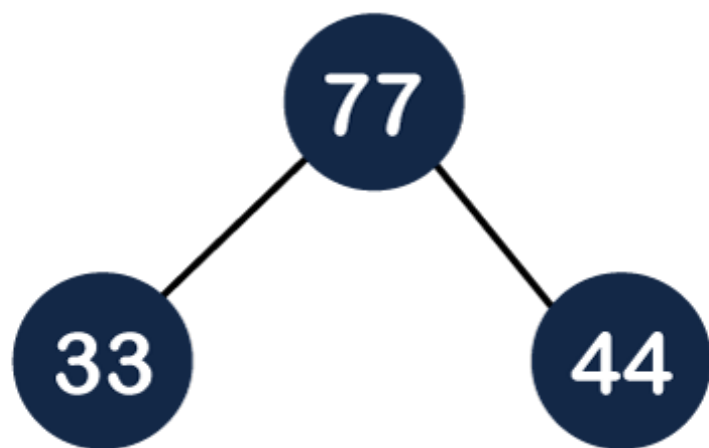
Step 2: The next element is 33. As we know that insertion in the binary tree always starts from the left side so 44 will be added at the left of 33 as shown below:



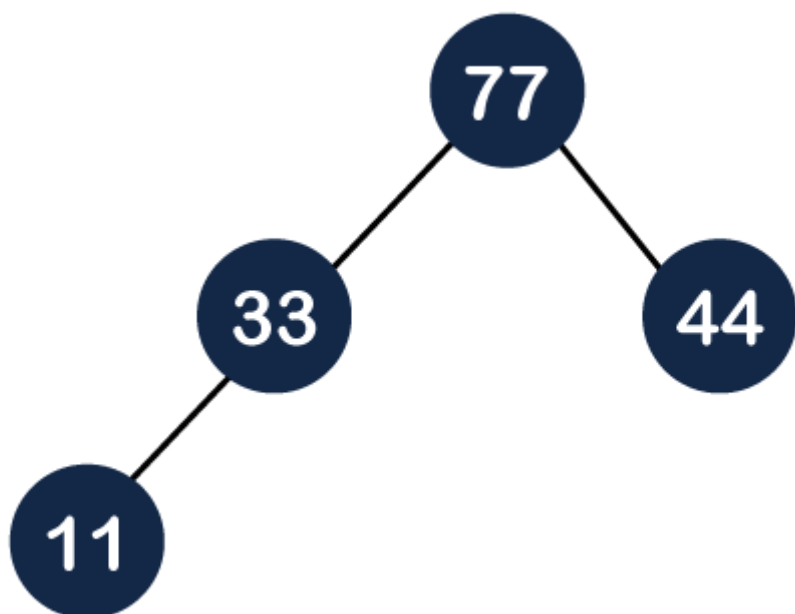
Step 3: The next element is 77 and it will be added to the right of the 44 as shown below:



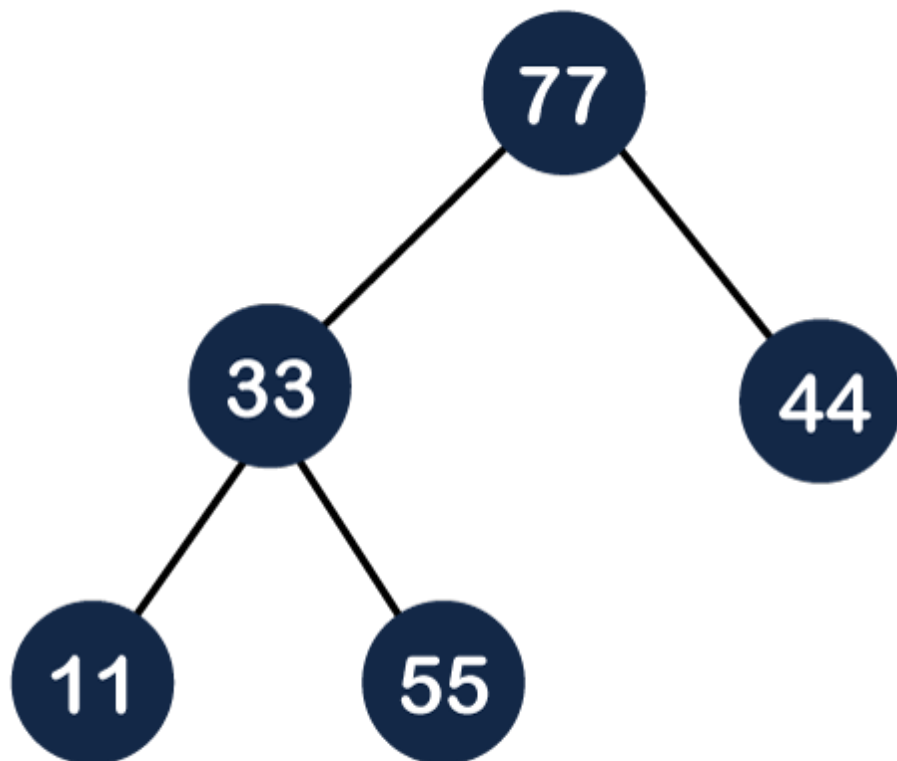
As we can observe in the above tree that it does not satisfy the max heap property, i.e., parent node 44 is less than the child 77. So, we will swap these two values as shown below:



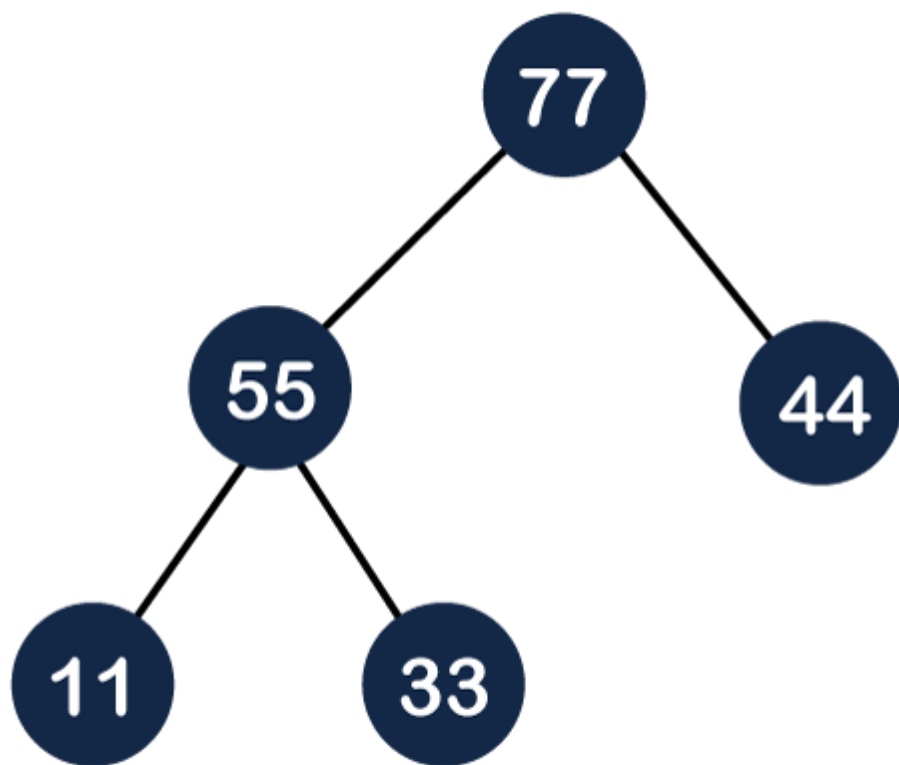
Step 4: The next element is 11. The node 11 is added to the left of 33 as shown below:



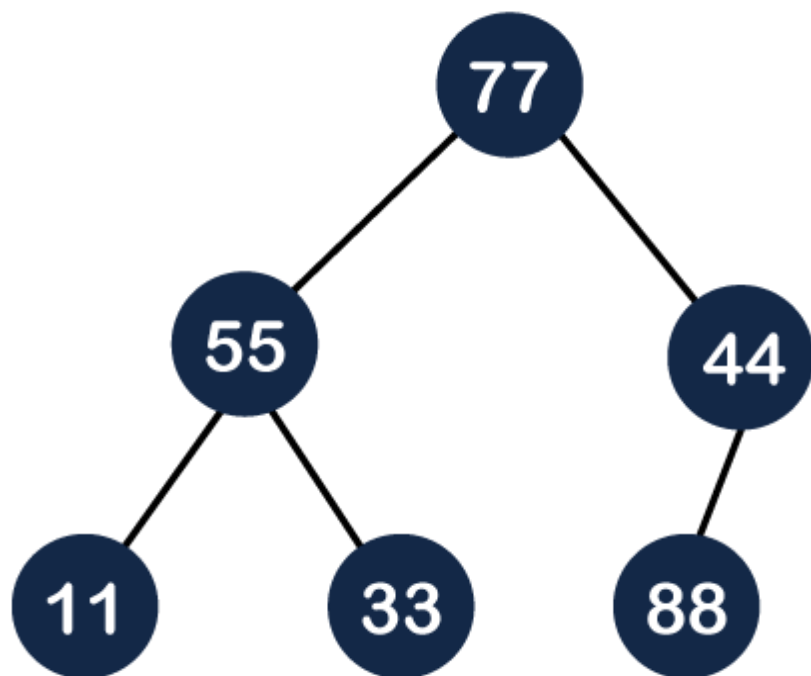
Step 5: The next element is 55. To make it a complete binary tree, we will add the node 55 to the right of 33 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because $33 < 55$, so we will swap these two values as shown below:



Step 6: The next element is 88. The left subtree is completed so we will add 88 to the left of 44 as shown below:



As we can observe in the above figure that it does not satisfy the property of the max heap because $44 < 88$, so we will swap these two values as shown below:

Again, it is violating the max heap property because $88 > 77$ so we will swap these two values as shown below:

Step 7: The next element is 66. To make a complete binary tree, we will add the 66 element to the right side of 77 as shown below:

In the above figure, we can observe that the tree satisfies the property of max heap; therefore, it is a heap tree.

Deletion in Heap Tree

In Deletion in the heap tree, the root node is always deleted and it is replaced with the last element.

Let's understand the deletion through an example.

Step 1: In the above tree, the first 30 node is deleted from the tree and it is replaced with the 15 element as shown below:

Now we will heapify the tree. We will check whether the 15 is greater than either of its child or not. 15 is less than 20 so we will swap these two values as shown below:

Again, we will compare 15 with its child. Since 15 is greater than 10 so no swapping will occur.

Algorithm to heapify the tree

1. MaxHeapify(A, n, i)
2. {
3. int largest = i;
4. int l = 2i;
5. int r = 2i+1;
6. while(l <= n && A[l] > A[largest])
7. {
8. largest = l;

```

9.  }
10. while(r<=n && A[r]>A[largest])
11. {
12. largest=r;
13. }
14. if(largest!=i)
15. {
16. swap(A[largest], A[i]);
17. heapify(A, n, largest);
18. }}

```

Splay Tree

Splay trees are the self-balancing or self-adjusted binary search trees. In other words, we can say that the splay trees are the variants of the binary search trees. The prerequisite for the splay trees that we should know about the binary search trees.

As we already know, the time complexity of a binary search tree in every case. The time complexity of a binary search tree in the average case is $O(\log n)$ and the time complexity in the worst case is $O(n)$. In a binary search tree, the value of the left subtree is smaller than the root node, and the value of the right subtree is greater than the root node; in such case, the time complexity would be $O(\log n)$. If the binary tree is left-skewed or right-skewed, then the time complexity would be $O(n)$. To limit the skewness, the *AVL and Red-Black tree* came into the picture, having $O(\log n)$ time complexity for all the operations in all the cases. We can also improve this time complexity by doing more practical implementations, so the new Tree *data structure* was designed, known as a Splay tree.

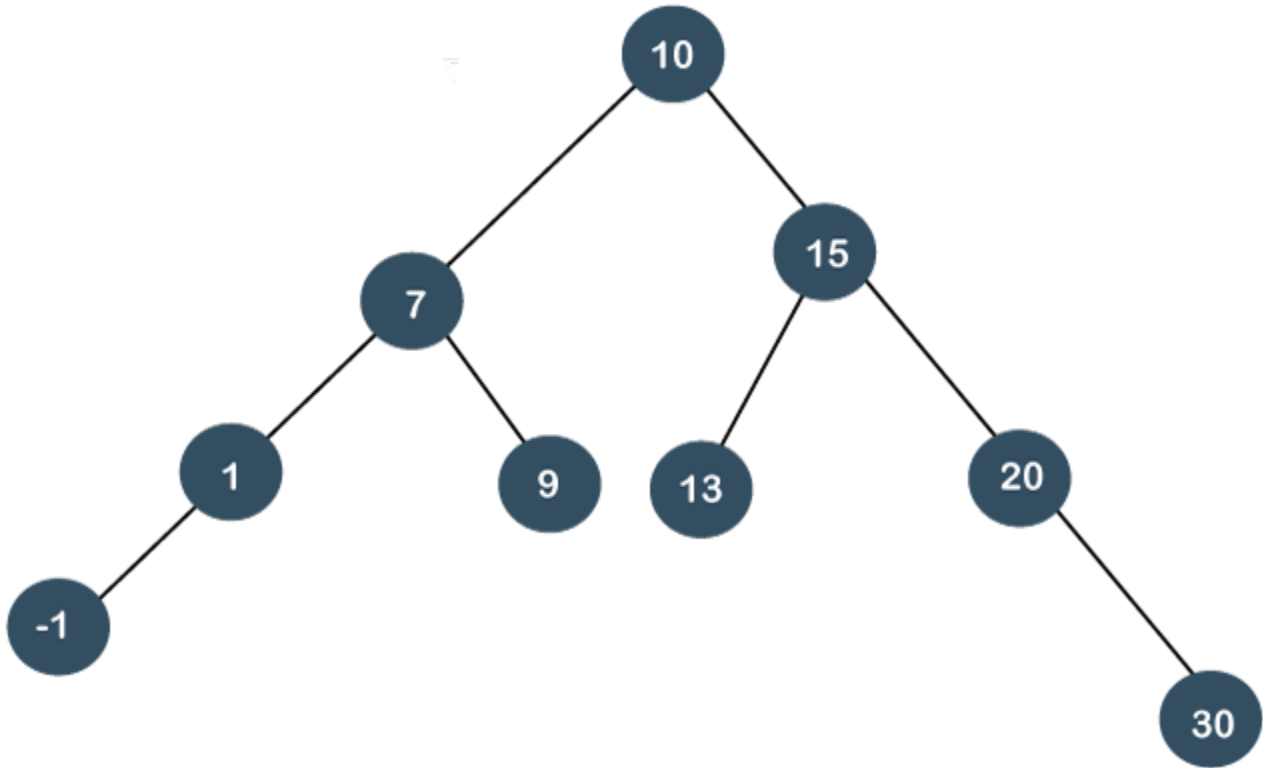
What is a Splay Tree?

A splay tree is a self-balancing tree, but *AVL* and *Red-Black trees* are also self-balancing trees then. What makes the splay tree unique two trees. It has one extra property that makes it unique is splaying.

A splay tree contains the same operations as a *Binary search tree*, i.e., Insertion, deletion and searching, but it also contains one more operation, i.e., splaying. So. all the operations in the splay tree are followed by splaying.

Splay trees are not strictly balanced trees, but they are roughly balanced trees. Let's understand the search operation in the splay-tree.

Suppose we want to search 7 element in the tree, which is shown below:



To search any element in the splay tree, first, we will perform the standard binary search tree operation. As 7 is less than 10 so we will come to the left of the root node. After performing the search operation, we need to perform splaying. Here splaying means that the operation that we are performing on any element should become the root node after performing some rearrangements. The rearrangement of the tree will be done through the rotations.

Note: The splay tree can be defined as the self-adjusted tree in which any operation performed on the element would rearrange the tree so that the element on which operation has been performed becomes the root node of the tree.

Rotations

There are six types of rotations used for splaying:

1. Zig rotation (Right rotation)
2. Zag rotation (Left rotation)
3. Zig zag (Zig followed by zag)
4. Zag zig (Zag followed by zig)
5. Zig zig (two right rotations)
6. Zag zag (two left rotations)

Factors required for selecting a type of rotation

The following are the factors used for selecting a type of rotation:

- Does the node which we are trying to rotate have a grandparent?
- Is the node left or right child of the parent?
- Is the node left or right child of the grandparent?

Cases for the Rotations

Case 1: If the node does not have a grand-parent, and if it is the right child of the parent, then we carry out the left rotation; otherwise, the right rotation is performed.

Case 2: If the node has a grandparent, then based on the following scenarios; the rotation would be performed:

Scenario 1: If the node is the right of the parent and the parent is also right of its parent, then **zig zig right right rotation** is performed.

Scenario 2: If the node is left of a parent, but the parent is right of its parent, then **zig zag right left rotation** is performed.

Scenario 3: If the node is right of the parent and the parent is right of its parent, then **zig zig left left rotation** is performed.

Scenario 4: If the node is right of a parent, but the parent is left of its parent, then **zig zag right-left rotation** is performed.

Now, let's understand the above rotations with examples.

To rearrange the tree, we need to perform some rotations. The following are the types of rotations in the splay tree:

- Zig rotations

The zig rotations are used when the item to be searched is either a root node or the child of a root node (i.e., left or the right child).

The following are the cases that can exist in the splay tree while searching:

Case 1: If the search item is a root node of the tree.

Case 2: If the search item is a child of the root node, then the two scenarios will be there:

1. If the child is a left child, the right rotation would be performed, known as a zig right rotation.
2. If the child is a right child, the left rotation would be performed, known as a zig left rotation.

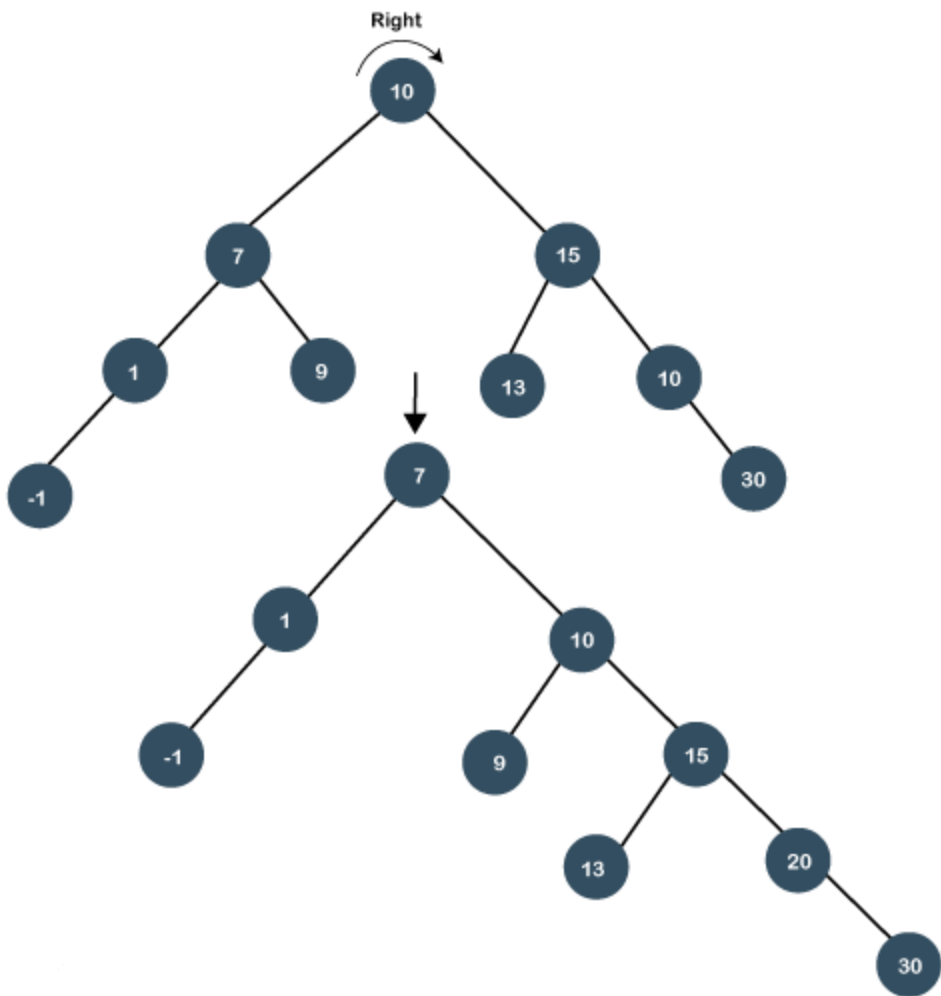
Let's look at the above two scenarios through an example.

Consider the below example:

In the above example, we have to search 7 element in the tree. We will follow the below steps:

Step 1: First, we compare 7 with a root node. As 7 is less than 10, so it is a left child of the root node.

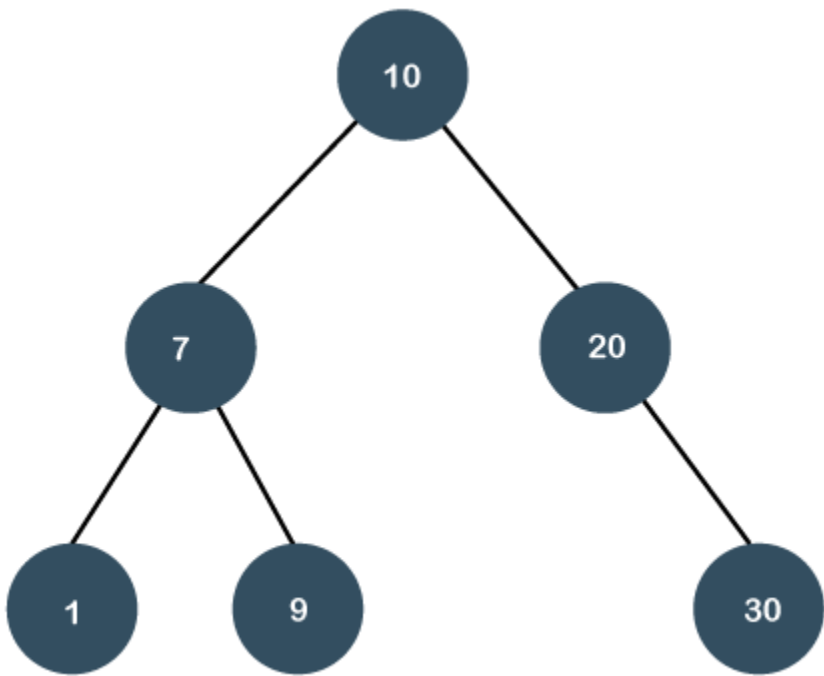
Step 2: Once the element is found, we will perform splaying. The right rotation is performed so that 7 becomes the root node of the tree, as shown below:



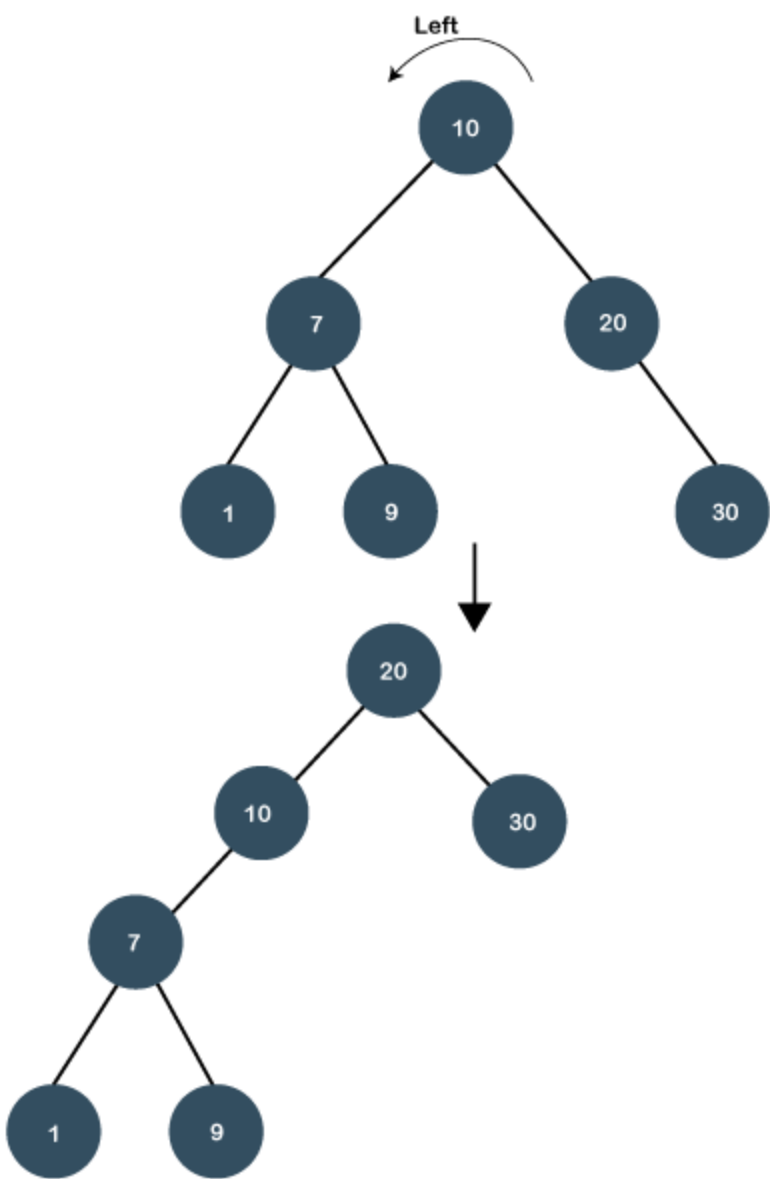
Let's consider another example.

In the above example, we have to search 20 element in the tree. We will follow the below steps:

Step 1: First, we compare 20 with a root node. As 20 is greater than the root node, so it is a right child of the root node.



Step 2: Once the element is found, we will perform splaying. The left rotation is performed so that 20 element becomes the root node of the tree.



- Zig zig rotations

Sometimes the situation arises when the item to be searched is having a parent as well as a grandparent. In this case, we have to perform four rotations for splaying.

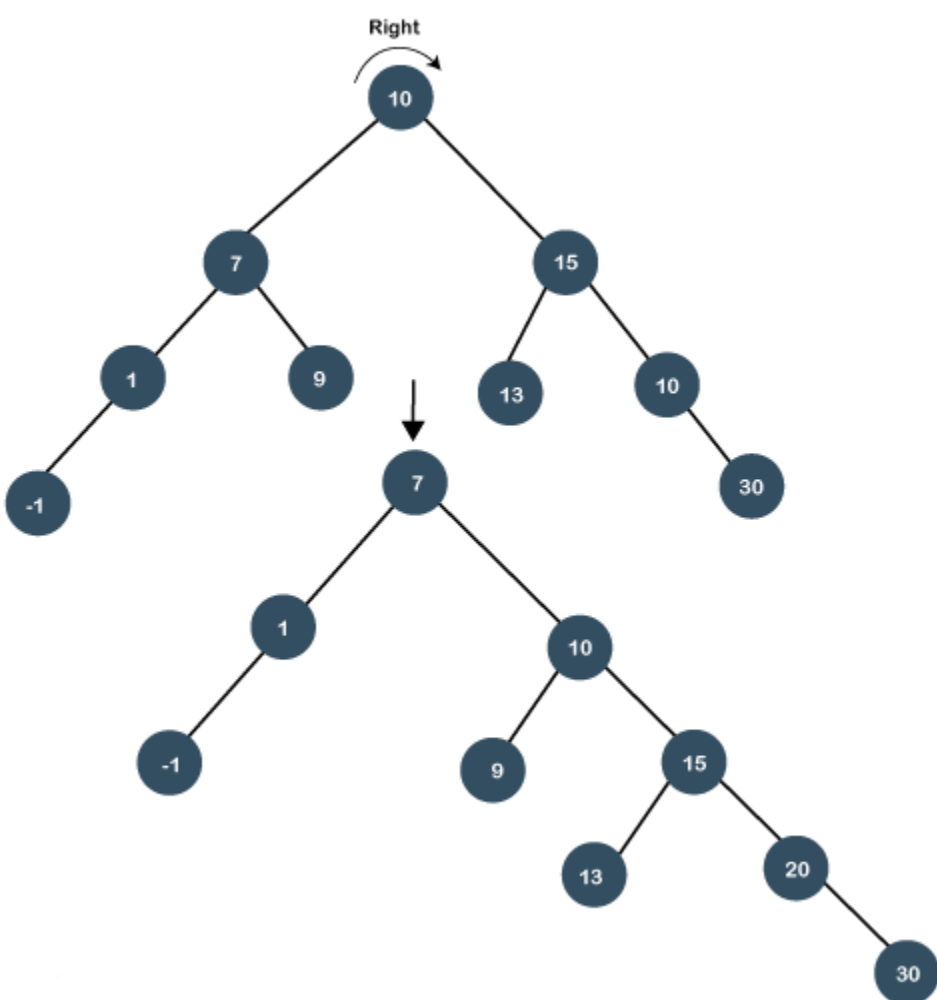
Let's understand this case through an example.

Suppose we have to search 1 element in the tree, which is shown below:

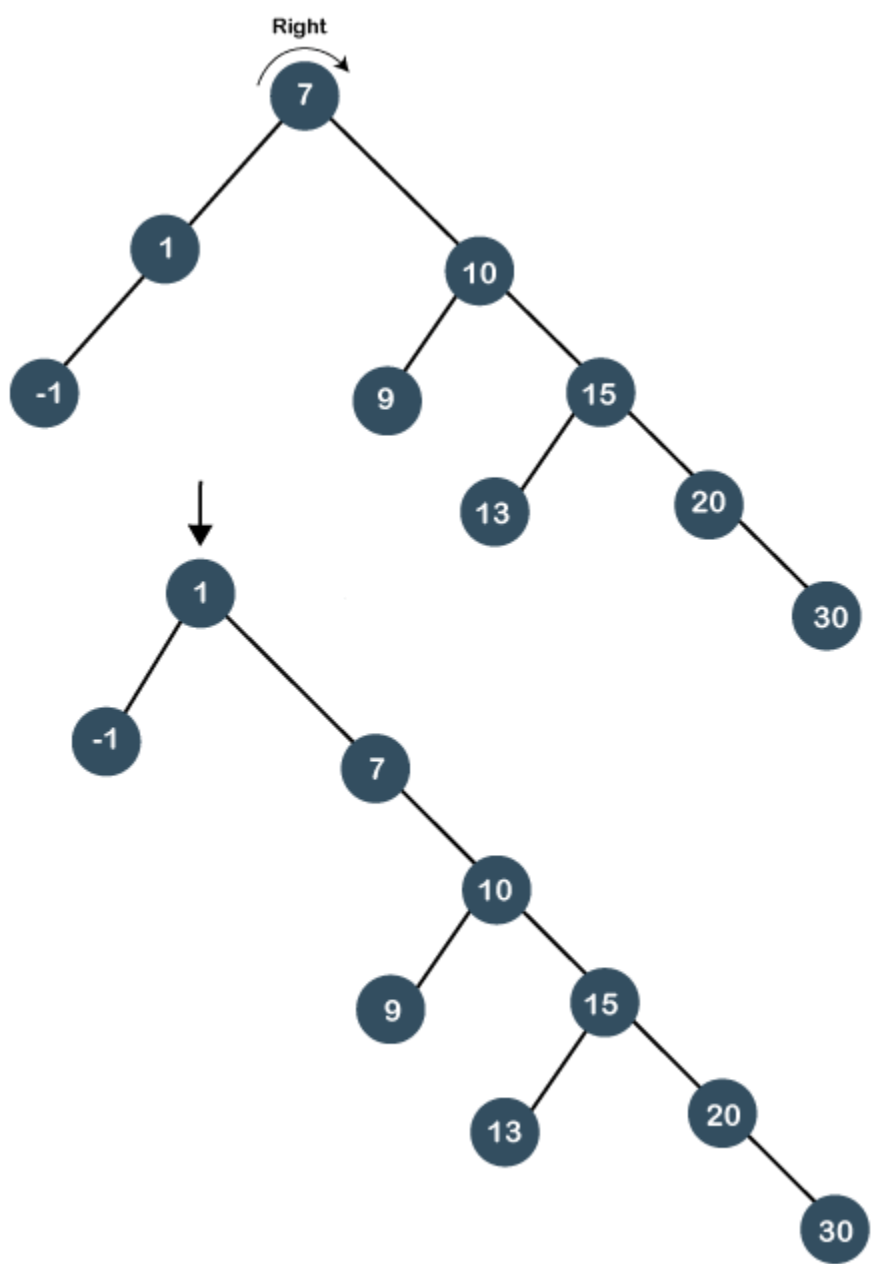
Step 1: First, we have to perform a standard BST searching operation in order to search the 1 element. As 1 is less than 10 and 7, so it will be at the left of the node 7. Therefore, element 1 is having a parent, i.e., 7 as well as a grandparent, i.e., 10.

Step 2: In this step, we have to perform splaying. We need to make node 1 as a root node with the help of some rotations. In this case, we cannot simply perform a zig or zag rotation; we have to implement zig zig rotation.

In order to make node 1 as a root node, we need to perform two right rotations known as zig zig rotations. When we perform the right rotation then 10 will move downwards, and node 7 will come upwards as shown in the below figure:



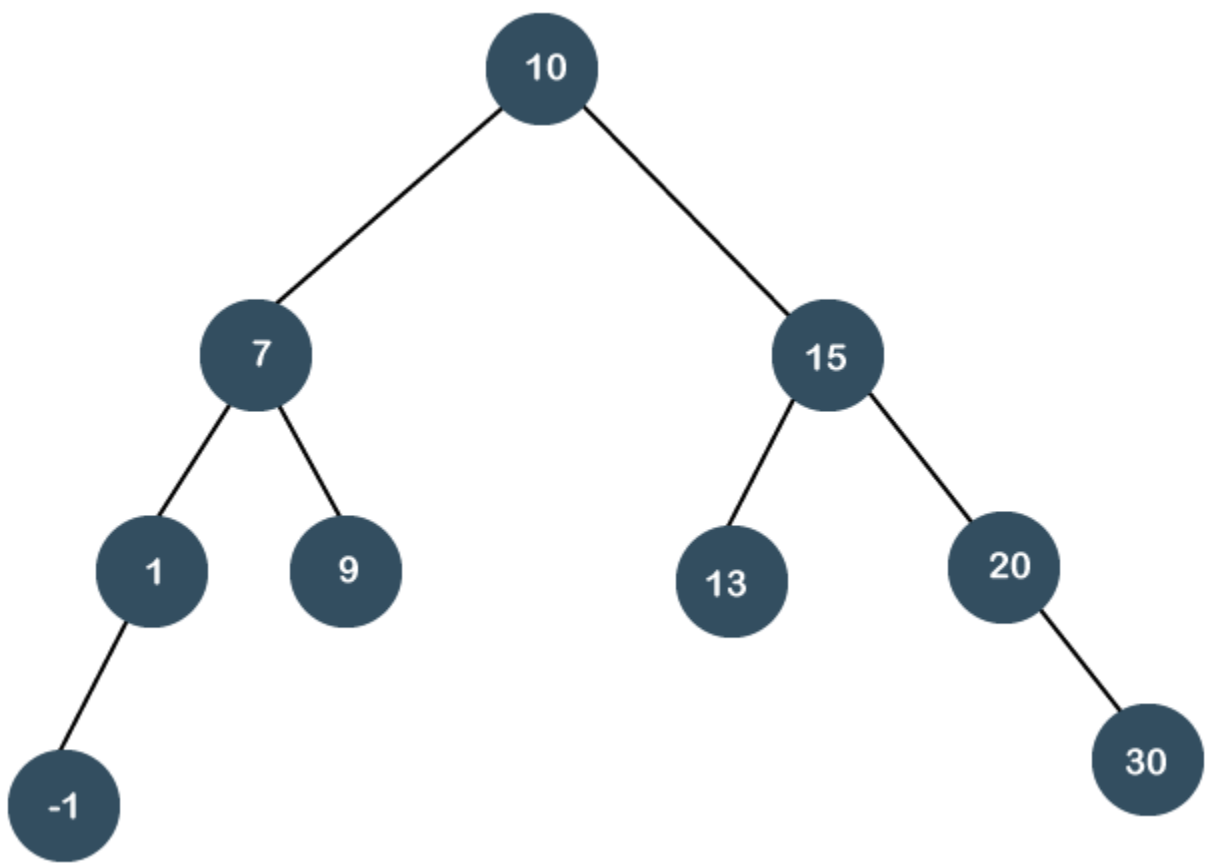
Again, we will perform zig right rotation, node 7 will move downwards, and node 1 will come upwards as shown below:



As we observe in the above figure that node 1 has become the root node of the tree; therefore, the searching is completed.

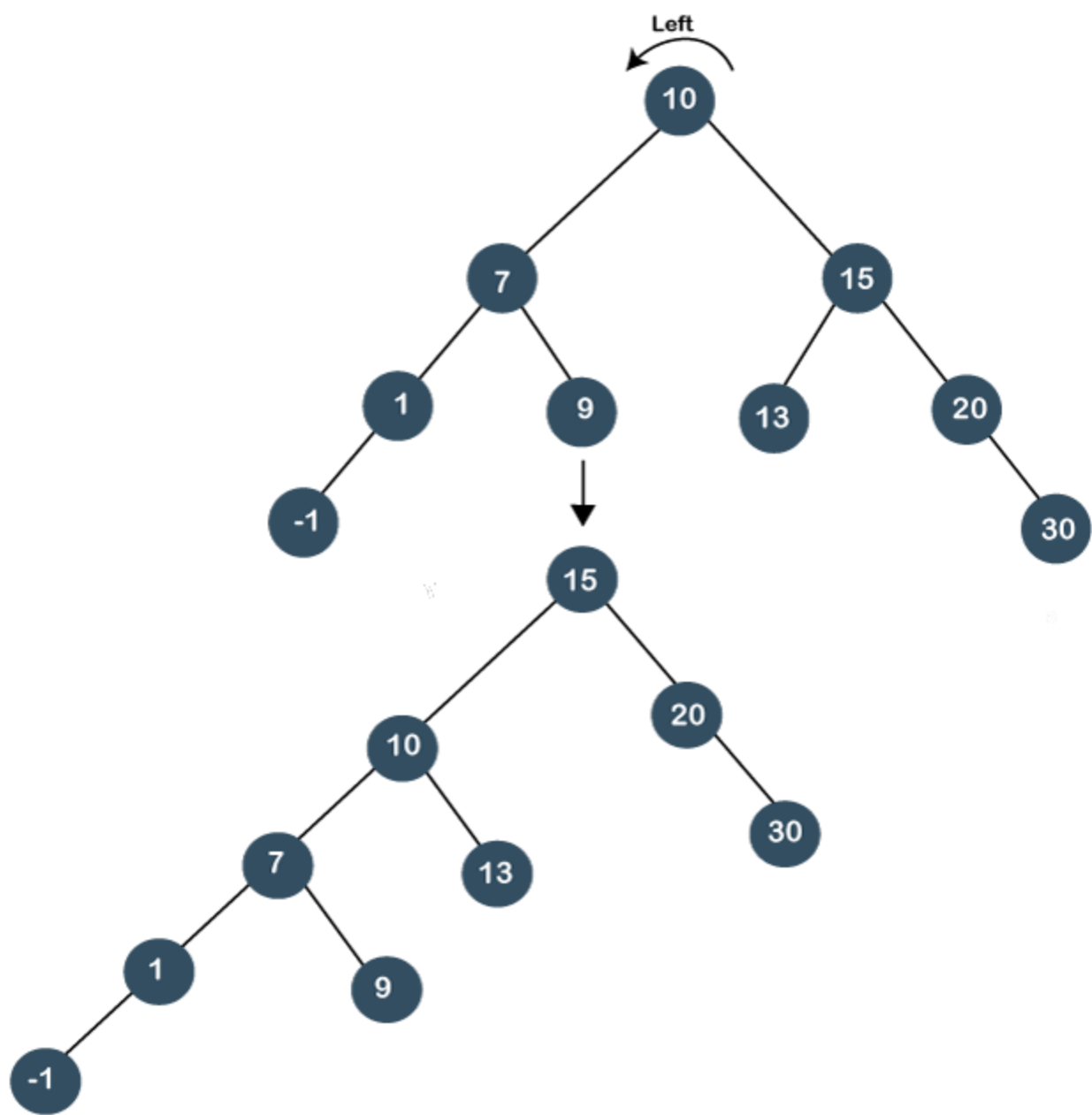
Suppose we want to search 20 in the below tree.

In order to search 20, we need to perform two left rotations. Following are the steps required to search 20 node:

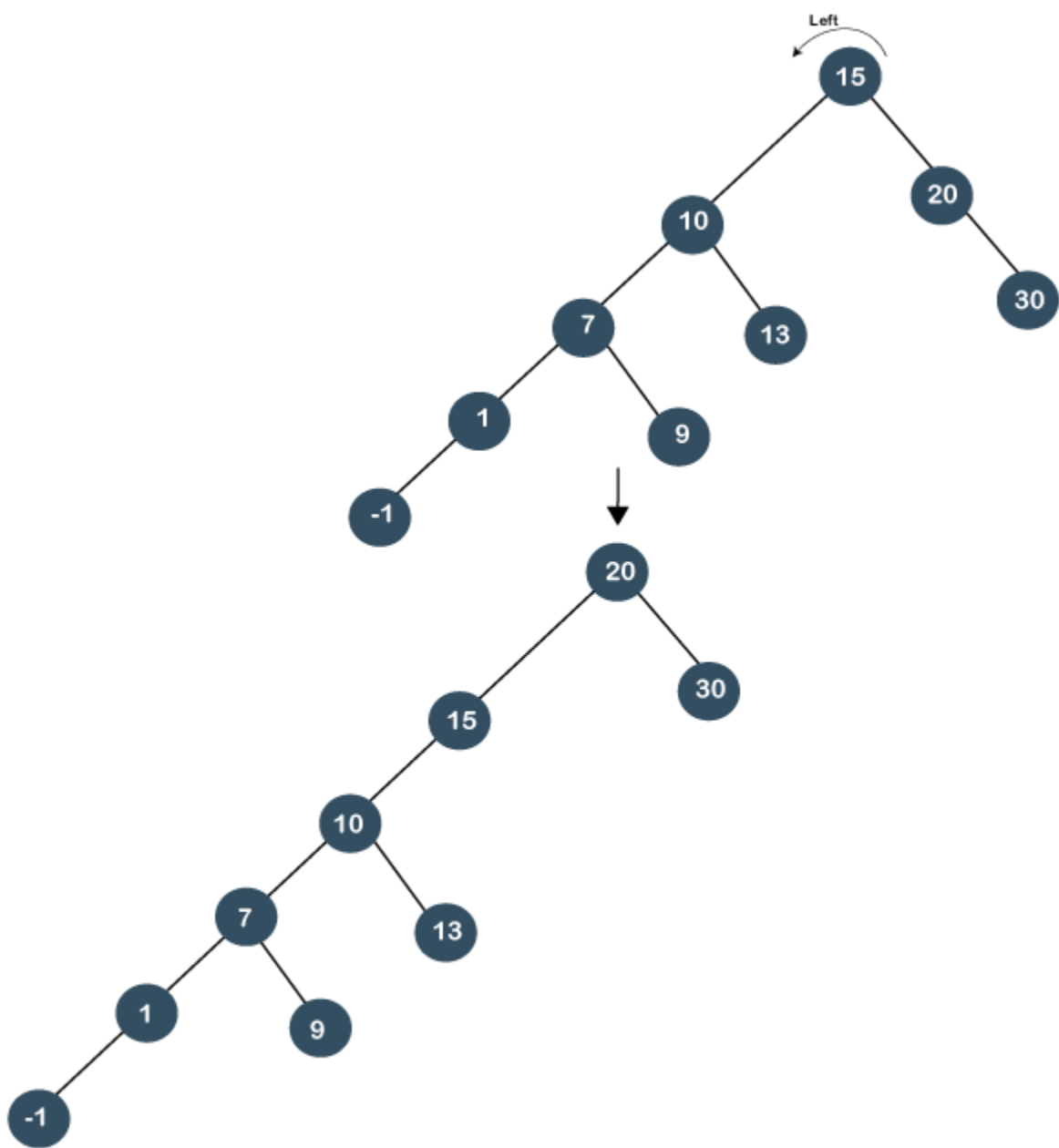


Step 1: First, we perform the standard BST searching operation. As 20 is greater than 10 and 15, so it will be at the right of node 15.

Step 2: The second step is to perform splaying. In this case, two left rotations would be performed. In the first rotation, node 10 will move downwards, and node 15 would move upwards as shown below:



In the second left rotation, node 15 will move downwards, and node 20 becomes the root node of the tree, as shown below:



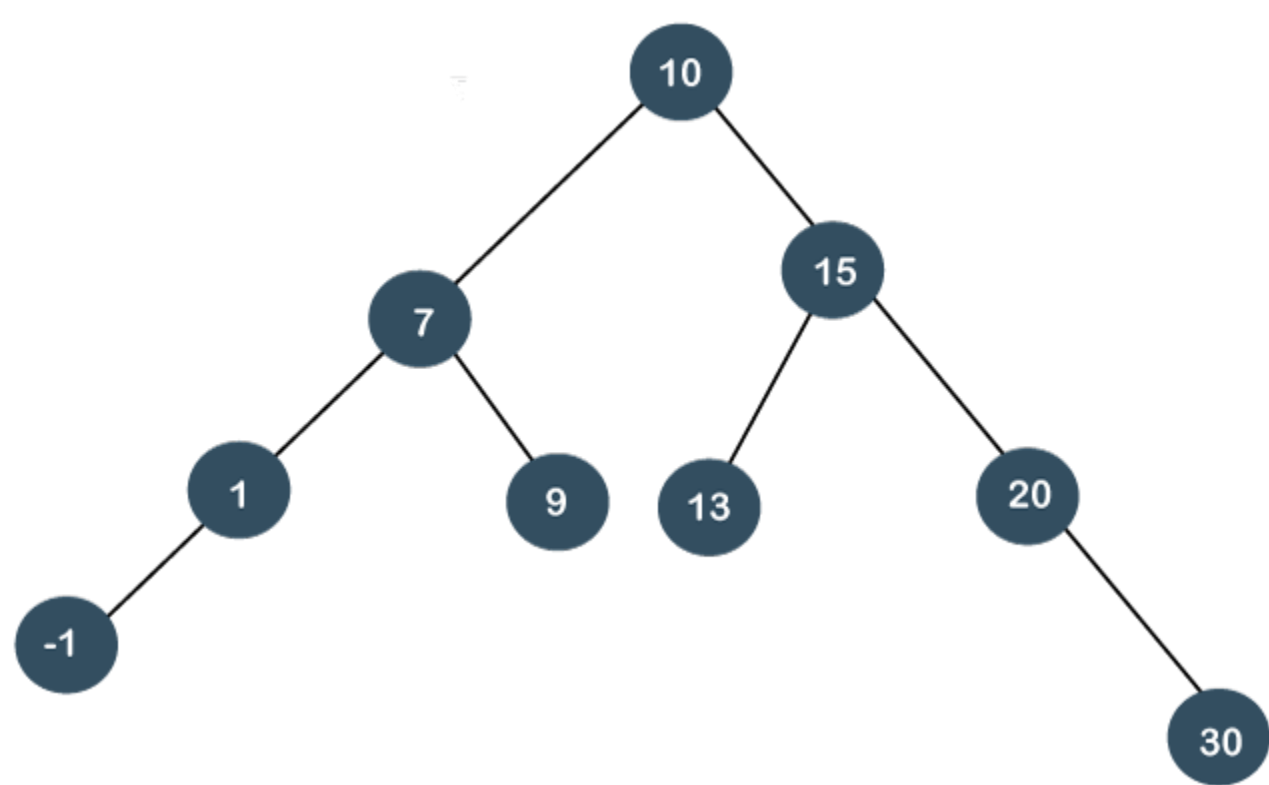
As we have observed that two left rotations are performed; so it is known as a zig zig left rotation.

- Zig zag rotations

Till now, we have read that both parent and grandparent are either in RR or LL relationship. Now, we will see the RL or LR relationship between the parent and the grandparent.

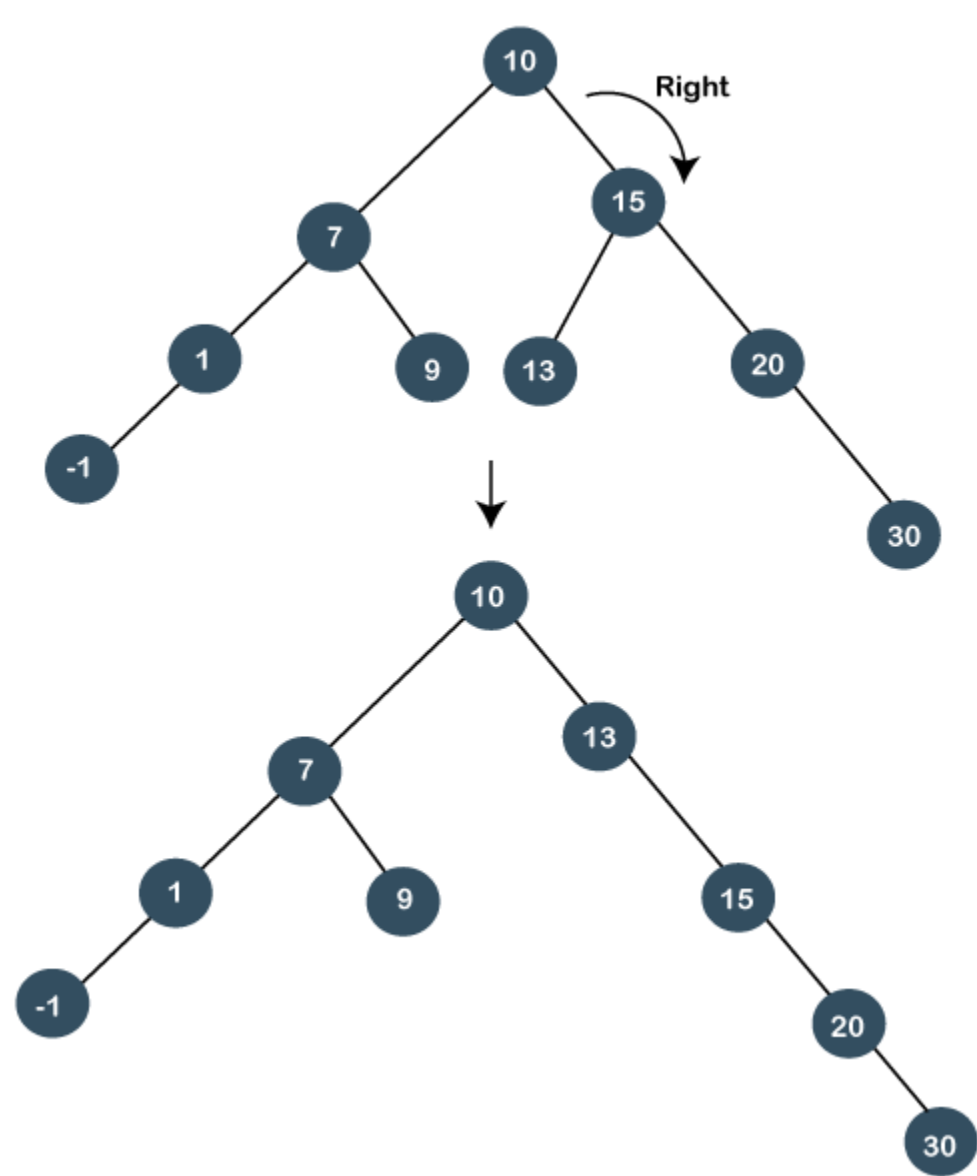
Let's understand this case through an example.

Suppose we want to search 13 element in the tree which is shown below:

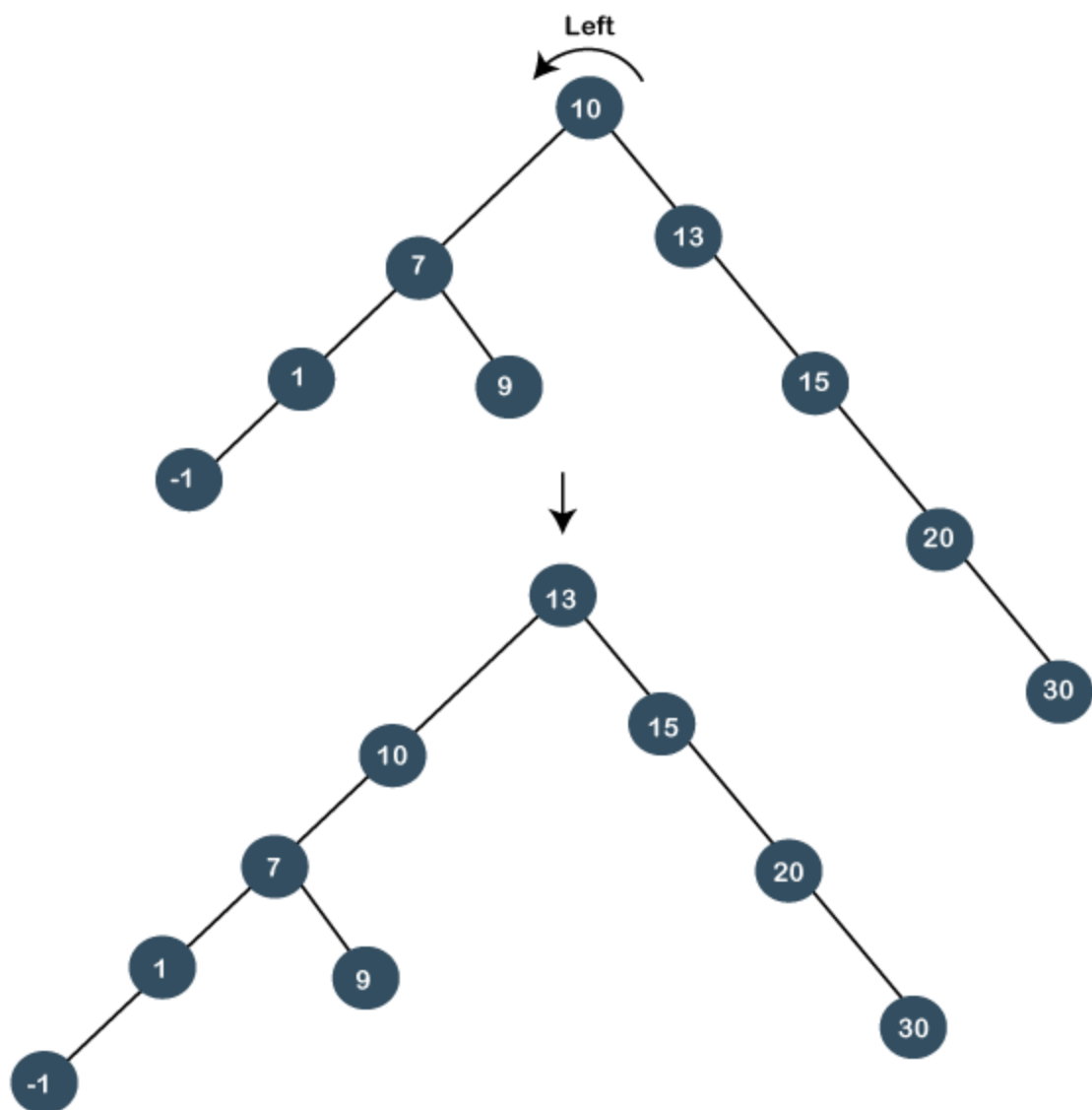


Step 1: First, we perform standard BST searching operation. As 13 is greater than 10 but less than 15, so node 13 will be the left child of node 15.

Step 2: Since node 13 is at the left of 15 and node 15 is at the right of node 10, so RL relationship exists. First, we perform the right rotation on node 15, and 15 will move downwards, and node 13 will come upwards, as shown below:



Still, node 13 is not the root node, and 13 is at the right of the root node, so we will perform left rotation known as a zag rotation. The node 10 will move downwards, and 13 becomes the root node as shown below:

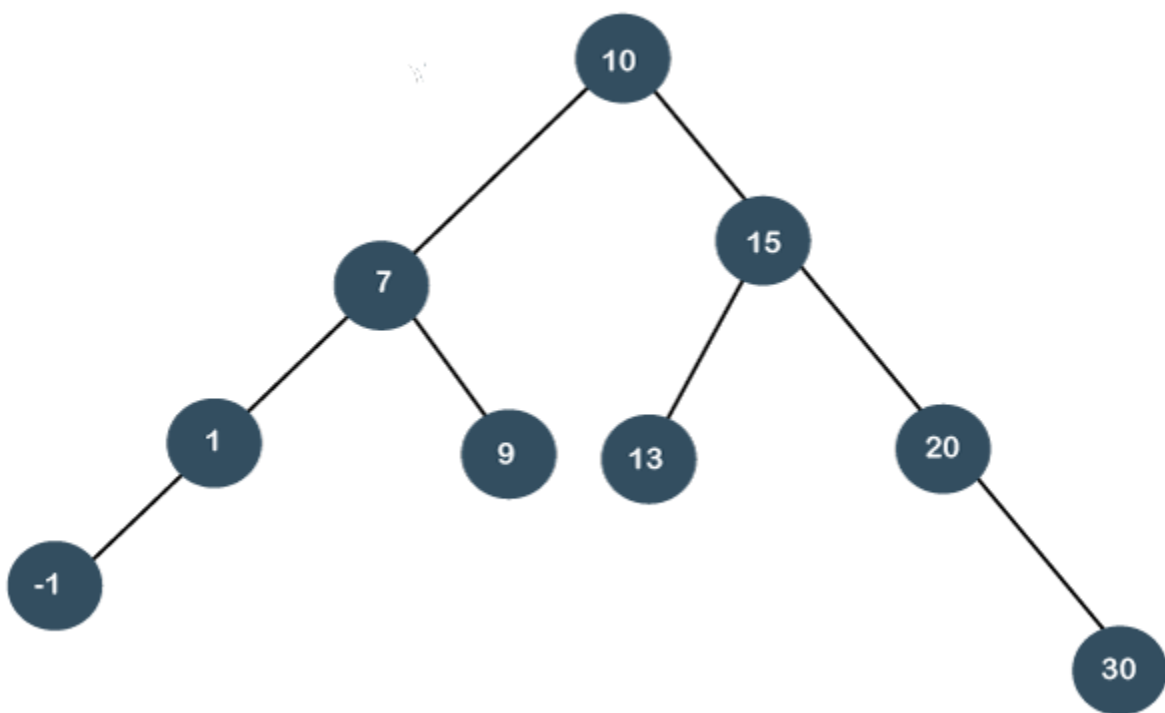


As we can observe in the above tree that node 13 has become the root node; therefore, the searching is completed. In this case, we have first performed the zig rotation and then zag rotation; so, it is known as a zig zag rotation.

- Zag zig rotation

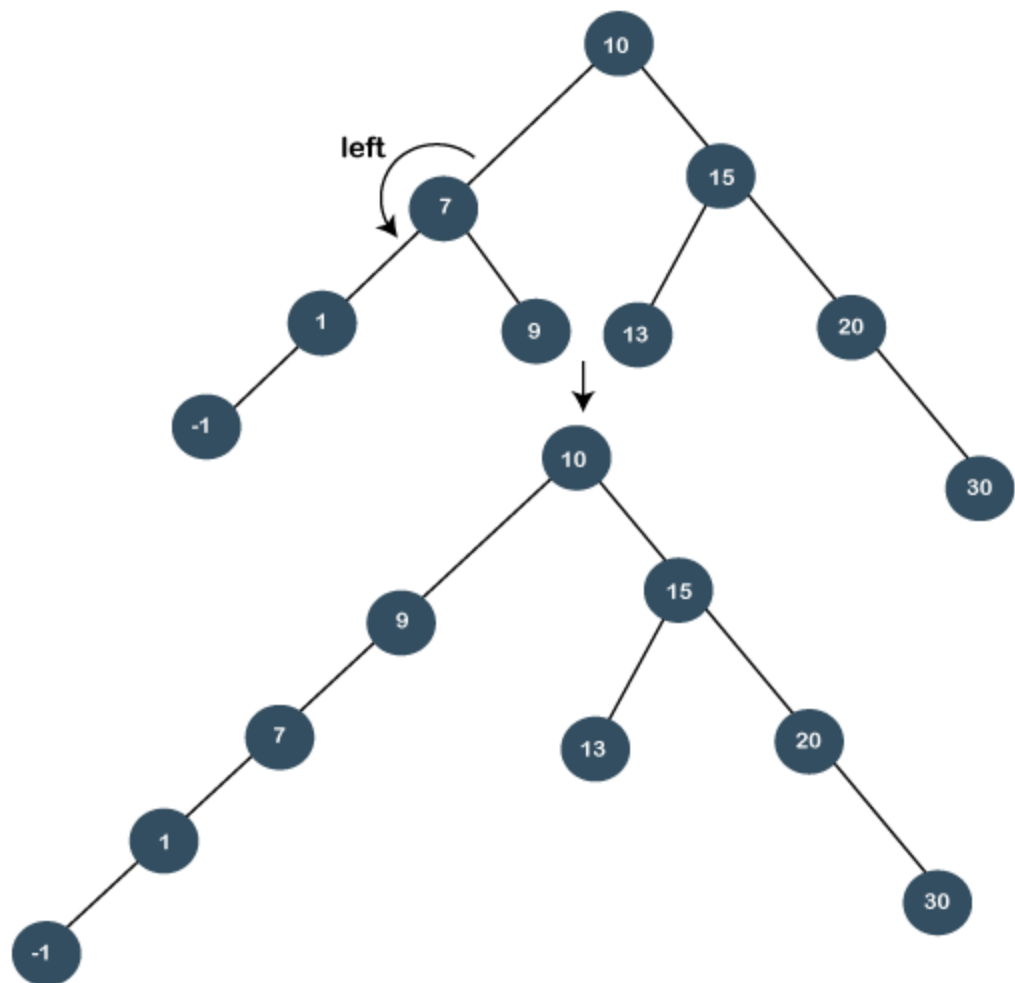
Let's understand this case through an example.

Suppose we want to search 9 element in the tree, which is shown below:

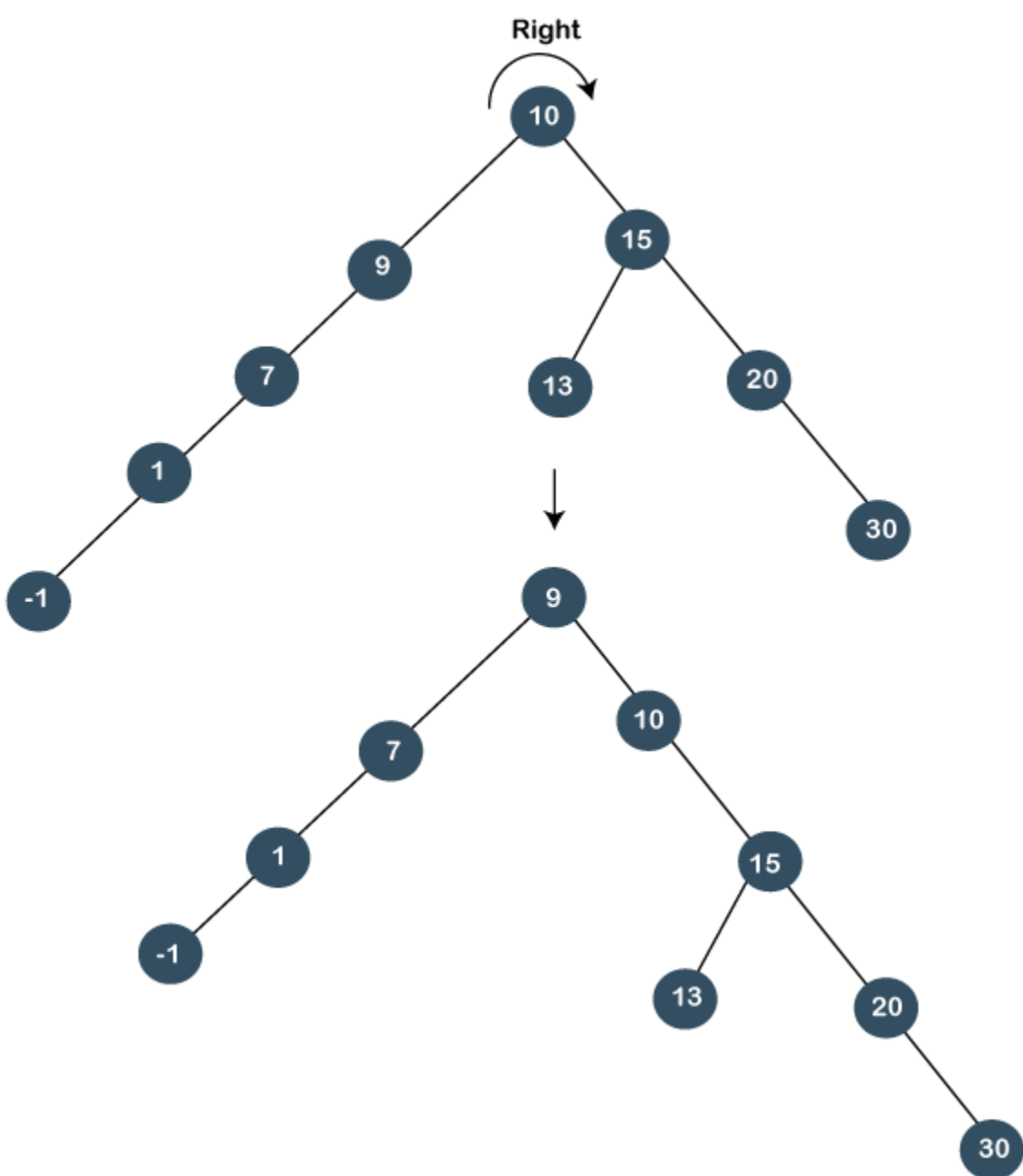


Step 1: First, we perform the standard BST searching operation. As 9 is less than 10 but greater than 7, so it will be the right child of node 7.

Step 2: Since node 9 is at the right of node 7, and node 7 is at the left of node 10, so LR relationship exists. First, we perform the left rotation on node 7. The node 7 will move downwards, and node 9 moves upwards as shown below:



Still the node 9 is not a root node, and 9 is at the left of the root node, so we will perform the right rotation known as zig rotation. After performing the right rotation, node 9 becomes the root node, as shown below:



As we can observe in the above tree that node 13 is a root node; therefore, the searching is completed. In this case, we have first performed the zag rotation (left rotation), and then zig rotation (right rotation) is performed, so it is known as a zag zig rotation.

Advantages of Splay tree

- In the splay tree, we do not need to store the extra information. In contrast, in AVL trees, we need to store the balance factor of each node that requires extra space, and Red-Black trees also require to store one extra bit of information that denotes the color of the node, either Red or Black.
- It is the fastest type of Binary Search tree for various practical applications. It is used in Windows NT and GCC compilers.

- It provides better performance as the frequently accessed nodes will move nearer to the root node, due to which the elements can be accessed quickly in splay trees. It is used in the cache implementation as the recently accessed data is stored in the cache so that we do not need to go to the memory for accessing the data, and it takes less time.

Drawback of Splay tree

The major drawback of the splay tree would be that trees are not strictly balanced, i.e., they are roughly balanced. Sometimes the splay trees are linear, so it will take $O(n)$ time complexity.

Insertion operation in Splay tree

In the **insertion** operation, we first insert the element in the tree and then perform the splaying operation on the inserted element.

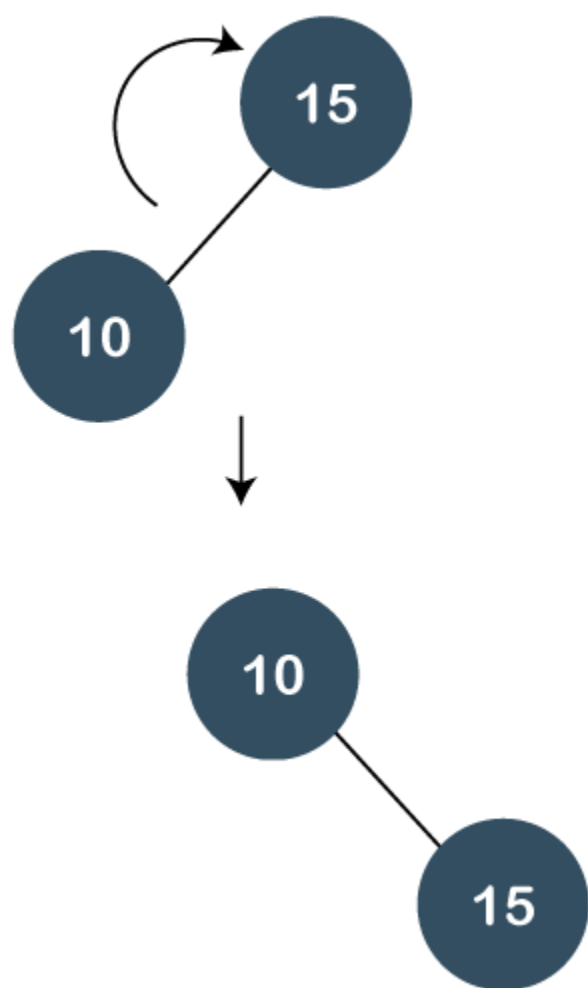
15, 10, 17, 7

Step 1: First, we insert node 15 in the tree. After insertion, we need to perform splaying. As 15 is a root node, so we do not need to perform splaying.



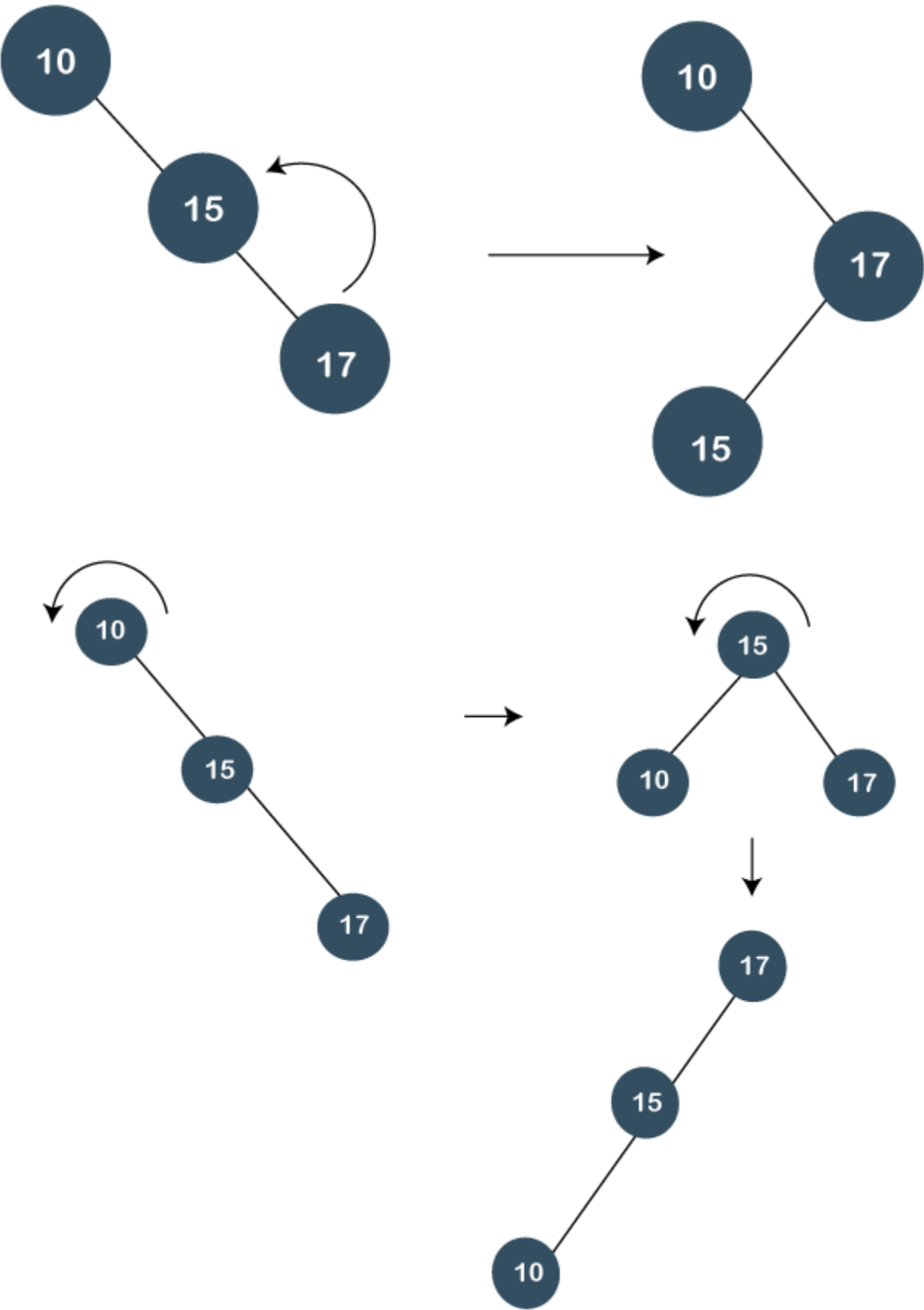
Step 2: The next element is 10. As 10 is less than 15, so node 10 will be the left child of node 15, as shown below:

Now, we perform **splaying**. To make 10 as a root node, we will perform the right rotation, as shown below:



Step 3: The next element is 17. As 17 is greater than 10 and 15 so it will become the right child of node 15.

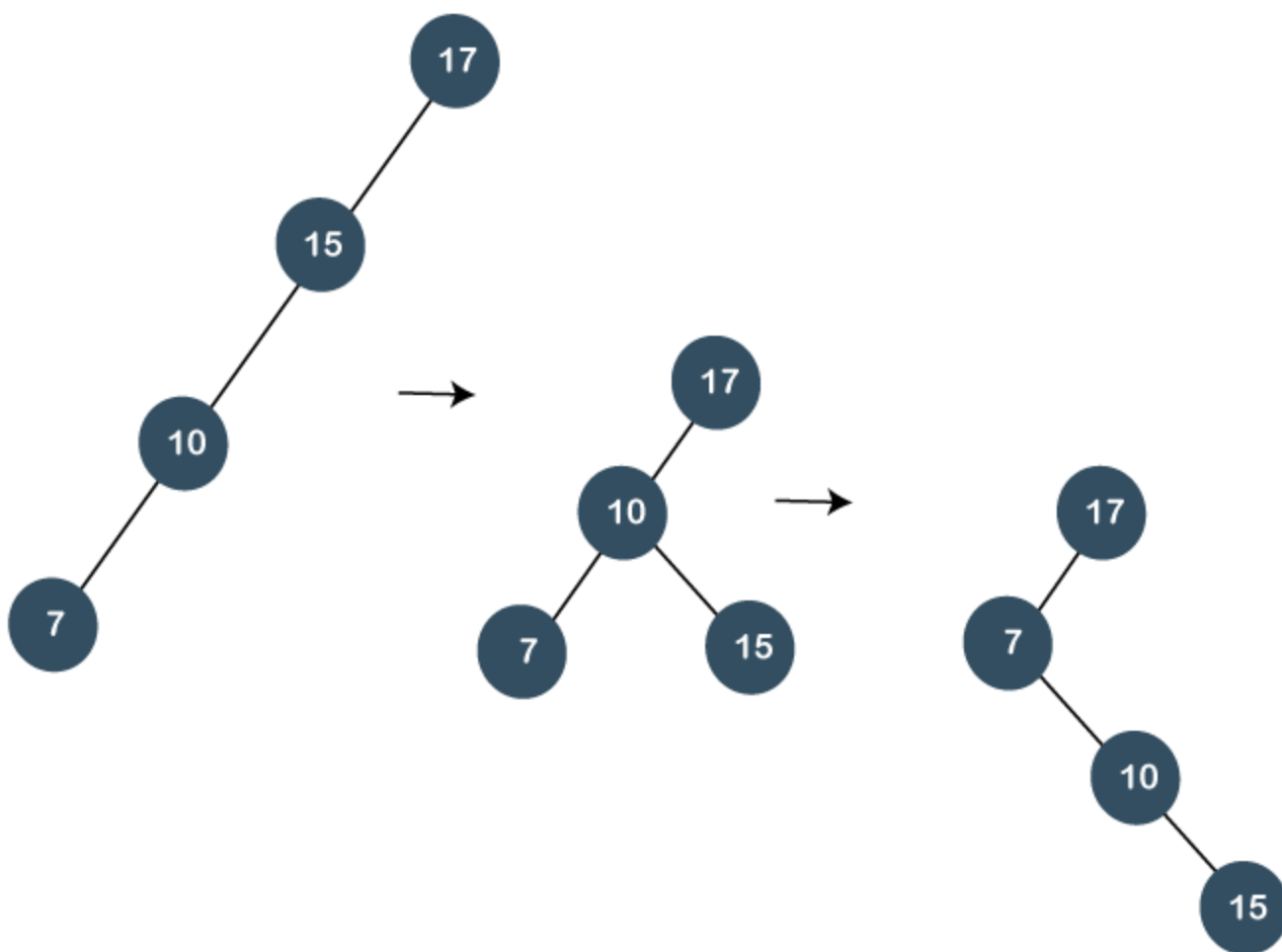
Now, we will perform splaying. As 17 is having a parent as well as a grandparent so we will perform zig zig rotations.



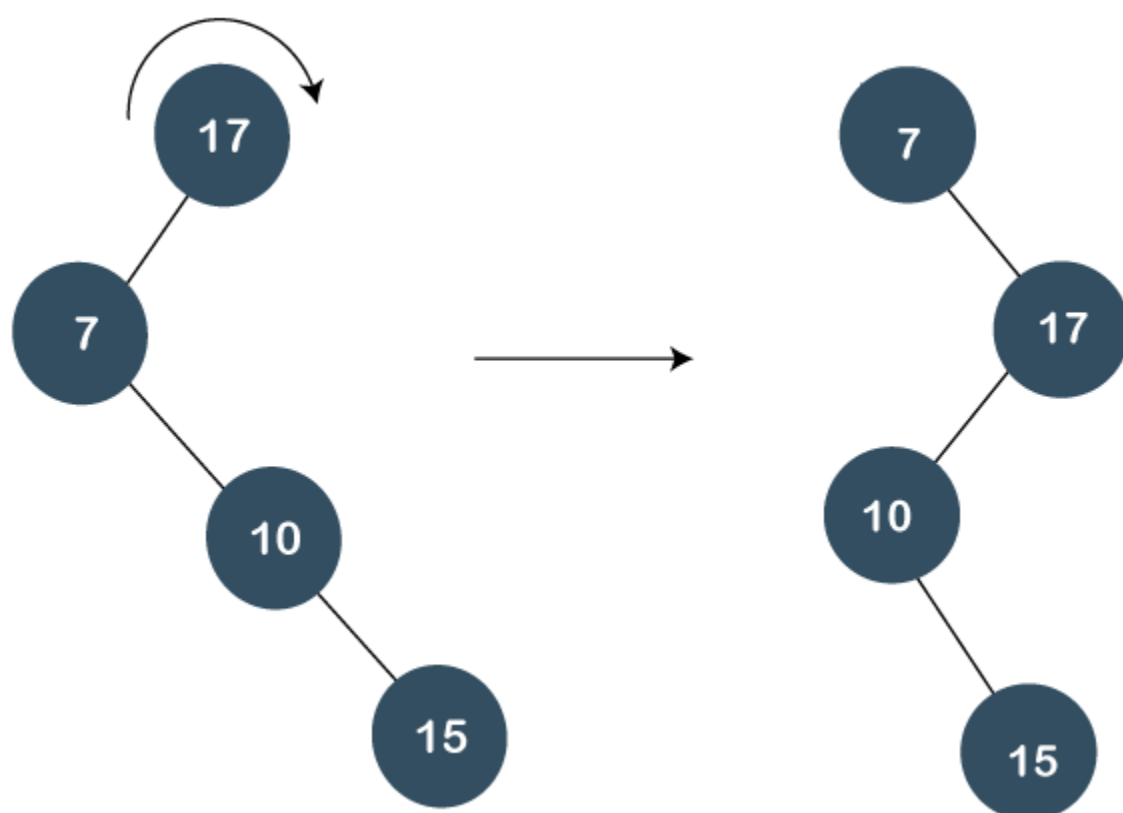
In the above figure, we can observe that 17 becomes the root node of the tree; therefore, the insertion is completed.

Step 4: The next element is 7. As 7 is less than 17, 15, and 10, so node 7 will be left child of 10.

Now, we have to splay the tree. As 7 is having a parent as well as a grandparent so we will perform two right rotations as shown below:



Still the node 7 is not a root node, it is a left child of the root node, i.e., 17. So, we need to perform one more right rotation to make node 7 as a root node as shown below:



Algorithm for Insertion operation

1. Insert(T, n)
2. temp= T_root
3. y=NULL
4. while(temp!=NULL)
5. y=temp
6. if(n->data <temp->data)
7. temp=temp->left
8. else
9. temp=temp->right
10. n.parent= y
11. if(y==NULL)
12. T_root = n
13. else if (n->data < y->data)

14. y->left = n
15. else
16. y->right = n
17. Splay(T, n)

In the above algorithm, T is the tree and n is the node which we want to insert. We have created a temp variable that contains the address of the root node. We will run the while loop until the value of temp becomes NULL.

Once the insertion is completed, splaying would be performed

Algorithm for Splaying operation

1. Splay(T, N)
2. while(n->parent !=Null)
3. if(n->parent==T->root)
4. if(n==n->parent->left)
5. right_rotation(T, n->parent)
6. else
7. left_rotation(T, n->parent)
8. else
9. p= n->parent
10. g = p->parent
11. if(n=n->parent->left && p=p->parent->left)
12. right.rotation(T, g), right.rotation(T, p)
13. else if(n=n->parent->right && p=p->parent->right)
14. left.rotation(T, g), left.rotation(T, p)
15. else if(n=n->parent->left && p=p->parent->right)
16. right.rotation(T, p), left.rotation(T, g)
17. else
18. left.rotation(T, p), right.rotation(T, g)
- 19.
20. Implementation of right.rotation(T, x)
21. right.rotation(T, x)
22. y= x->left
23. x->left=y->right
24. y->right=x
25. return y

In the above implementation, x is the node on which the rotation is performed, whereas y is the left child of the node x.

Implementation of left.rotation(T, x)

1. left.rotation(T, x)
2. y=x->right
3. x->right = y->left
4. y->left = x
5. return y

In the above implementation, x is the node on which the rotation is performed and y is the right child of the node x.

Deletion in Splay tree

As we know that splay trees are the variants of the Binary search tree, so deletion operation in the splay tree would be similar to the BST, but the only difference is that the delete operation is followed in splay trees by the splaying operation.

Types of Deletions:

There are two types of deletions in the splay trees:

1. Bottom-up splaying
2. Top-down splaying

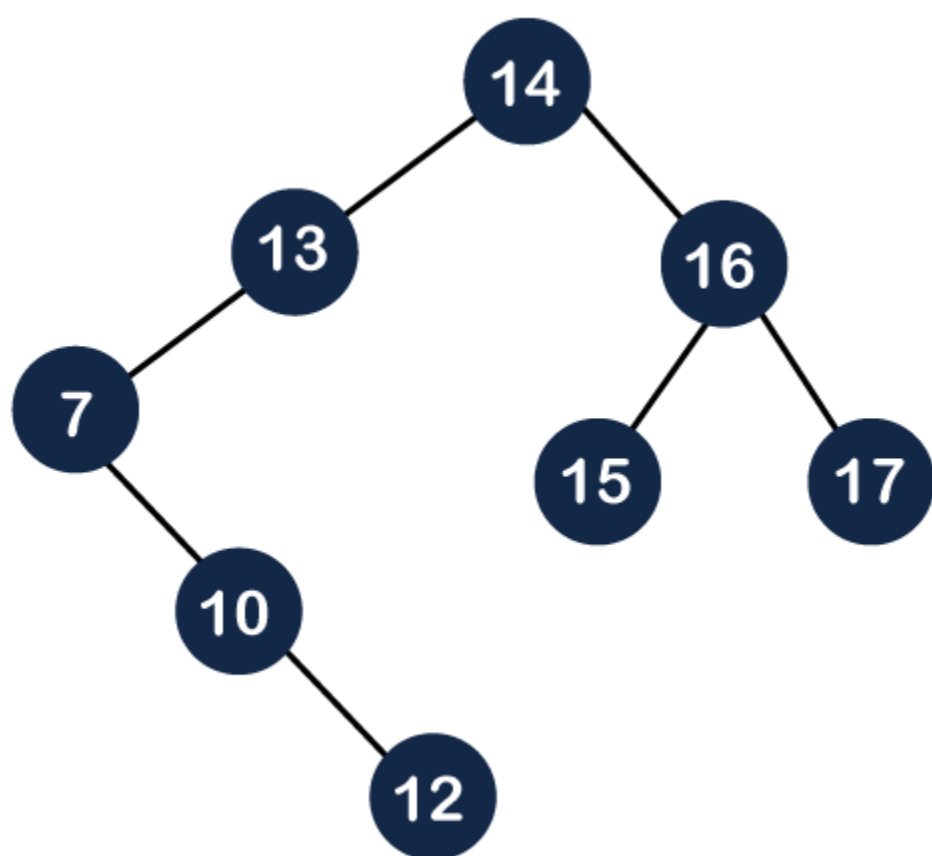
Bottom-up splaying

In bottom-up splaying, first we delete the element from the tree and then we perform the splaying on the deleted node.

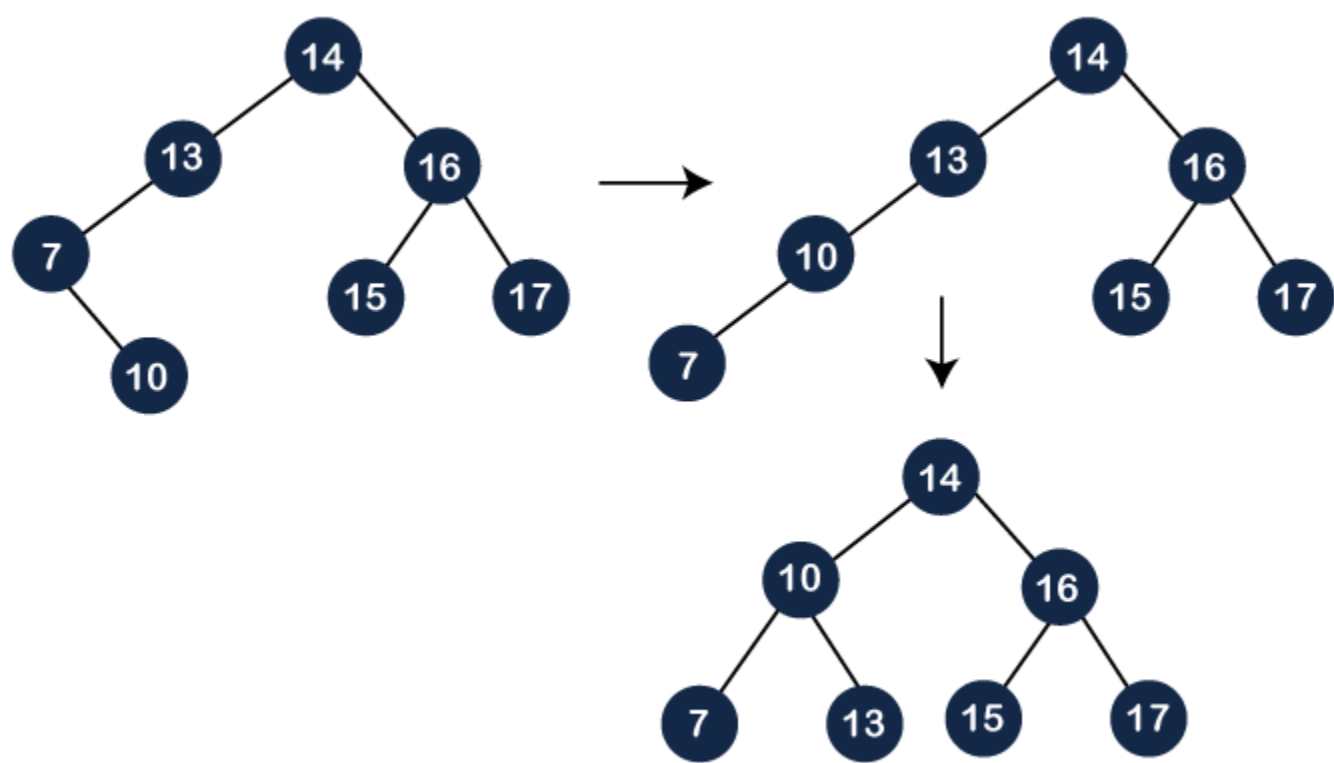
Let's understand the deletion in the Splay tree.

Suppose we want to delete 12, 14 from the tree shown below:

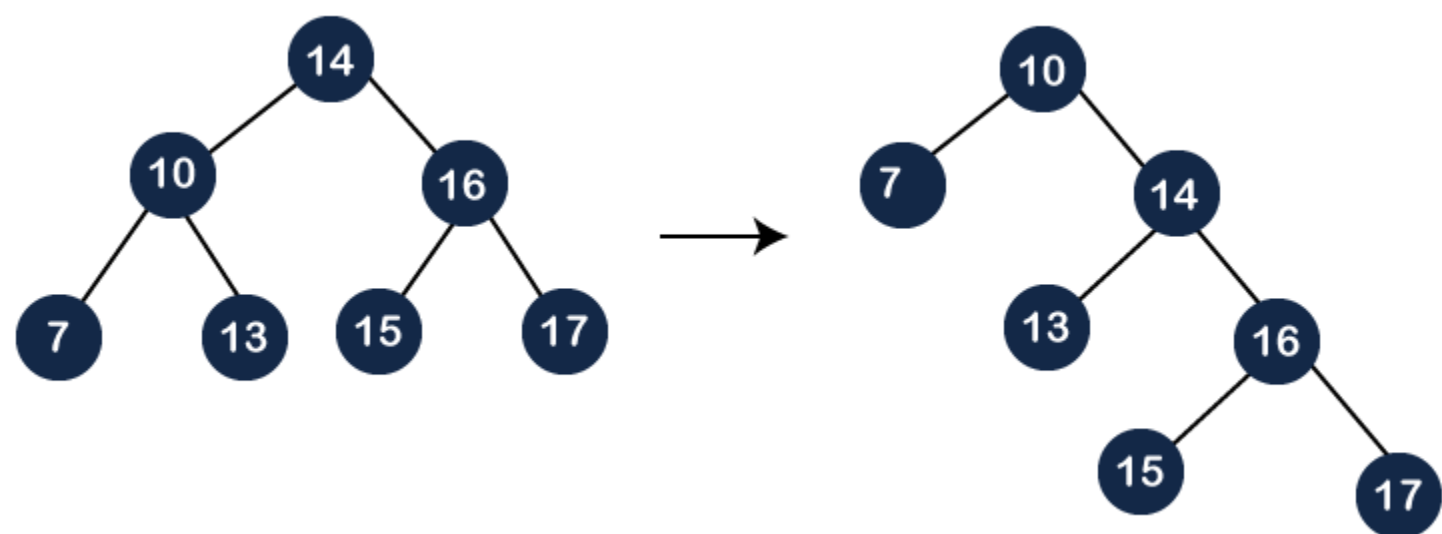
- First, we simply perform the standard BST deletion operation to delete 12 element. As 12 is a leaf node, so we simply delete the node from the tree.



The deletion is still not completed. We need to splay the parent of the deleted node, i.e., 10. We have to perform **Splay(10)** on the tree. As we can observe in the above tree that 10 is at the right of node 7, and node 7 is at the left of node 13. So, first, we perform the left rotation on node 7 and then we perform the right rotation on node 13, as shown below:

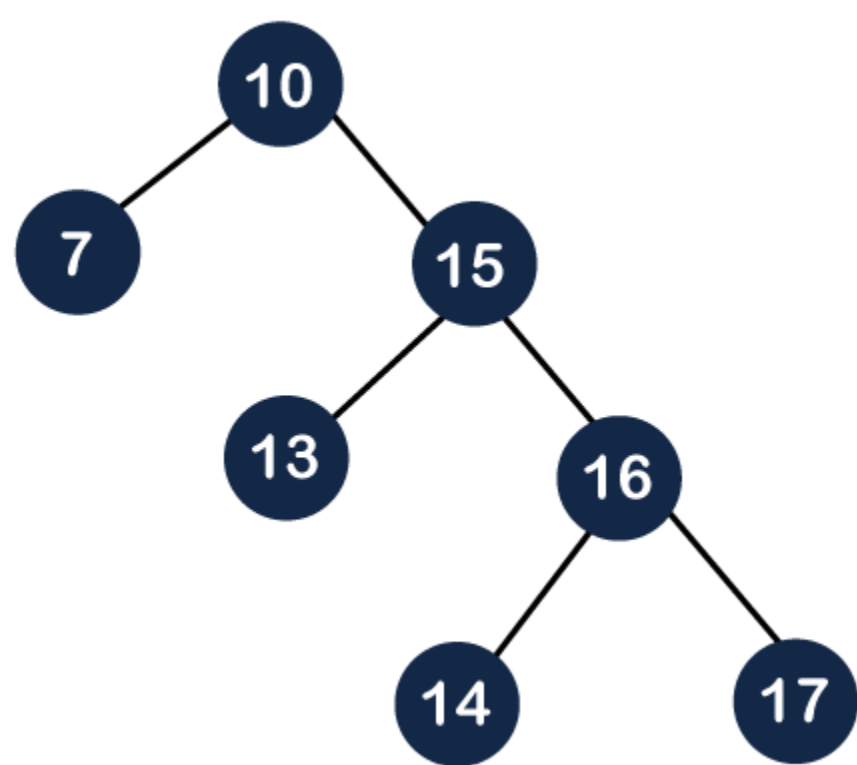


Still, node 10 is not a root node; node 10 is the left child of the root node. So, we need to perform the right rotation on the root node, i.e., 14 to make node 10 a root node as shown below:

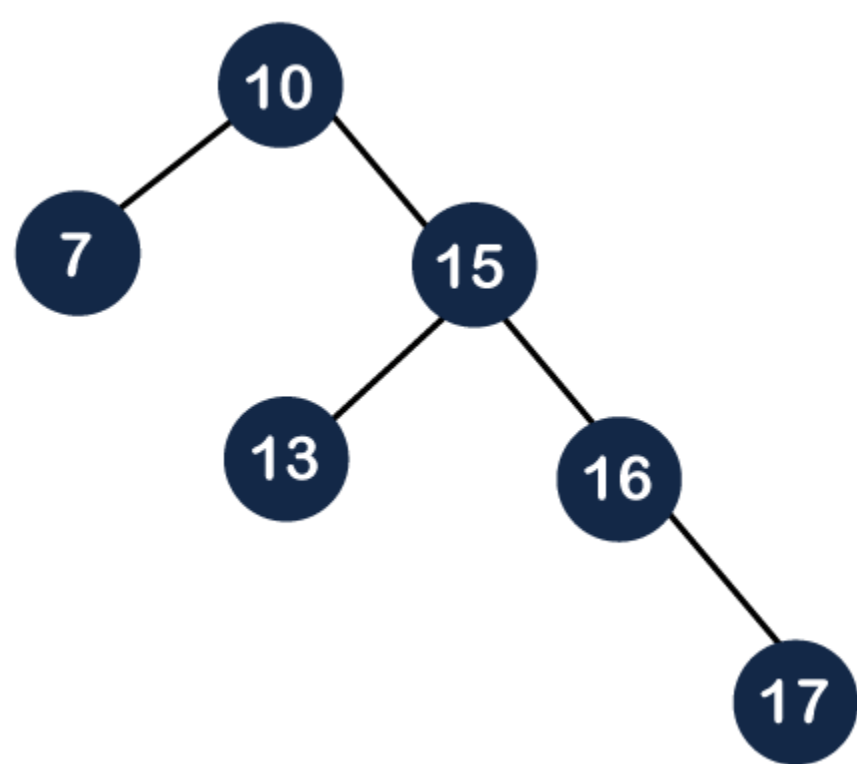


- Now, we have to delete the 14 element from the tree, which is shown below:

As we know that we cannot simply delete the internal node. We will replace the value of the node either using **inorder predecessor** or **inorder successor**. Suppose we use inorder successor in which we replace the value with the lowest value that exist in the right subtree. The lowest value in the right subtree of node 14 is 15, so we replace the value 14 with 15. Since node 14 becomes the leaf node, so we can simply delete it as shown below:



Still, the deletion is not completed. We need to perform one more operation, i.e., splaying in which we need to make the parent of the deleted node as the root node. Before deletion, the parent of node 14 was the root node, i.e., 10, so we do need to perform any splaying in this case.

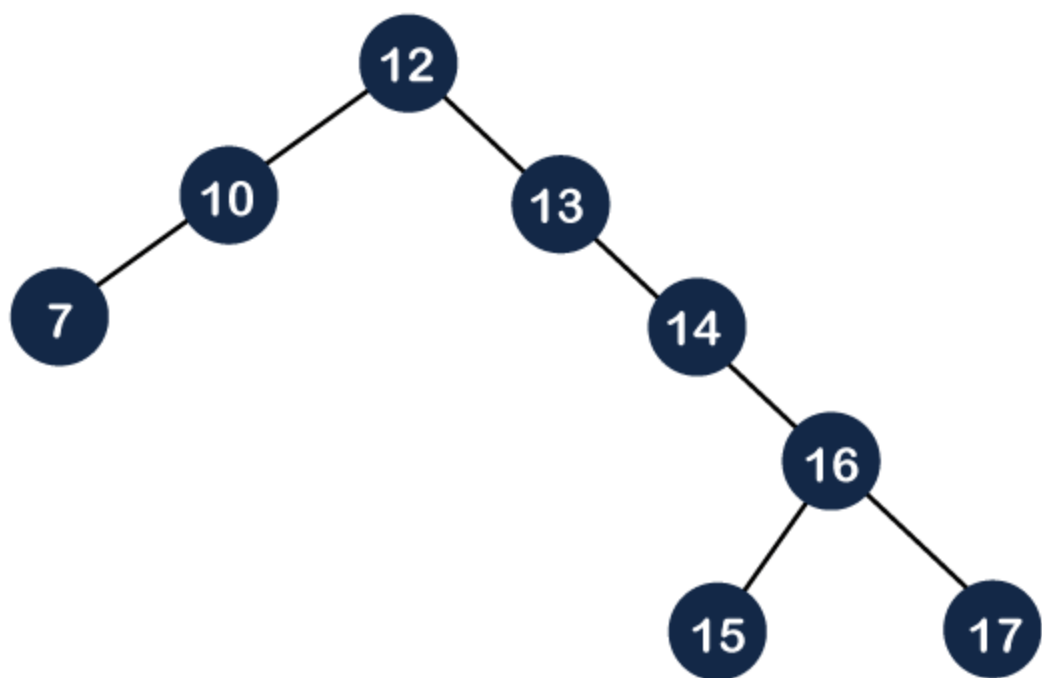


Top-down splaying

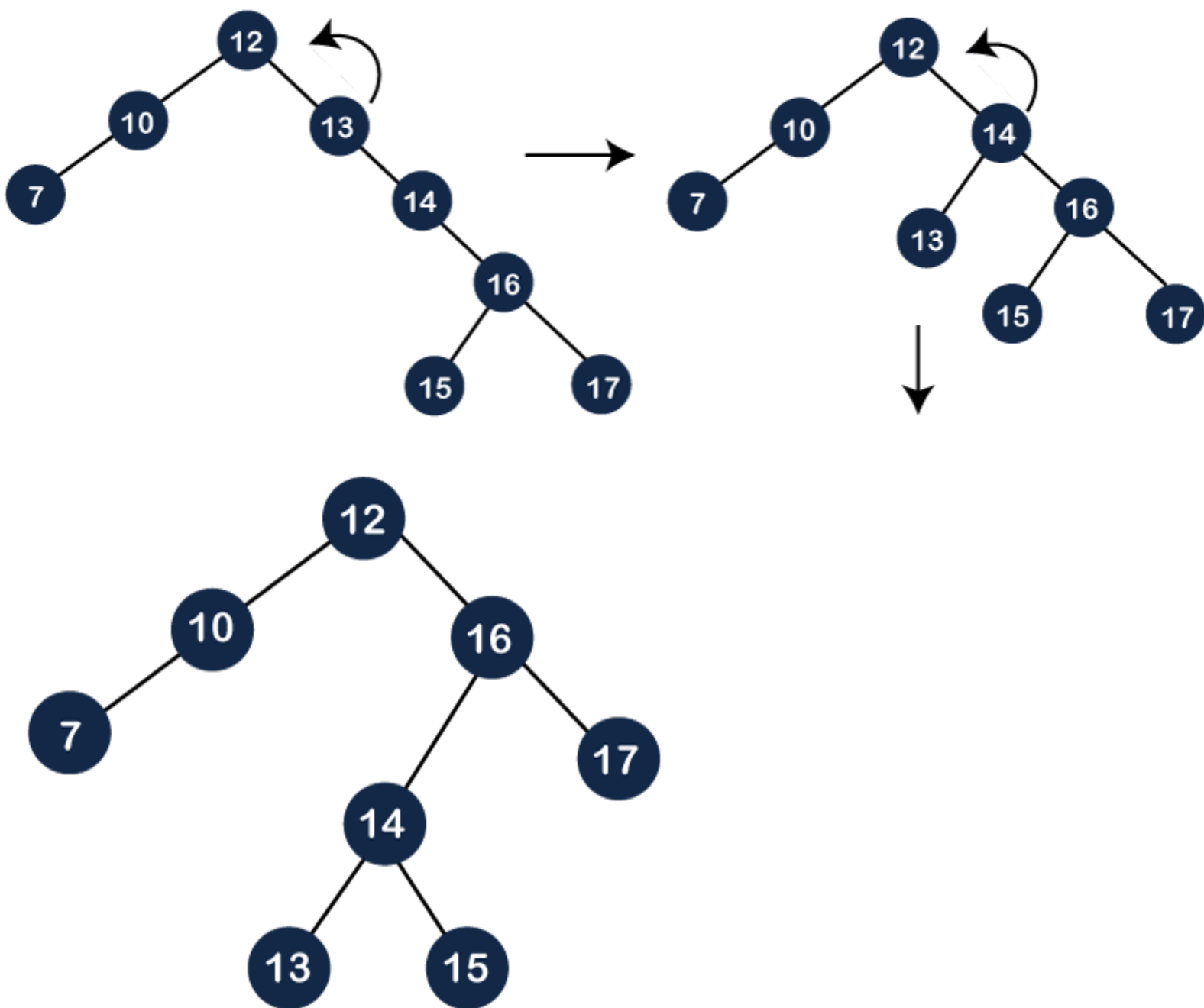
In top-down splaying, we first perform the splaying on which the deletion is to be performed and then delete the node from the tree. Once the element is deleted, we will perform the join operation.

Let's understand the top-down splaying through an example.

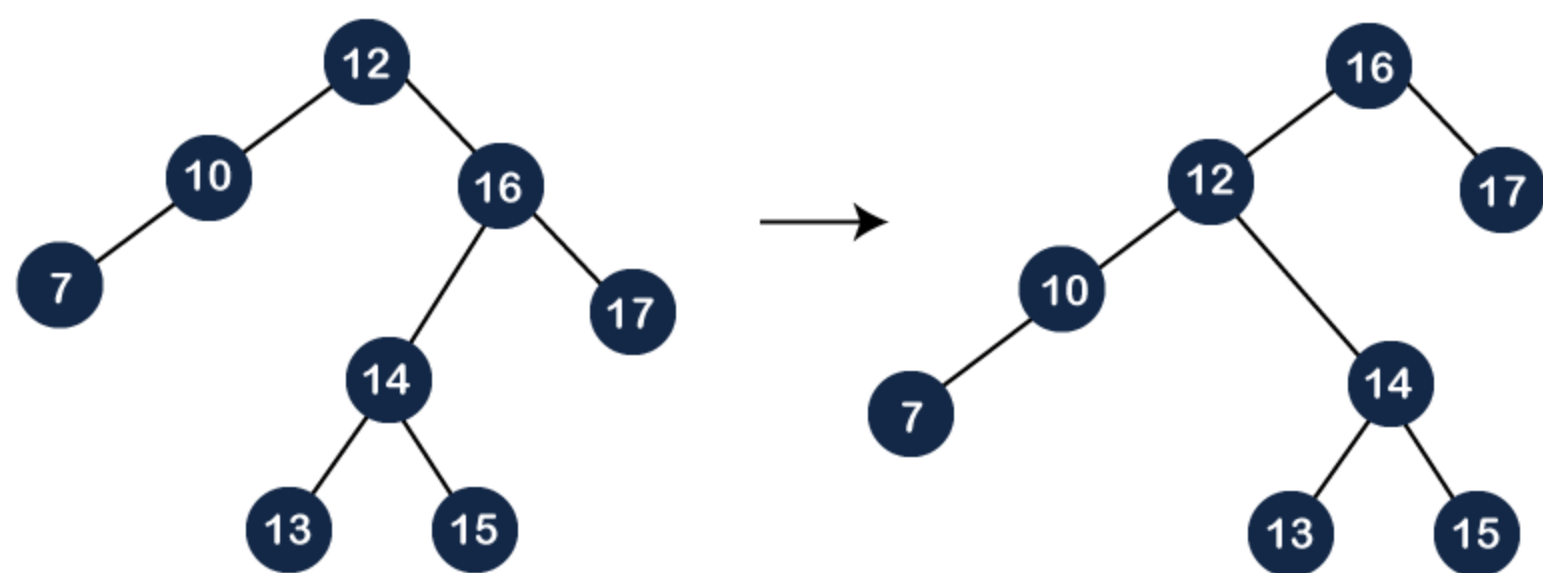
Suppose we want to delete 16 from the tree which is shown below:



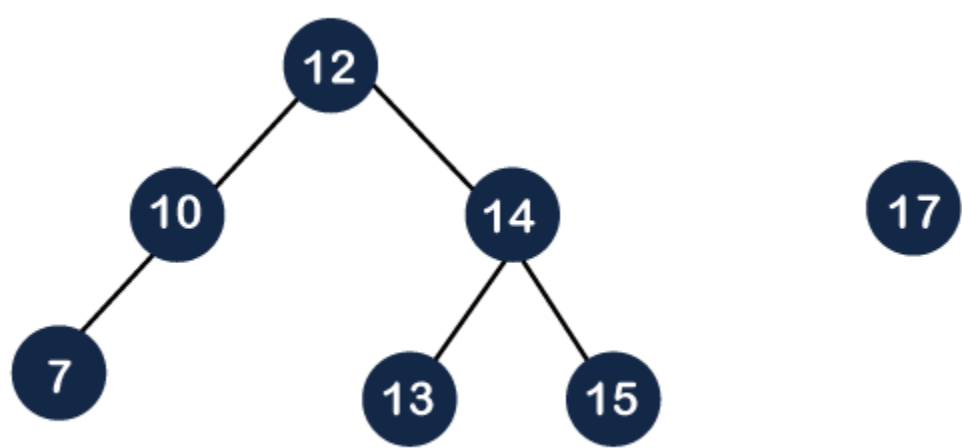
Step 1: In top-down splaying, first we perform splaying on the node 16. The node 16 has both parent as well as grandparent. The node 16 is at the right of its parent and the parent node is also at the right of its parent, so this is a zag zag situation. In this case, first, we will perform the left rotation on node 13 and then 14 as shown below:



The node 16 is still not a root node, and it is a right child of the root node, so we need to perform left rotation on the node 12 to make node 16 as a root node.

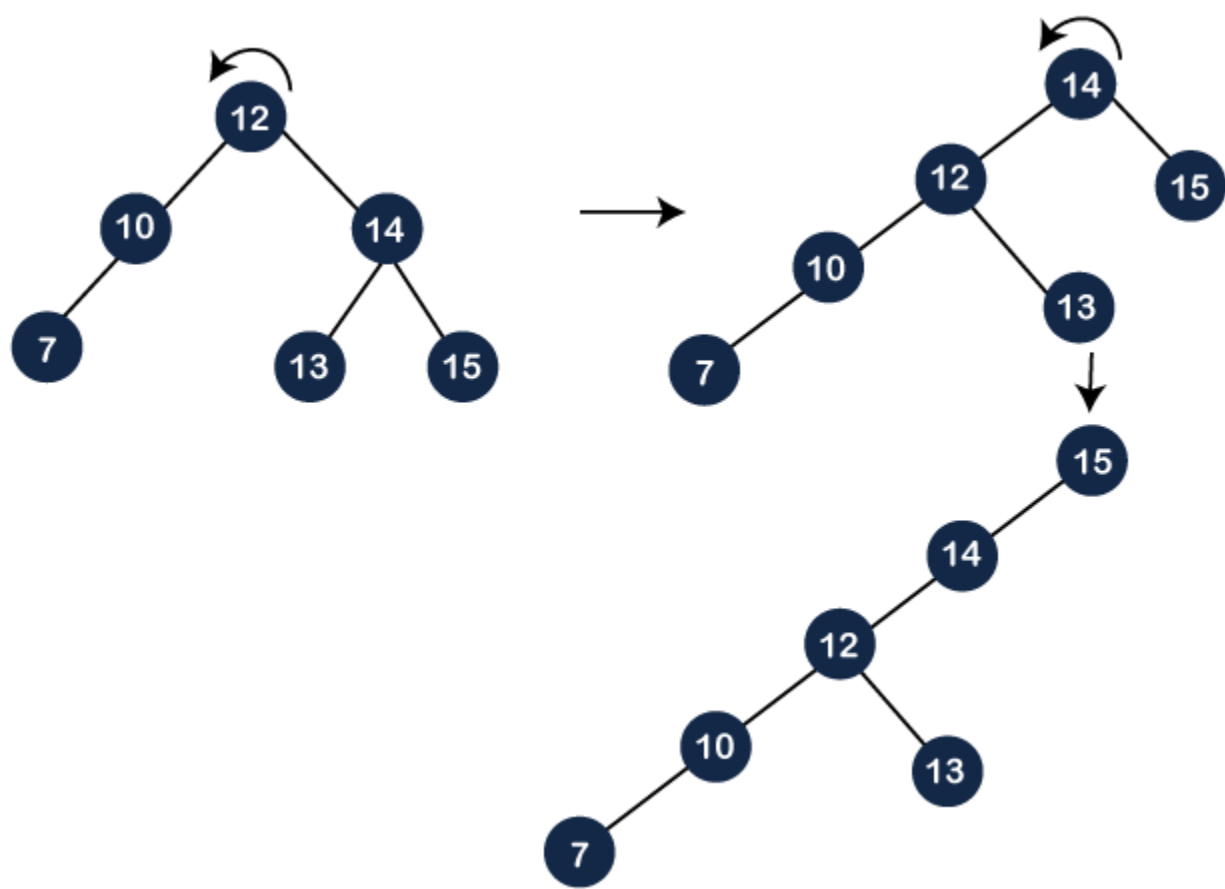


Once the node 16 becomes a root node, we will delete the node 16 and we will get two different trees, i.e., left subtree and right subtree as shown below:

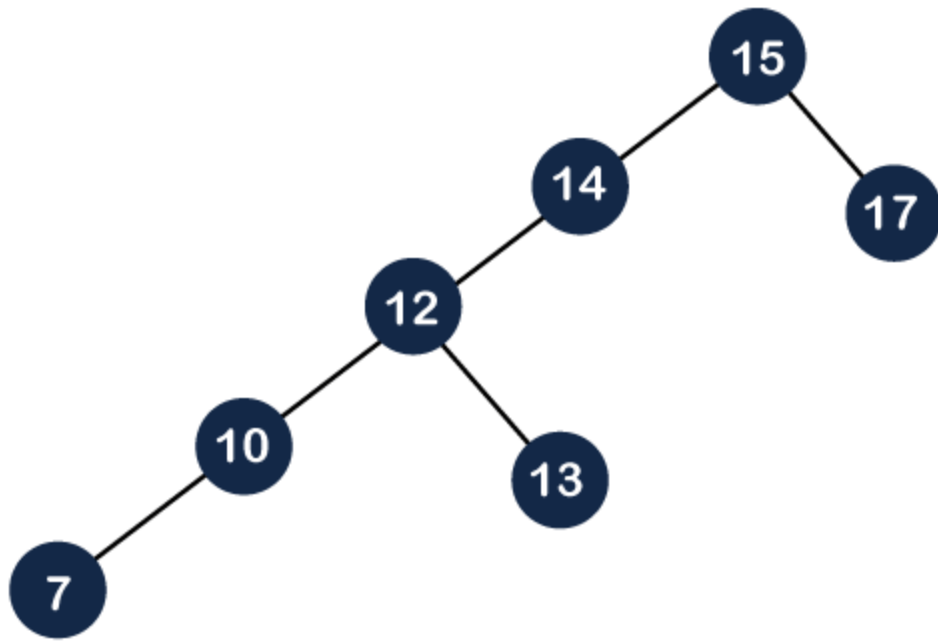


As we know that the values of the left subtree are always lesser than the values of the right subtree. The root of the left subtree is 12 and the root of the right subtree is 17. The first step is to find the maximum element in the left subtree. In the left subtree, the maximum element is 15, and then we need to perform splaying operation on 15.

As we can observe in the above tree that the element 15 is having a parent as well as a grandparent. A node is right of its parent, and the parent node is also right of its parent, so we need to perform two left rotations to make node 15 a root node as shown below:



After performing two rotations on the tree, node 15 becomes the root node. As we can see, the right child of the 15 is NULL, so we attach node 17 at the right part of the 15 as shown below, and this operation is known as a **join** operation.



Note: If the element is not present in the splay tree, which is to be deleted, then splaying would be performed. The splaying would be performed on the last accessed element before reaching the NULL.

Algorithm of Delete operation

1. If(root==NULL)
2. **return** NULL
3. Splay(root, data)
4. If data!= root->data
5. Element is not present
6. If root->left==NULL
7. root=root->right
8. **else**
9. temp=root
10. Splay(root->left, data)
11. root1->right=root->right
12. free(temp)
13. **return** root

In the above algorithm, we first check whether the root is Null or not; if the root is NULL means that the tree is empty. If the tree is not empty, we will perform the splaying operation on the element which is to be deleted. Once the splaying operation is completed, we will compare the root data with the element which is to be deleted; if both are not equal means that the element is not present in the tree. If they are equal, then the following cases can occur:

Case 1: The left of the root is NULL, the right of the root becomes the root node.

Case 2: If both left and right exist, then we splay the maximum element in the left subtree. When the splaying is completed, the maximum element becomes the root of the left subtree. The right subtree would become the right child of the root of the left subtree.

Fundamental of the Data Structure

A data structure is a specialized format of data for arranging and storing data so that any user can easily access and worked within an appropriate data to run a program efficiently. Computer memory data can be organized logically or mathematically, and this process is known as a **data structure**. In general, selecting a particular format of data depends on two factors. The data must be rich enough to satisfy the real relationships of the data in the real world. And on the other hand, the **data structure** should be so simple that one can easily process the data when it needs to be used.

Characteristics of data structures

Following are the characteristics of the data structures as follows:

1. **Linear:** A linear describes data characteristics whether the data items are arranged in sequential form like an array.
2. **Non-Linear:** A Non-Linear data structure describes the characteristics of data items that are not in sequential form like a tree, graph.
3. **Static:** It is a static data structure that describes the size and structures of a collection of data items associated with a memory location at compile time that are fixed. Example - Array.
4. **Homogenous:** It is a characteristic of data structures representing whether the data type of all elements is the same. Example- Array.
5. **Non-Homogenous:** It is a characteristic of data structures representing whether the data type elements may or may not be the same.

6. **Dynamic:** It is a dynamic data structure that defines the shrinking and expanding of data items at the run time or the program's execution. It is also related to the utilization of memory location that can be changed at the program's run time. Example: Linked list.
7. It has some rules that define how the data items are related to each other.
8. It defines some rules to display the relationship between data items and how they interact with each other.
9. It has some operations used to perform on data items like insertion, searching, deletion, etc.
10. It helps in reducing the usage of memory resources.
11. **Time Complexity:** The execution time of a program in a data structure should be minimum as possible.
12. **Space Complexity:** The memory usage through all data items in a data structure should be less possible.

Basic Operations of Data Structures

Some basic functions are chosen based on all types of data that occur in a data structure.

1. **Traversing:** It is used to visit each variable once. It is also referred to as visiting elements in the data structure.
2. **Searching:** It is used to find the location of a given element in the whole data structure. Example, an array.
3. **Insertion:** It is used to insert an element at the specified position in data elements.
4. **Deletion:** A deletion operation is used to delete an element from the specified location.
5. **Sorting:** It is used to arrange or sort the data elements in ascending or descending order. However, the data items are arranged in some logical order, such as Name key, account number, etc.
6. **Merging:** The Merge operation is used to join two or more sorted data elements to a data structure.

Needs of the data structures

Data is a basic fact or entity information that is used to calculate or manipulate. Two types of data are used in a data structure, such as numerical data and alphanumeric data. These data structures specify the nature of the data item undergoing some function. Integers and floating points are types of numeric data types. The data can be single or a set of values that are organized in a particular fashion. The data organization leads to memory that creates a logical relationship between data items and makes it necessary to use data structures.

Advantages of Data Structures

There are some advantages of data structure:

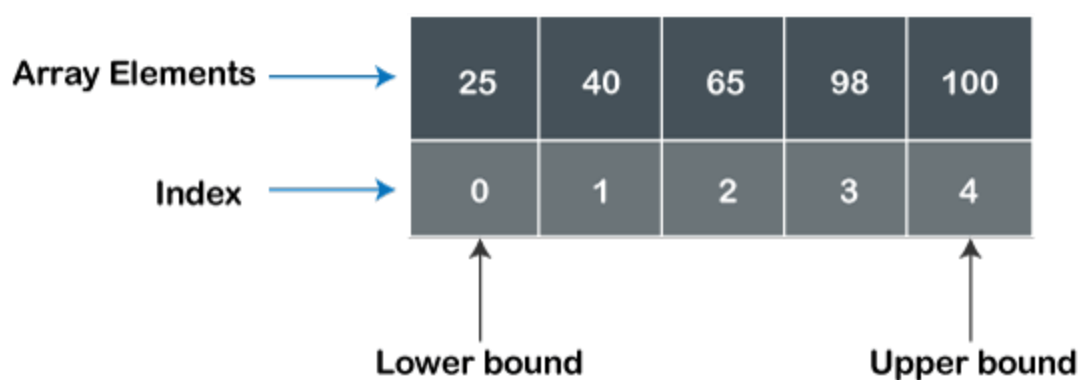
1. **Efficiency:** The efficiency and organization of a program depend on the selection of the right data structures. Suppose we want to search for a particular item from a collection of data records. In that case, our data should be organized in **linear** like an array that helps to perform the sequential search, such as element by element. However, it is efficient but more **time** consuming because we need to arrange all elements and then search each element. Hence, we choose a better option in a data structure, making the search process more efficient, like a binary search tree, selection or hash tables.
2. **Reusability:** In the data structure, many operations make the programs reusable. For example, when we write programs or implement a particular data structure, we can use it from any source location or place to get the same results.
3. **Abstraction:** The data structure is maintained by the ADT, which provides different levels of abstraction. The client can interact with data structures only through the interface.
4. The data structure helps to simplify the process of collection of data through the software systems.
5. It is used to save collection data storage on a computer that can be used by various programs.

Disadvantages of data structures

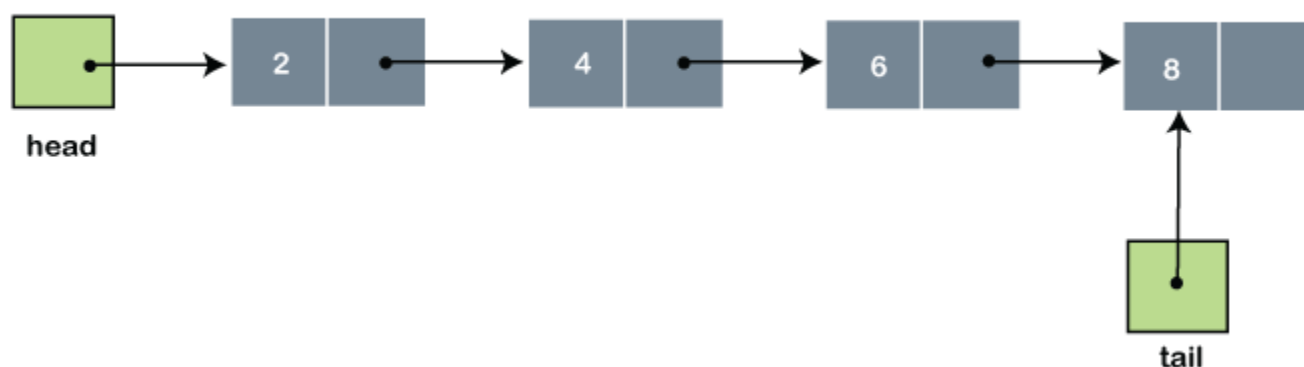
1. A user who has deep knowledge about the functionality of the data structure can make changes to it.
2. If there is an error in the data structure, an expert can detect the bug; The original user cannot help themselves solve the problem and fix it.

Following are the data structures and their uses are as follows:

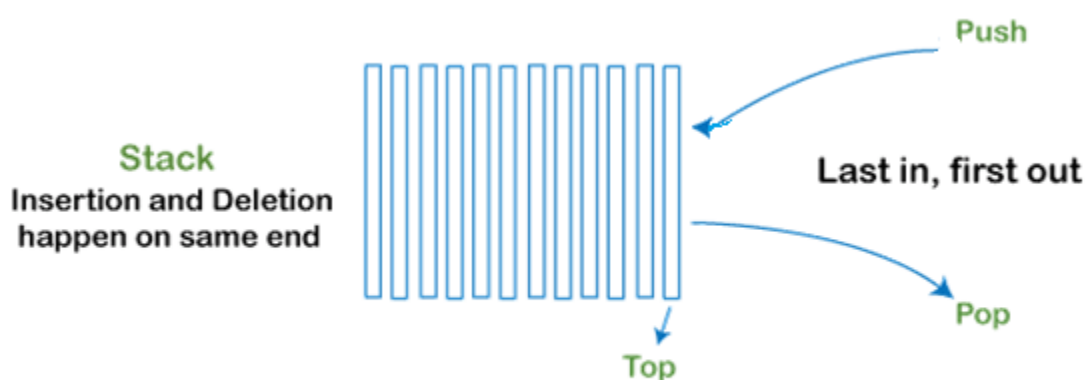
Array: An array data structure is a collection of elements of the same data type used to store data in contiguous memory locations. It has a fixed size collection of data elements that cannot be changed at runtime of a program. It is mostly used in a computer program to organize the data so that related items or values can be easily searched or sorted in a system.



Linked List: It is a collection of data links called nodes where each node contains a data value and the address of the next link. All elements of the linked list are not stored in neighbouring memory locations. In simple words, it is a sequence of data nodes that connect through links. **Each** node of a list consists of two items: the basic idea behind the uses of a linked list in a data structure is that each node within the structure includes a data part where we store values, and a pointer indicates the next node can be found. The linked list's starting point denotes the **head** of the list, and the endpoints represent the node's **tail**.



Stack: It is a linear data structure that enables the elements to be inserted and deleted from one end, called the **Top of Stack** (TOS). A stack data structure follows the last in first out (LIFO) operation to insert and remove an element from the stack list. It is an abstract data type with two possible operations to insert and delete elements in the stack: **push** and **pop**. A **push** operation is used in the stack to insert elements at the top of the list, hide those elements that already available in the list, or initialize the stack if it is empty. The **pop** operation is used in the stack data structure to delete a data item from the top of the list.

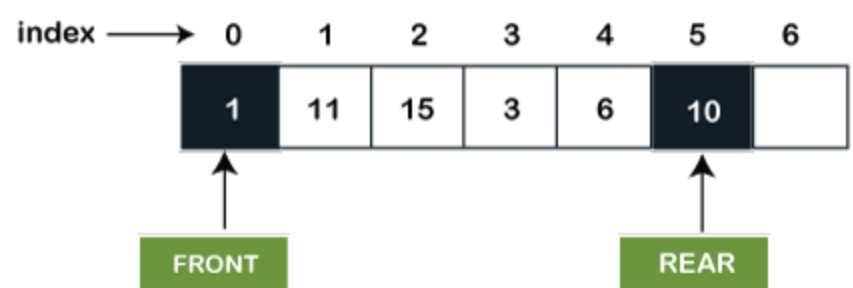


Queue: It is a linear data structure that enables the insertion at one end of the list, known as the **rear**, and the deletion of the elements occurred at another end of the list, called the **front**. It is a sequential collection of data elements based on the **First in First out** (FIFO) data structure, which means the first inserted elements in a queue will be removed first from the queue list.

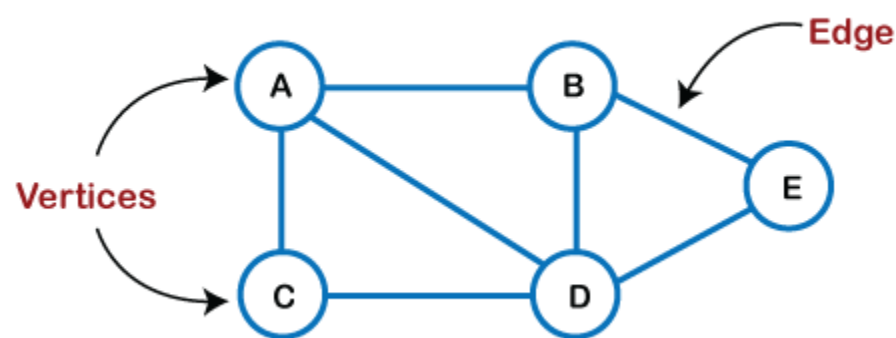
These queue operations are described below:

1. **Enqueue():** It is a queue operation used to insert an element to the list.
2. **Dequeue():** It is a queue operation used to delete an item from the list.
3. **Peek():** It is used to get the first element of the queue list without removing it.
4. **IsFull():** It is an IsFull operation that indicates whether the list is full.
5. **IsEmpty():** It is an IsEmpty operation that represents whether the list is empty.

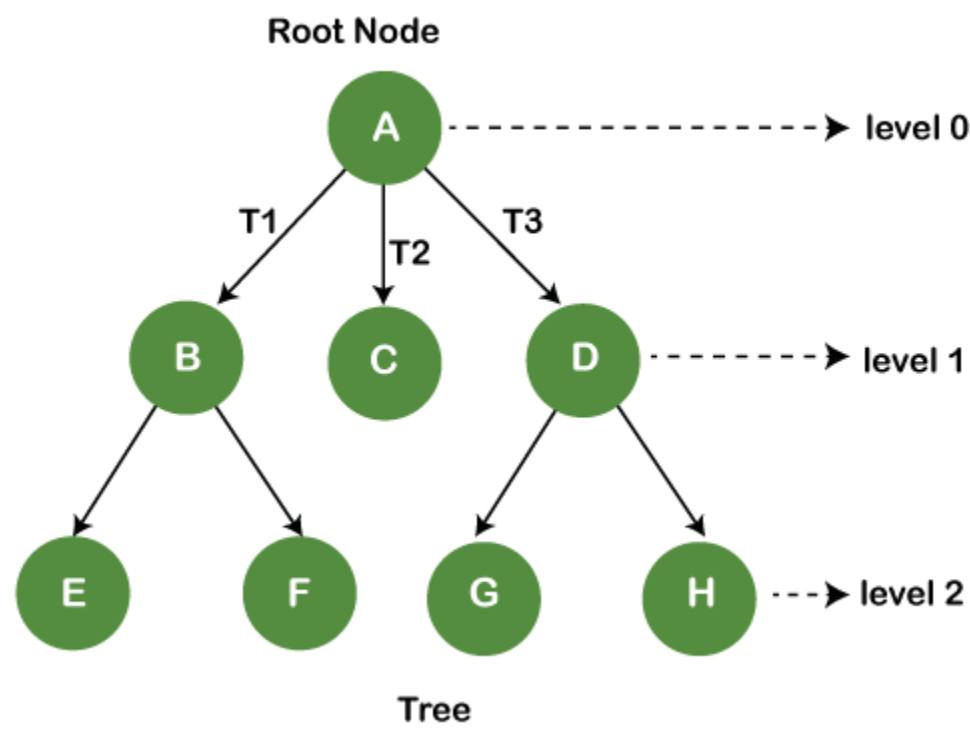
Queue in Data Structure



Graphs: A graph is a non- linear data structure consisting of finite sets of vertices (**nodes**) and **edges** to create an illustrated representation of a set of objects. These edges and nodes are connecting through any two nodes in the graph. The connected node can be represented as a directed or undirected graph. In a **directed** graph, each node is directly connected with edges in only one direction. In an **undirected** graph, each node is connected with edges in all directions. Hence it is also known as a bidirectional node.

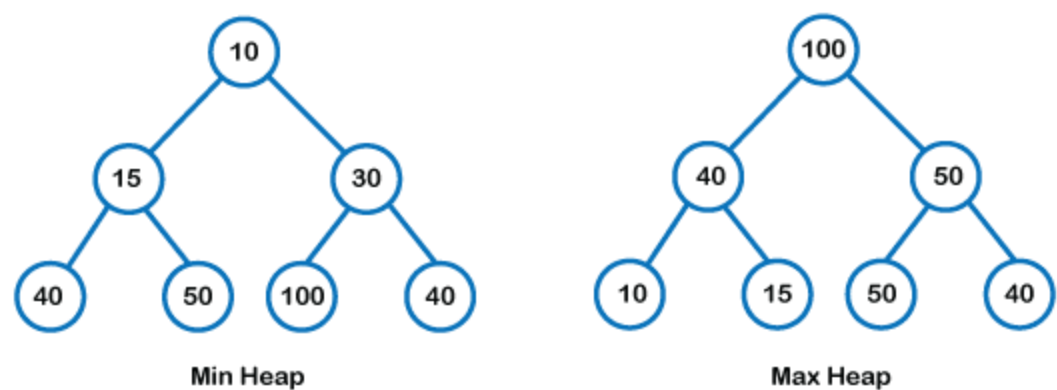


Trees: It is a type of **non-linear** data structure representing the **hierarchical** data. It is a finite set of elements where one of these nodes or elements is called a **root** node, and the remaining elements of a data structure consisting of a value called **Subtrees**. Every node of the tree maintains the parent-child relationship, where only one **parent** node and the remaining node in the tree is the **child** node. A node can have multiple child nodes but has a single parent node. There are some types of trees, such as a binary tree, binary search tree, expression trees, **AVL tree** and **B trees**.

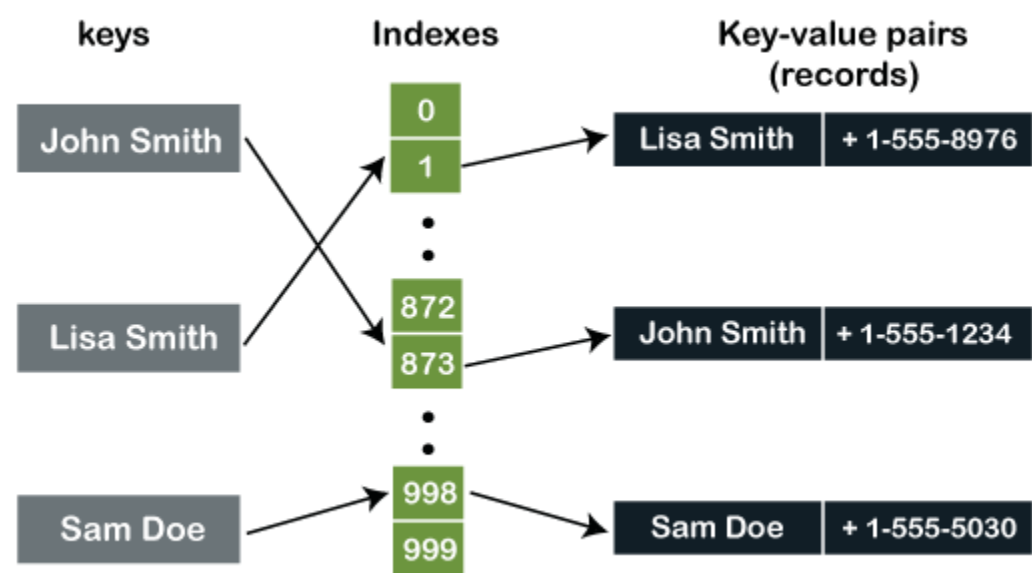


Heap: A **heap** data structure is a special type of complete binary tree that satisfies the heap property and arranged the elements in a specific order. **Max** and **Min** heap are the types of heap data structures. In Max heap, the root node's value is always higher or equal to the value of all existing child nodes in the heap tree. While in Min heap, the value of the root node/element is always shorter than the existing elements of the heap node. Each child node's value in the min-heap is equal to or larger than the parent node's value.

Heap Data Structure



Hash Table: It is a non-linear type of data structure that holds and organizes data in key-value pairs to access the particular keys/data elements. A key is a null value that is mapped or linked to an element. Hashing makes our data structure much simpler and faster when performing insertion and search operations on various data elements, regardless of the data's size.



Dictionary: A dictionary is a type of data structure that holds data elements in a group of objects and is similar to a hash table, except that it is an ordered or unordered collection of data elements in key-value pairs. Each key is associated with a single value. When we retrieve a key, the dictionary will return the associated value of a key.

For example, students = {'James' = 25, 'Jasmine' = '17', 'Rosy' = '19', 'Margret' = '24', 'Price' = '28'}

Hash Table

Hash table is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster.

A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

For example, suppose the key value is John and the value is the phone number, so when we pass the key value in the hash function shown as below:

Hash(key)= index;

When we pass the key in the hash function, then it gives the index.

Hash(john) = 3;

The above example adds the john at the index 3.

Drawback of Hash function

A Hash function assigns each value with a unique key. Sometimes hash table uses an imperfect hash function that causes a collision because the hash function generates the same key of two different values.

Hashing

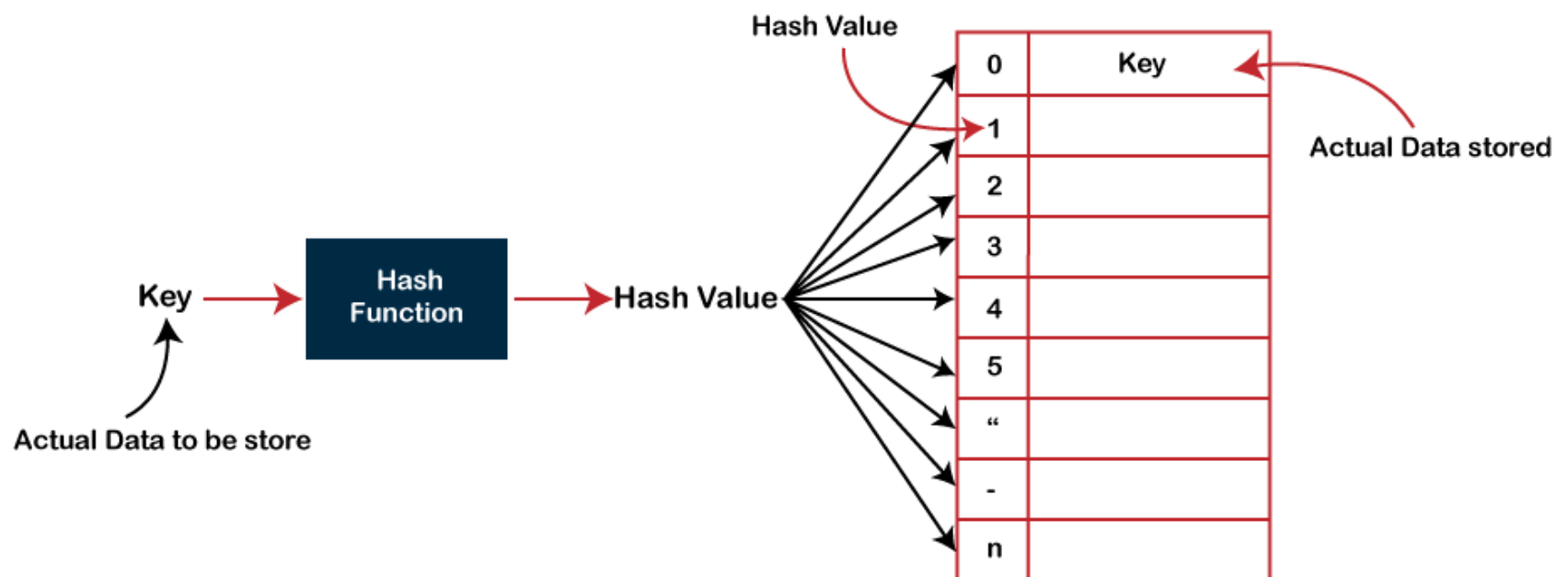
Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is O(1). Till now, we read the two techniques for searching, i.e., [linear search](#) and [binary search](#). The worst time complexity in linear search is O(n),

and $O(\log n)$ in binary search. In both the searching techniques, the searching depends upon the number of elements but we want the technique that takes a constant time. So, hashing technique came that provides a constant time.

In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as:

$$\text{Index} = \text{hash}(\text{key})$$



There are three ways of calculating the hash function:

- Division method
- Folding method
- Mid square method

In the division method, the hash function can be defined as:

$$h(k_i) = k_i \% m;$$

where **m** is the size of the hash table.

For example, if the key value is 6 and the size of the hash table is 10. When we apply the hash function to key 6 then the index would be:

$$h(6) = 6 \% 10 = 6$$

The index is 6 at which the value is stored.

Collision

When the two different values have the same value, then the problem occurs between the two values, known as a collision. In the above example, the value is stored at index 6. If the key value is 26, then the index would be:

$$h(26) = 26 \% 10 = 6$$

Therefore, two values are stored at the same index, i.e., 6, and this leads to the collision problem. To resolve these collisions, we have some techniques known as collision techniques.

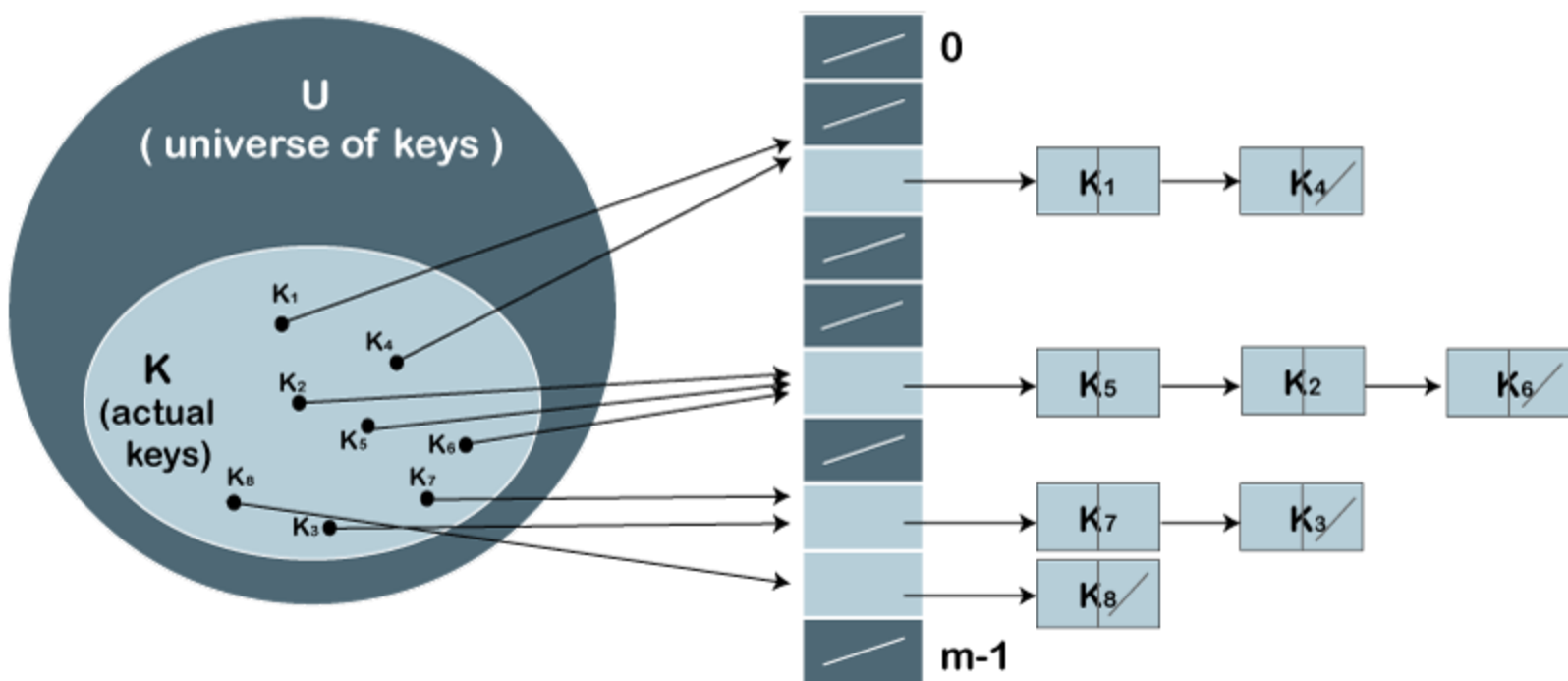
The following are the collision techniques:

- Open Hashing: It is also known as closed addressing.
- Closed Hashing: It is also known as open addressing.

Open Hashing

In Open Hashing, one of the methods used to resolve the collision is known as a chaining method.

Collision Resolution by Chaining



Let's first understand the chaining to resolve the collision.

Suppose we have a list of key values

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and $h(k) = 2k+3$

In this case, we cannot directly use $h(k) = k_i/m$ as $h(k) = 2k+3$

- The index of key value 3 is:

$$\text{index} = h(3) = (2(3)+3)\%10 = 9$$

The value 3 would be stored at the index 9.

- The index of key value 2 is:

$$\text{index} = h(2) = (2(2)+3)\%10 = 7$$

The value 2 would be stored at the index 7.

- The index of key value 9 is:

$$\text{index} = h(9) = (2(9)+3)\%10 = 1$$

The value 9 would be stored at the index 1.

- The index of key value 6 is:

$$\text{index} = h(6) = (2(6)+3)\%10 = 5$$

The value 6 would be stored at the index 5.

- The index of key value 11 is:

$$\text{index} = h(11) = (2(11)+3)\%10 = 5$$

The value 11 would be stored at the index 5. Now, we have two values (6, 11) stored at the same index, i.e., 5. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 11 to this list. After the creation of the new list, the newly created list will be linked to the list having value 6.

- The index of key value 13 is:

$$\text{index} = h(13) = (2(13)+3)\%10 = 9$$

The value 13 would be stored at index 9. Now, we have two values (3, 13) stored at the same index, i.e., 9. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 13 to this list. After the creation of the new list, the newly created list will be linked to the list having value 3.

- The index of key value 7 is:

$index = h(7) = (2(7)+3)\%10 = 7$

The value 7 would be stored at index 7. Now, we have two values (2, 7) stored at the same index, i.e., 7. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 7 to this list. After the creation of the new list, the newly created list will be linked to the list having value 2.

- The index of key value 12 is:

$index = h(12) = (2(12)+3)\%10 = 7$

According to the above calculation, the value 12 must be stored at index 7, but the value 2 exists at index 7. So, we will create a new list and add 12 to the list. The newly created list will be linked to the list having a value 7.

The calculated index value associated with each key value is shown in the below table:

key	Location(u)
3	$((2*3)+3)\%10 = 9$
2	$((2*2)+3)\%10 = 7$
9	$((2*9)+3)\%10 = 1$
6	$((2*6)+3)\%10 = 5$
11	$((2*11)+3)\%10 = 5$
13	$((2*13)+3)\%10 = 9$
7	$((2*7)+3)\%10 = 7$
12	$((2*12)+3)\%10 = 7$

Closed Hashing

In Closed hashing, three techniques are used to resolve the collision:

1. Linear probing
2. Quadratic probing
3. Double Hashing technique

Linear Probing

Linear probing is one of the forms of open addressing. As we know that each cell in the hash table contains a key-value pair, so when the collision occurs by mapping a new key to the cell already occupied by another key, then linear probing technique searches for the closest free locations and adds a new key to that empty cell. In this case, searching is performed sequentially, starting from the position where the collision occurs till the empty cell is not found.

Let's understand the linear probing through an example.

Consider the above example for the linear probing:

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and $h(k) = 2k+3$

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5 respectively. The calculated index value of 11 is 5 which is already occupied by another key value, i.e., 6. When linear probing is applied, the nearest empty cell to the index 5 is 6; therefore, the value 11 will be added at the index 6.

The next key value is 13. The index value associated with this key value is 9 when hash function is applied. The cell is already filled at index 9. When linear probing is applied, the nearest empty cell to the index 9 is 0; therefore, the value 13 will be added at the index 0.

The next key value is 7. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 8; therefore, the value 7 will be added at the index 8.

The next key value is 12. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 2; therefore, the value 12 will be added at the index 2.

Quadratic Probing

In case of linear probing, searching is performed linearly. In contrast, quadratic probing is an open addressing technique that uses quadratic polynomial for searching until a empty slot is found.

It can also be defined as that it allows the insertion k_i at first free location from $(u+i^2)\%m$ where $i=0$ to $m-1$.

Let's understand the quadratic probing through an example.

Consider the same example which we discussed in the linear probing.

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and $h(k) = 2k+3$

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5, respectively. We do not need to apply the quadratic probing technique on these key values as there is no occurrence of the collision.

The index value of 11 is 5, but this location is already occupied by the 6. So, we apply the quadratic probing technique.

When $i = 0$

Index = $(5+0^2)\%10 = 5$

When $i=1$

Index = $(5+1^2)\%10 = 6$

Since location 6 is empty, so the value 11 will be added at the index 6.

The next element is 13. When the hash function is applied on 13, then the index value comes out to be 9, which we already discussed in the chaining method. At index 9, the cell is occupied by another value, i.e., 3. So, we will apply the quadratic probing technique to calculate the free location.

When $i=0$

Index = $(9+0^2)\%10 = 9$

When $i=1$

Index = $(9+1^2)\%10 = 0$

Since location 0 is empty, so the value 13 will be added at the index 0.

The next element is 7. When the hash function is applied on 7, then the index value comes out to be 7, which we already discussed in the chaining method. At index 7, the cell is occupied by another value, i.e., 2. So, we will apply the quadratic probing technique to calculate the free location.

When $i=0$

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

When $i=1$

Index = $(7+1^2)\%10 = 8$

Since location 8 is empty, so the value 7 will be added at the index 8.

The next element is 12. When the hash function is applied on 12, then the index value comes out to be 7. When we observe the hash table then we will get to know that the cell at index 7 is already occupied by the value 2. So, we apply the Quadratic probing technique on 12 to determine the free location.

When $i=0$

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

When $i=1$

Index = $(7+1^2)\%10 = 8$

When $i=2$

Index = $(7+2^2)\%10 = 1$

When $i=3$

Index = $(7+3^2)\%10 = 6$

When i=4

Index = $(7+4^2)\%10 = 3$

Since the location 3 is empty, so the value 12 would be stored at the index 3.

The final hash table would be:

0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	7
9	3

Therefore, the order of the elements is 13, 9, _ , 12, _ , 6, 11, 2, 7, 3.

Double Hashing

Double hashing is an open addressing technique which is used to avoid the collisions. When the collision occurs then this technique uses the secondary hash of the key. It uses one hash value as an index to move forward until the empty location is found.

In double hashing, two hash functions are used. Suppose $h_1(k)$ is one of the hash functions used to calculate the locations whereas $h_2(k)$ is another hash function. It can be defined as "insert k_i at first free place from $(u+v*i)\%m$ where $i=(0 \text{ to } m-1)$ ". In this case, u is the location computed using the hash function and v is equal to $(h_2(k)\%m)$.

Consider the same example that we use in quadratic probing.

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and

$h_1(k) = 2k+3$

$h_2(k) = 3k+1$

key	Location (u)	v	probe
3	$((2*3)+3)\%10 = 9$	-	1
2	$((2*2)+3)\%10 = 7$	-	1
9	$((2*9)+3)\%10 = 1$	-	1
6	$((2*6)+3)\%10 = 5$	-	1
11	$((2*11)+3)\%10 = 5$	$(3(11)+1)\%10 =4$	3
13	$((2*13)+3)\%10 = 9$	$(3(13)+1)\%10 = 0$	
7	$((2*7)+3)\%10 = 7$	$(3(7)+1)\%10 = 2$	
12	$((2*12)+3)\%10 = 7$	$(3(12)+1)\%10 = 7$	2

As we know that no collision would occur while inserting the keys (3, 2, 9, 6), so we will not apply double hashing on these key values.

On inserting the key 11 in a hash table, collision will occur because the calculated index value of 11 is 5 which is already occupied by some another value. Therefore, we will apply the double hashing technique on key 11. When the key value is 11, the value of v is 4.

Now, substituting the values of u and v in $(u+v*i)\%m$

When i=0

Index = $(5+4*0)\%10 = 5$

When i=1

Index = $(5+4*1)\%10 = 9$

When i=2

Index = $(5+4*2)\%10 = 3$

Since the location 3 is empty in a hash table; therefore, the key 11 is added at the index 3.

The next element is 13. The calculated index value of 13 is 9 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 0.

Now, substituting the values of u and v in $(u+v*i)\%m$

When i=0

Index = $(9+0*0)\%10 = 9$

We will get 9 value in all the iterations from 0 to m-1 as the value of v is zero. Therefore, we cannot insert 13 into a hash table.

The next element is 7. The calculated index value of 7 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 2.

Now, substituting the values of u and v in $(u+v*i)\%m$

When i=0

Index = $(7 + 2*0)\%10 = 7$

When i=1

Index = $(7+2*1)\%10 = 9$

When i=2

Index = $(7+2*2)\%10 = 1$

When i=3

Index = $(7+2*3)\%10 = 3$

When i=4

Index = $(7+2*4)\%10 = 5$

When i=5

Index = $(7+2*5)\%10 = 7$

When i=6

Index = $(7+2*6)\%10 = 9$

When i=7

Index = $(7+2*7)\%10 = 1$

When i=8

Index = $(7+2*8)\%10 = 3$

When i=9

Index = $(7+2*9)\%10 = 5$

Since we checked all the cases of i (from 0 to 9), but we do not find suitable place to insert 7. Therefore, key 7 cannot be inserted in a hash table.

The next element is 12. The calculated index value of 12 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 7.

Now, substituting the values of u and v in $(u+v*i)\%m$

When i=0

Index = $(7+7*0)\%10 = 7$

When i=1

Index = $(7+7*1)\%10 = 4$

Since the location 4 is empty; therefore, the key 12 is inserted at the index 4.

The final hash table would be:

0	
1	9
2	
3	11
4	12
5	6
6	
7	2
8	
9	3

The order of the elements is _, 9, _, 11, 12, 6, _, 2, _, 3.

Preorder Traversal

The tree which is shown below has some nodes which are completely filled with the characters. As we already know that in Depth First Traversal, if we go in one direction, then we visit all the nodes in that direction. In other words, if we go in one direction then we visit the whole subtree in that direction. After visiting all the nodes in that subtree, we move in another direction. In the above example, if we are at the root, and we move to the left subtree then we visit all the nodes in the left subtree then only we can move to the right subtree for traversal. Here, the problem is reduced in a recursive manner; therefore, we can say that visiting all the nodes of the tree is visiting the left subtree, then right subtree, and then the root node of the tree. In Depth First strategy, the relative order of visiting left subtree, right subtree, and root is different. The depth-first strategy has one constraint that the left subtree is always traversed before the right subtree.

What is a Preorder Traversal?

The preorder traversal is a traversal in which first we visit the root node, then the left subtree, and then the right subtree. It can be represented as:

<root> <left> <right>

Steps for the Preorder traversal

Step 1: First, we visit the root node.

Step 2: Secondly, we visit the left subtree.

Step 3: Lastly, we visit the right subtree.

Algorithm of Preorder traversal

Step 1: Repeat Steps 2 to 4 while TREE != NULL

Step 2: Write TREE -> DATA

Step 3: PREORDER(TREE -> LEFT)

Step 4: PREORDER(TREE -> RIGHT)
[END OF LOOP]

Step 5: END

Implementation of Binary Tree in C

```
1. #include<stdio.h>
2. // Creating a node structure
3. struct node
4. {
5.     int data;
6.     struct node *left, *right;
7. };
8.
9. // Creating a tree..
10. struct node* create()
11. {
12.     struct node *temp;
13.     int data;
14.     temp = (struct node *)malloc(sizeof(struct node)); // creating a new node
15.     printf("\nEnter the data: ");
16.     scanf("%d", &data);
17.
18.     if(data == -1)
19.     {
20.         return NULL;
21.     }
22.     temp->data = data;
23.     printf("\nEnter the left child of %d", data);
24.     temp->left = create(); // creating a left child
25.     printf("\nEnter the right child of %d", data);
26.     temp->right = create(); // creating a right child
27.     return temp;
28. }
29.
30. //Preorder traversal
31. void preorder(struct node *root)
32. {
33.     if(root == NULL)
34.         return;
35.     else
36.         printf("%d ", root->data);
37.     preorder(root->left);
38.     preorder(root->right);
39. }
40. void main()
```

```
41.  {
42.    struct node *root;
43.    root = create(); // calling create function.
44.    preorder(root); // calling preorder function
45. }
```

Explanation of the above code

In the above code, we have defined two functions, i.e., **create()** and **preorder()** function. The create() function is used to create a tree by creating a new node on every recursive call. In the preorder() function, the root node of type node structure is passed as an argument. The root node is a pointer that points to the root node of the tree. As we know that in preorder traversal, first, we visit the root node, then the left subtree, and then we traverse the right subtree. In the preorder() function, first, we print the root node, then we made a recursive call on the left subtree to print all the nodes of the left subtree and then we made a recursive call on the right subtree to print all the nodes of the right subtree.

Output

```
Enter the data: 78
Enter the left child of 78
Enter the data: 12
Enter the left child of 12
Enter the data: -1
Enter the right child of 12
Enter the data: -1
Enter the right child of 78
Enter the data: 98
Enter the left child of 98
Enter the data: -1
Enter the right child of 98
Enter the data: -1
78 12 98
...Program finished with exit code 0
Press ENTER to exit console.
```

Tree Traversal

Here, tree traversal means **traversing** or **visiting** each node of a tree. Linear data structures like Stack, Queue, linked list have only one way for traversing, whereas the tree has various ways to traverse or visit each node. The following are the three different ways of traversal:

- **Inorder traversal**
- **Preorder traversal**
- **Postorder traversal**

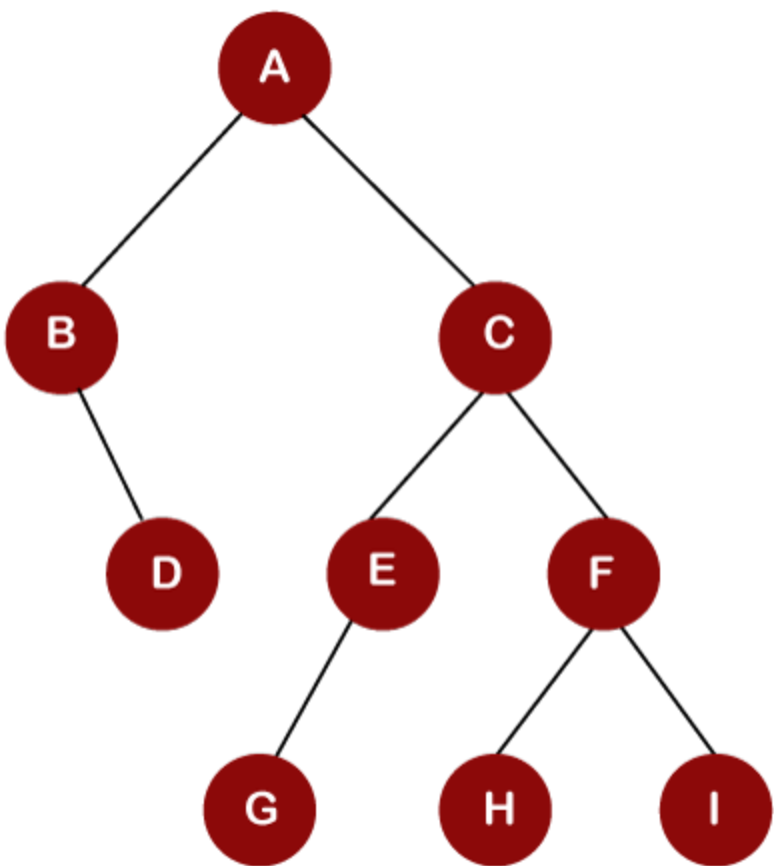
Let's look at each traversal one by one.

Inorder Traversal

An inorder traversal is a traversal technique that follows the policy, i.e., **Left Root Right**. Here, Left Root Right means that the left subtree of the root node is traversed first, then the root node, and then the right subtree of the root node is traversed. Here, inorder name itself suggests that the root node comes in between the left and the right subtrees.

Let's understand the inorder traversal through an example.

Consider the below tree for the inorder traversal.



First, we will visit the left part, then root, and then the right part of performing the inorder traversal. In the above tree, A is a root node, so we move to the left of the A, i.e., B. As node B does not have any left child so B will be printed as shown below:



After visiting node B, we move to the right child of node B, i.e., D. Since node D is a leaf node, so node D gets printed as shown below:



The left part of node A is traversed. Now, we will visit the root node, i.e., A, and it gets printed as shown below:



Once the traversing of left part and root node is completed, we move to the right part of the root node. We move to the right child of node A, i.e., C. The node C has also left child, i.e., E and E has also left child, i.e., G. Since G is a leaf node, so G gets printed as shown below:



The root node of G is E, so it gets printed as shown below:



Since E does not have any right child, so we move to the root of the E node, i.e., C. C gets printed as shown below:



Once the left part of node C and the root node, i.e., C, are traversed, we move to the right part of Node C. We move to the node F and node F has a left child, i.e., H. Since H is a leaf node, so it gets printed as shown below:



Now we move to the root node of H, i.e., F and it gets printed as shown below:



After visiting the F node, we move to the right child of node F, i.e., I, and it gets printed as shown below:



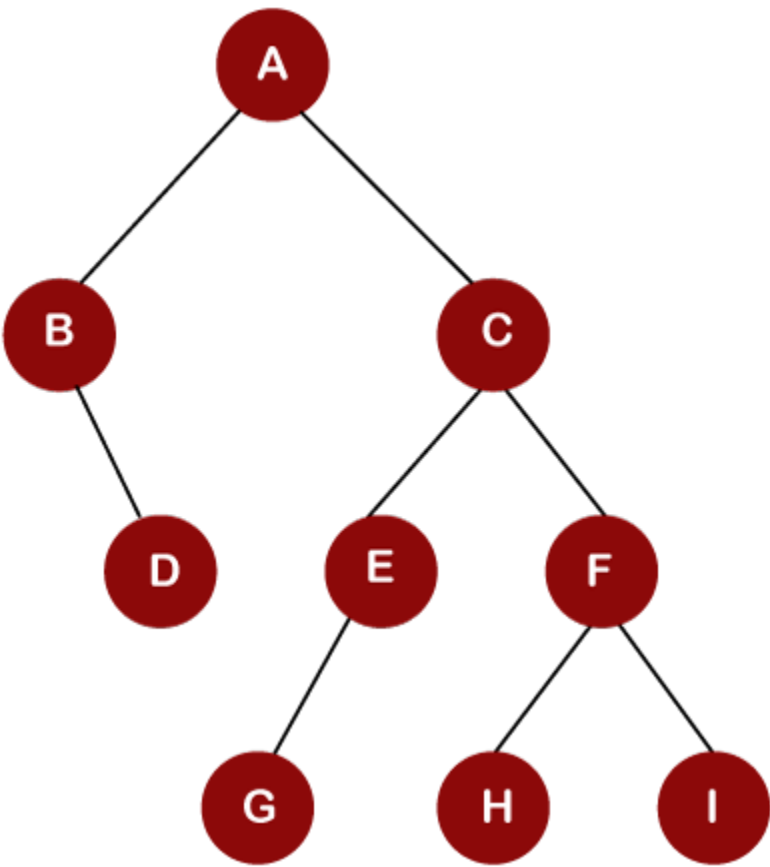
Therefore, the inorder traversal of the above tree is B, D, A, G, E, C, H, F, I.

Preorder Traversal

A preorder traversal is a traversal technique that follows the policy, i.e., **Root Left Right**. Here, Root Left Right means root node of the tree is traversed first, then the left subtree and finally the right subtree is traversed. Here, the Preorder name itself suggests that the root node would be traversed first.

Let's understand the Preorder traversal through an example.

Consider the below tree for the Preorder traversal.



To perform the preorder traversal, we first visit the root node, then the left part, and then we traverse the right part of the root node. As node A is the root node in the above tree, so it gets printed as shown below:



Once the root node is traversed, we move to the left subtree. In the left subtree, B is the root node for its right child, i.e., D. Therefore, B gets printed as shown below:



Since node B does not have a left child, and it has only a right child; therefore, D gets printed as shown below:



Once the left part of the root node A is traversed, we move to the right part of node A. The right child of node A is C. Since C is a root node for all the other nodes; therefore, C gets printed as shown below:



Now we move to the left child, i.e., E of node C. Since node E is a root node for node G; therefore, E gets printed as shown below:



The node E has a left child, i.e., G, and it gets printed as shown below:



Since the left part of the node C is completed, so we move to the right part of the node C. The right child of node C is node F. The node F is a root node for the nodes H and I; therefore, the node F gets printed as shown below:



Once the node F is visited, we will traverse the left child, i.e., H of node F as shown below:



Now we will traverse the right child, i.e., I of node F, as shown below:



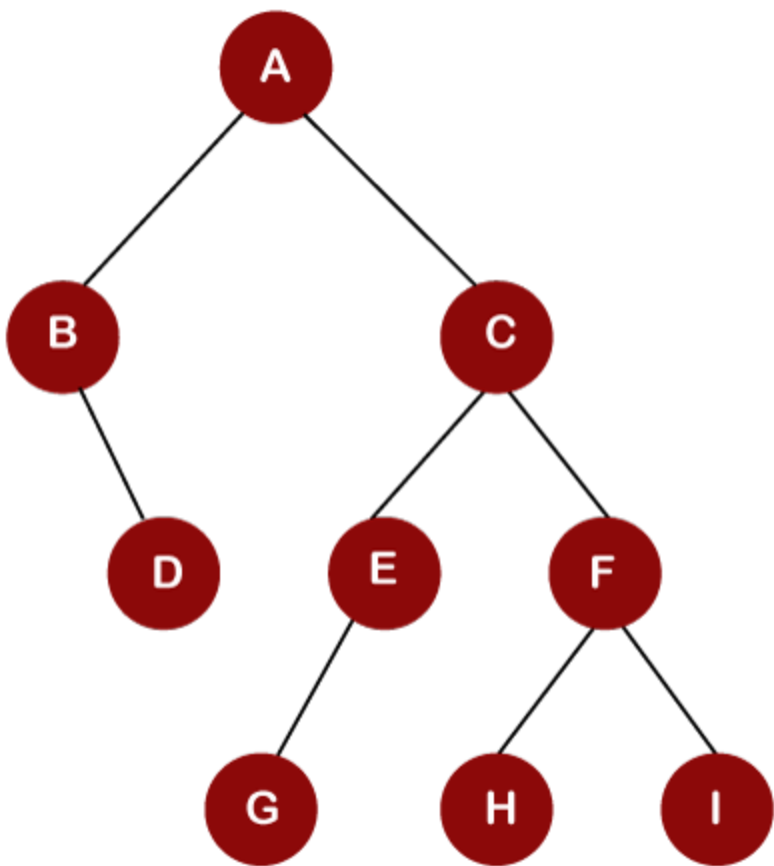
Therefore, the preorder traversal of the above tree is A, B, D, C, E, G, F, H, I.

Postorder Traversal

A Postorder traversal is a traversal technique that follows the policy, i.e., **Left Right Root**. Here, Left Right Root means the left subtree of the root node is traversed first, then the right subtree, and finally, the root node is traversed. Here, the Postorder name itself suggests that the root node of the tree would be traversed at the last.

Let's understand the Postorder traversal through an example.

Consider the below tree for the Postorder traversal.



To perform the postorder traversal, we first visit the left part, then the right part, and then we traverse the root node. In the above tree, we move to the left child, i.e., B of node A. Since B is a root node for the node D; therefore, the right child, i.e., D of node B, would be traversed first and then B as shown below:



Once the traversing of the left subtree of node A is completed, then the right part of node A would be traversed. We move to the right child of node A, i.e., C. Since node C is a root node for the other nodes, so we move to the left child of node C, i.e., node E. The node E is a root node, and node G is a left child of node E; therefore, the node G is printed first and then E as shown below:



Once the traversal of the left part of the node C is traversed, then we move to the right part of the node C. The right child of node C is node F. Since F is also a root node for the nodes H and I; therefore, the left child 'H' is traversed first and then the right child 'I' of node F as shown below:



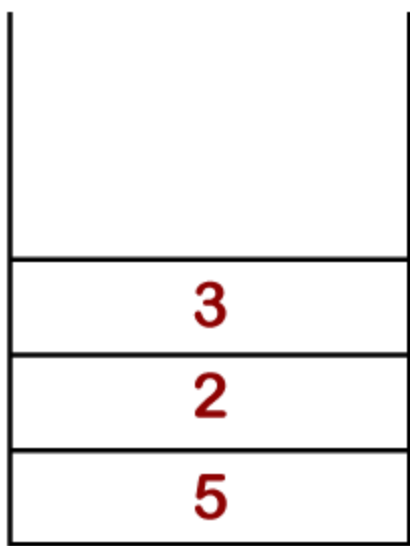
After traversing H and I, node F is traversed as shown below:



Once the left part and the right part of node C are traversed, then the node C is traversed as shown below:

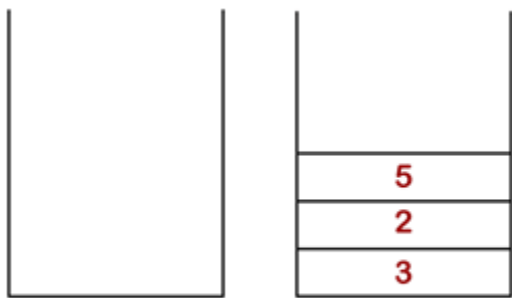


In the above tree, the left subtree and the right subtree of root node A have been traversed, the node A would be traversed.



Stack 1

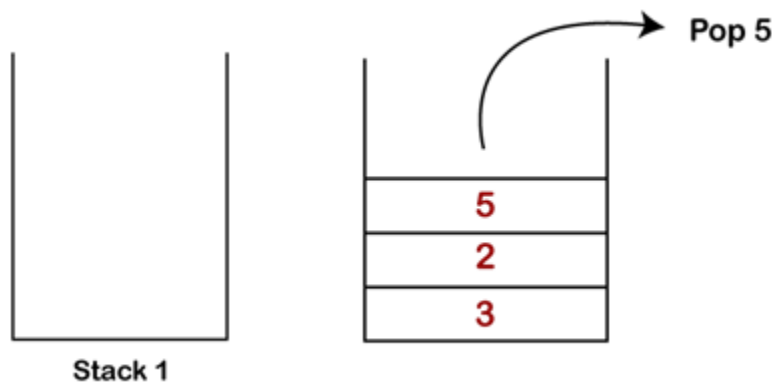
Now we will pop the elements from the Stack1 one by one and push them into the Stack2 as shown as below:



Stack 1

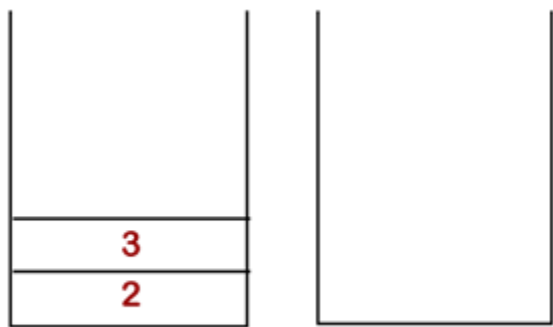
Stack 2

Once the elements are inserted into the Stack2, the topmost element is 5 so it would be popped out from the Stack 2 shown as below:



Stack 1

Once the topmost element is popped out from the Stack2, all the elements are moved back from Stack2 to Stack 1 shown as below:



Stack 1

Stack 2

There are two approaches to implement Queue using Stack:

- Making a dequeue operation costly
- Making a enqueue operation costly

First approach: Making a dequeue operation costly

If we implement the Queue using Stack by making a dequeue operation costly means that time complexity in enqueue operation would be **$O(1)$** and the time complexity in dequeue operation would be **$O(n)$** .

In this case, when we insert the elements in the stack then the elements are added on the top of the stack so it takes $O(1)$ time.

In case of dequeue operation, we need to consider two stacks named as Stack1 and Stack2. First, we insert the elements in the Stack1 and then we remove all the elements from the Stack1. Once all the elements are popped from the Stack1 then they are added in the Stack2. The topmost element would be popped out from the Stack2 and then all the elements from the Stack2 are moved back to Stack1. Here, dequeue operation is performed two times on the data so time complexity is $O(n)$.

Implementation in C

```
1. // Program to implement Queue using Stack in C.
2. #include<stdio.h>
3. #define N 5
4. int stack1[5], stack2[5]; // declaration of two stacks
5. // declaration of top variables.
6. int top1=-1, top2=-1;
7. int count=0;
8. // inserting the elements in stack1.
9. void push1(int data)
10. {
11. // Condition to check whether the stack1 is full or not.
12. if(top1==N-1)
13. {
14. printf("\n Stack is overflow...");
15. }
16. else
17. {
18. top1++; // Incrementing the value of top1
19. stack1[top1]=data; // pushing the data into stack1
20. }
21. }
22. // Removing the elements from the stack1.
23. int pop1()
24. {
25. // Condition to check whether the stack1 is empty or not.
26. if(top1== -1)
27. {
28. printf("\nStack is empty..");
29. }
30. else
31. {
32. int a=stack1[top1]; // Assigning the topmost value of stack1 to 'a' variable.
33. top1--; // decrementing the value of top1.
34. return a;
35. }
36. }
37. // pushing the data into the stack2.
38. void push2(int x)
39. {
40. // Condition to check whether the stack2 is full or not
41. if(top2==N-1)
42. {
43. printf("\nStack is full..");
44. }
45. else
46. {
47. top2++; // incrementing the value of top2.
48. stack2[top2]=x; // assigning the 'x' value to the Stack2
```

```

49.
50.}
51.}
52. // Removing the elements from the Stack2
53. int pop2()
54. {
55.     int element = stack2[top2]; // assigning the topmost value to element
56.     top2--; // decrement the value of top2
57.     return element;
58. }
59. void enqueue(int x)
60. {
61.     push1(x);
62.     count++;
63. }
64. void dequeue()
65. {
66.     if((top1== -1) && (top2== -1))
67. {
68.     printf("\nQueue is empty");
69. }
70. else
71. {
72.     for(int i=0;i<count;i++)
73.     {
74.         int element = pop1();
75.         push2(element);
76.     }
77. int b= pop2();
78. printf("\nThe dequeued element is %d", b);
79. printf("\n");
80. count--;
81. for(int i=0;i<count;i++)
82. {
83.     int a = pop2();
84.     push1(a);
85. }
86. }}
87. void display()
88. {
89.     for(int i=0;i<=top1;i++)
90.     {
91.         printf("%d , ", stack1[i]);
92.     }
93. }
94. void main()
95. {
96.     enqueue(10);
97.     enqueue(20);
98.     enqueue(30);
99.     dequeue();
100.     enqueue(40);
101.     display();
102. }

```

Output

```
The dequeued element is 10
20 , 30 , 40 ,

...Program finished with exit code 2
Press ENTER to exit console.
```

Second Approach: Making an enqueue operation costly.

If we implement the Queue using Stack by making a enqueue operation costly means that time complexity in enqueue operation would be **$O(n)$** and the time complexity in dequeue operation would be **$O(1)$** .

First, we will consider two stacks named as **stack1** and **stack2**. In case of enqueue operation, first all the elements will be popped from the stack1 and push it into the stack2. Once all the elements from the stack1 are pushed into the stack2, then the new element is added in the stack1. After adding the new element in the stack1, all the element are moved back from stack1 to stack2. Here, the time complexity of enqueue operation would be $O(n)$.

In stack1, the oldest element would be at the top of the stack, so time taken to perform a dequeue operation would be $O(1)$.

Implementation in C

```
1. #include<stdio.h>
2. #define N 5
3. int stack1[5], stack2[5]; // declaration of two stacks
4. // declaration of top variables.
5. int top1=-1, top2=-1;
6. int count=0;
7. // inserting the elements in stack1.
8. void push1(int data)
9. {
10. // Condition to check whether the stack1 is full or not.
11. if(top1==N-1)
12. {
13. printf("\n Stack is overflow...");
14. }
15. else
16. {
17. top1++; // Incrementing the value of top1
18. stack1[top1]=data; // pushing the data into stack1
19. }
20. }
21. // Removing the elements from the stack1.
22. int pop1()
23. {
24. // Condition to check whether the stack1 is empty or not.
25. if(top1== -1)
26. {
27. printf("\nStack is empty..");
28. }
29. else
30. {
31. int a=stack1[top1]; // Assigning the topmost value of stack1 to 'a' variable.
32. top1--; // decrementing the value of top1.
33. return a;
34. }
35. }
36. // pushing the data into the stack2.
37. void push2(int x)
```

```

38. {
39. // Condition to check whether the stack2 is full or not
40. if(top2==N-1)
41. {
42.   printf("\nStack is full..");
43. }
44. else
45. {
46.   top2++; // incrementing the value of top2.
47.   stack2[top2]=x; // assigning the 'x' value to the Stack2
48.
49. }
50. }
51. // Removing the elements from the Stack2
52. int pop2()
53. {
54.   int element = stack2[top2]; // assigning the topmost value to element
55.   top2--; // decrement the value of top2
56.   return element;
57. }
58. // inserting the data into the Queue
59. void enqueue(int x)
60. {
61.
62.   while(top1!=-1)
63.   {
64.     push2(pop1());
65.   }
66.   push1(x);
67.   while(top2!=-1)
68.   {
69.     push1(pop2());
70.   }
71.
72. }
73. // deleting the data from the Queue
74. int dequeue()
75. {
76.   int element = stack1[top1];
77.   top1--;
78.   return element;
79. }
80. // displaying the data of the Queue
81. void display()
82. {
83.   printf("\n");
84.   for(int i=top1; i>=0;i--)
85.   {
86.     printf("%d , ", stack1[i]);
87.   }
88. }
89. void main()
90. {
91.   enqueue(1);
92. enqueue(2);
93. enqueue(3);
94. int d = dequeue();

```

```
95. printf("\nThe deleted element is %d", d);
96. enqueue(4);
97. enqueue(5);
98. display();
99. }
```

Output

```
The deleted element is 1
2 , 3 , 4 , 5 ,

...Program finished with exit code 4
Press ENTER to exit console. □
```

Implementation of Stack using Queue

A stack is a linear data structure that follows the LIFO principle, which means that the element inserted first will be removed last. On the other hand, Queue is a linear data structure that follows the FIFO principle, which means that the added element will be removed first. Now, we will discuss how we can implement the **Stack** using **Queue**.

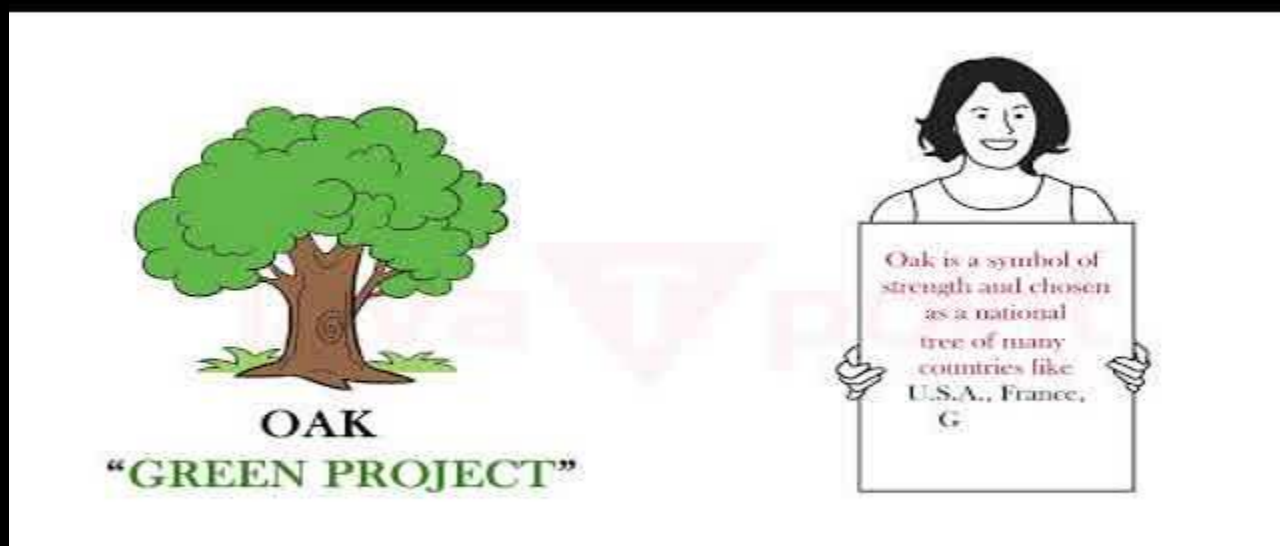
There are two approaches to implement stack using Queue:

- First, we can make the push operation costly.
- Second, we can make the pop operation costly.

First approach: Making a push operation costly

Let's understand through an example.

Suppose we have a list given below:



1, 5, 3, P, 2, P

In the above list, 'P' means that we have to implement the pop operation whereas, the integers 1, 5, 3, and 2 are to be inserted in the stack. We will implement through a Queue. Or we can say that we will implement the push and pop operations through Queue.

First, we create two empty Queues as shown below:



Queue 1



Queue 2

In the above, we have created two Queues, i.e., **Queue1** and **Queue2**. First, we push element 1 into the Queue1, so front and rear point to element 1 as shown below:



↑ ↑
Front Rear

Queue 1



After inserting element 1, we have to insert element 5 into Queue1. First, we pop the element 1 from the Queue1 and push it into the Queue2, and then we push the element 5 into the Queue1 as shown below:



↑ ↑
Front Rear

Queue 1



↑ ↑
Front Rear

Queue 2

As we can observe in the above figure, the front and rear in Queue1 point to element 5, whereas the front and rear in Queue2 point to element 1. Once the insertion of element 5 is completed, element 1 from Queue2 moves back to Queue2. In Queue1, the front will point to element 5, and rear will point to element 1, as shown below:



↑ ↑
Front Rear

Queue 1



Queue 2

Now the next element is 3 which we have to insert in Queue1. In order to achieve this, all the elements, i.e., 5 and 1 from the Queue1, will be popped out and added into the Queue2.



Queue 1



Queue 2

Once the elements are popped out from the Queue1, the element 3 will be pushed into the Queue1, and front will point to element 3 as shown below:

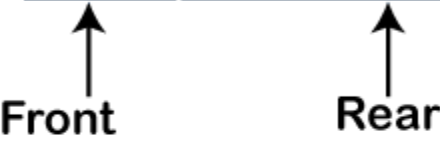


Queue 1



Queue 2

After pushing element 3 in Queue1, we will pop all the elements from Queue2 and push them back to Queue1. The front will point to element 3 and the rear will point to element 1, as shown below:

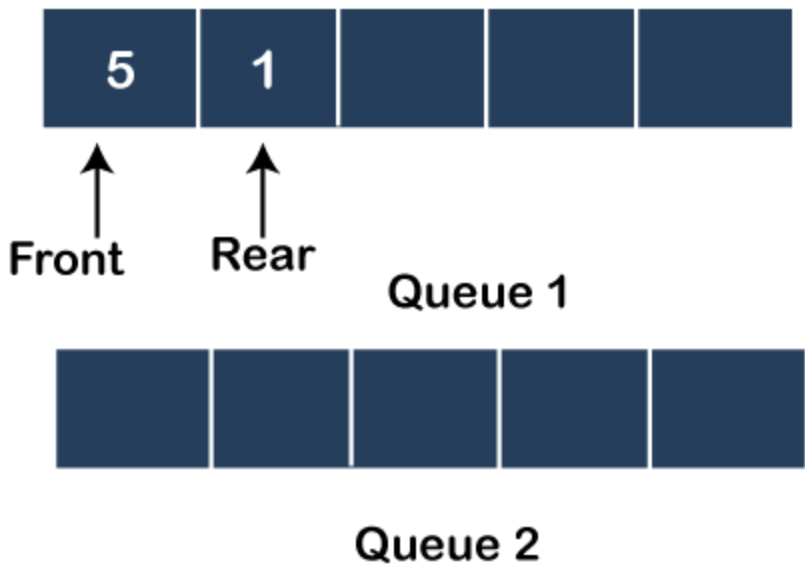


Queue 1

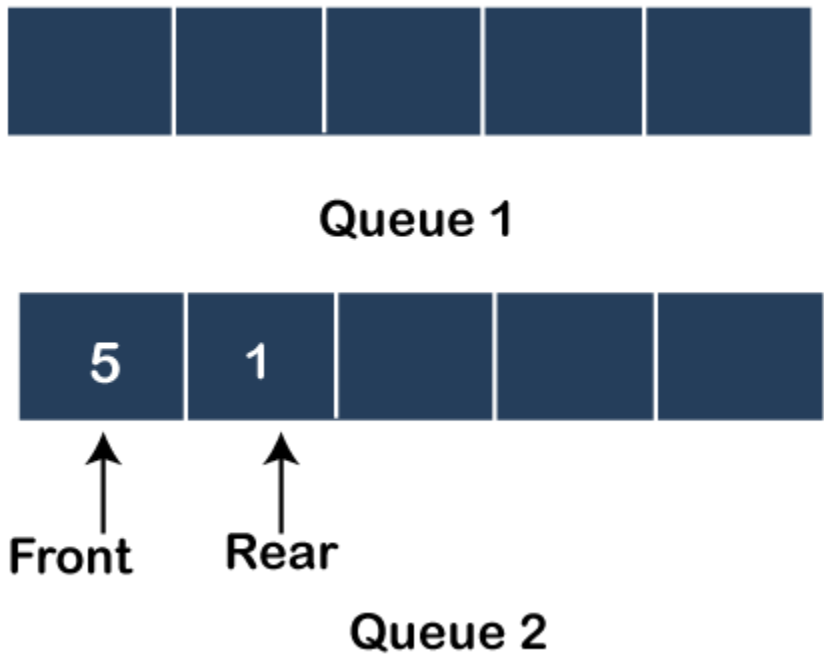


Queue 2

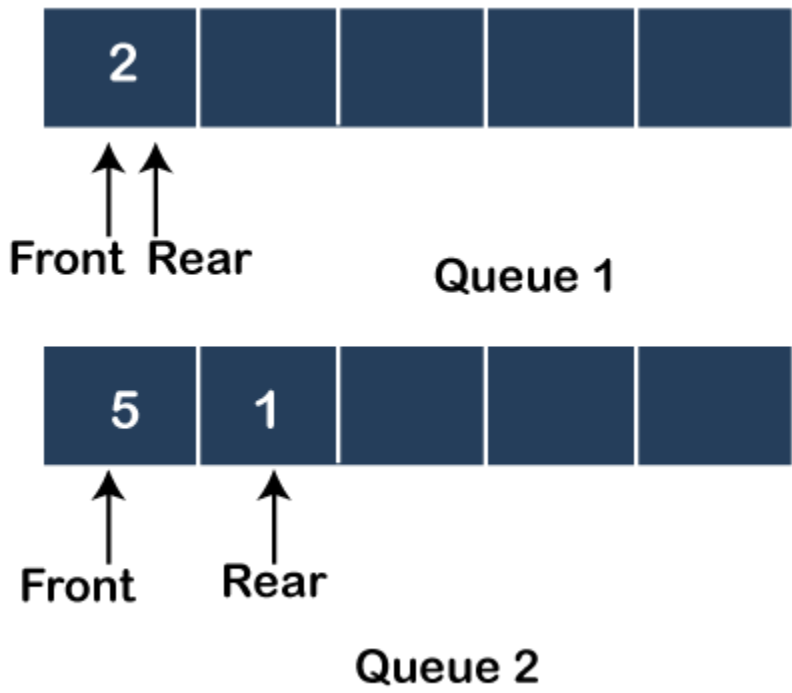
The next operation is a pop operation. Till now, we have observed that the push operation is costly, but the pop operation takes $O(1)$ time. So, we will pop element 3 from Queue1 and update the front pointer. The popped element will be printed in the output. Now front will point to element 5 as shown below:



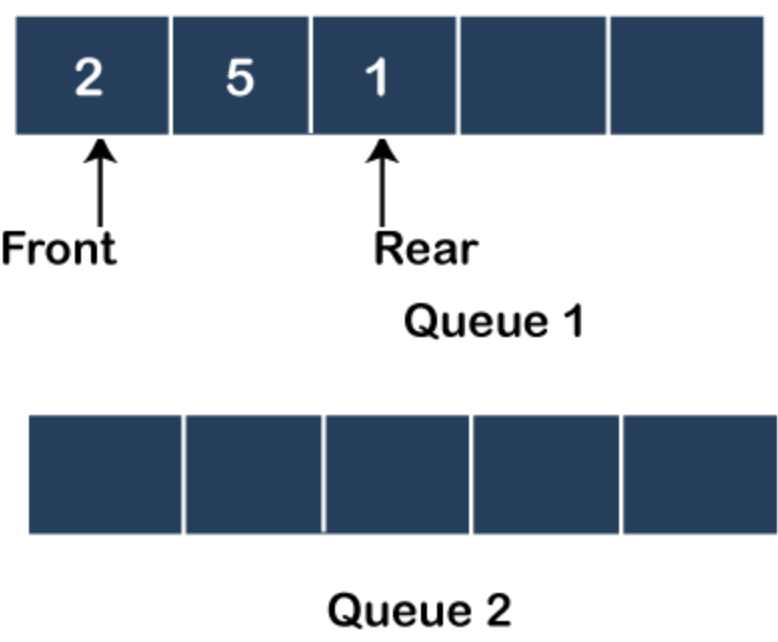
The next element to be inserted is 2. First, we will pop all the elements from the Queue1 and add into the Queue2 as shown below:



Once all the elements are popped out from the Queue1, element 2 would be pushed into the Queue1. The front and rear of Queue1 will point to element 2 as shown below:

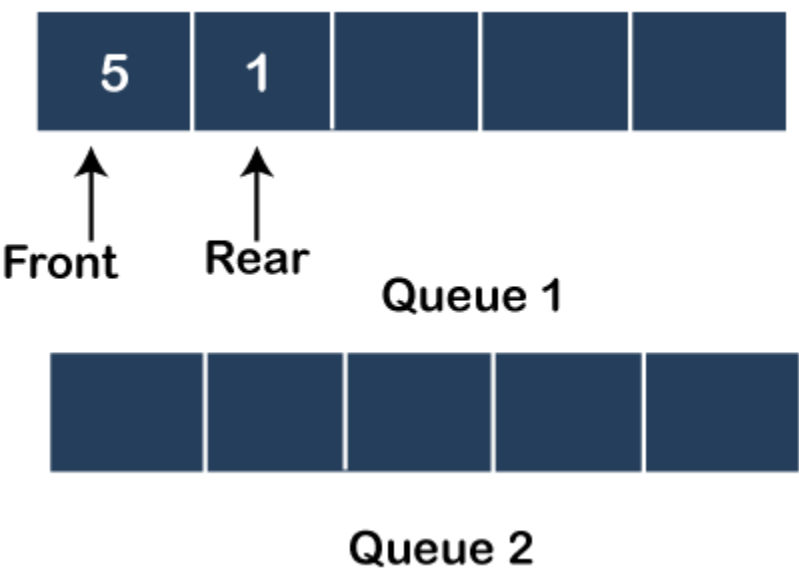


After inserting the element into Queue1, we will pop all the elements from Queue2 and move them back to Queue1 as shown below:



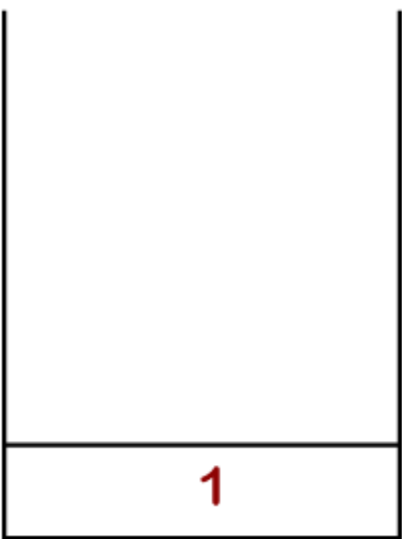
As we can observe in the above figure, the front points to the element 2 while the rear points to element 1.

The next operation is the pop operation. In the pop operation, element 2 would be popped out from the Queue1 and gets printed in the output. The front pointer gets updated and points to element 5 as shown below:

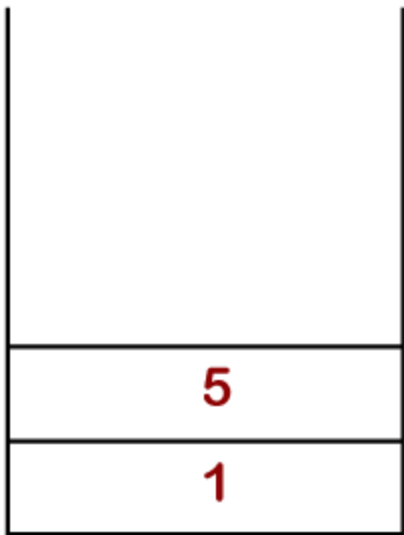


The output is 3, 2.

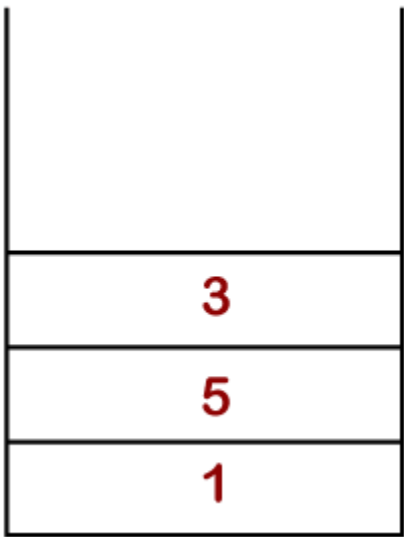
If we want to verify whether the output is correct or not, then we can use stack. First, we push element 1 into the stack as shown below:



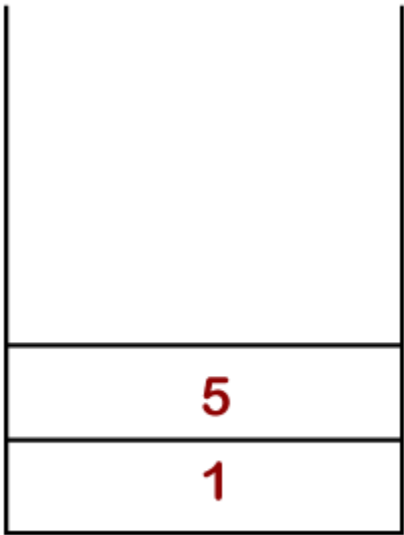
The next element 5 is pushed into the stack as shown below:



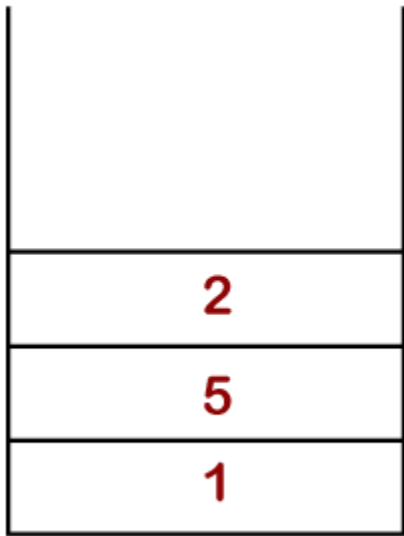
The next element is 3 will be pushed into the stack as shown below:



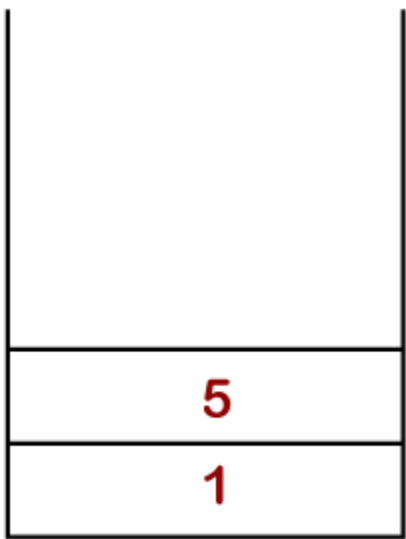
Now pop operation will be called, and element 3 will be popped out from the stack. The element 3 gets printed in the output as shown below:



The next element is 2 to be pushed into the stack:



After inserting 2, the pop operation will be called, and element 2 will be popped out from the stack. The element 2 gets printed in the output.



The output is 3, 2, which is same as the output generated through the implementation of the Queue.

Time Complexity

If we implement the stack through Queue, then push operation will take $O(n)$ time as all the elements need to be popped out from the Queue1 and push them back to Queue1.

The pop operation will take $O(1)$ time because we need to remove front element from the Queue.

Algorithm (when the push operation is costly)

Push Algorithm

The following are the steps to perform the push operation:

Step 1: Consider two queues, i.e., Q1 and Q2, and the element to be inserted in the queue is x.

Step 2: if Q1.isEmpty() then

```
Q1.enqueue(x);  
  
else  
  
size:= Q1.size();  
  
for i=0...size do  
  
Q2.enqueue(Q1.dequeue());  
  
end  
  
Q1.enqueue(x);  
  
for j=0....size do  
  
Q1.enqueue(Q2.dequeue());  
  
end
```

Pop Algorithm

The following are the steps to perform the pop operation:

Step 1: Consider two queues, i.e., Q1 and Q2, and we want to remove the element from the front of the queue.

Step 2: item:= Q1.dequeue();

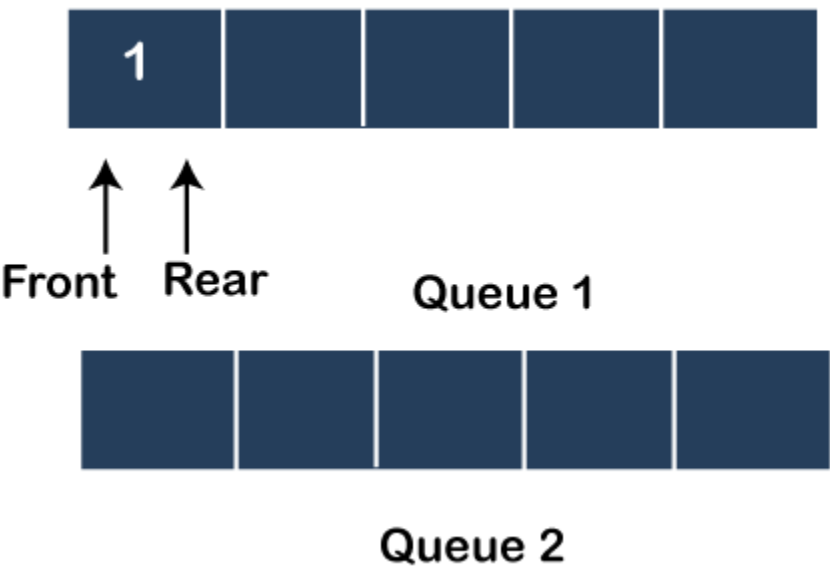
Step 3: return item;

Second approach: Making pop operation costly

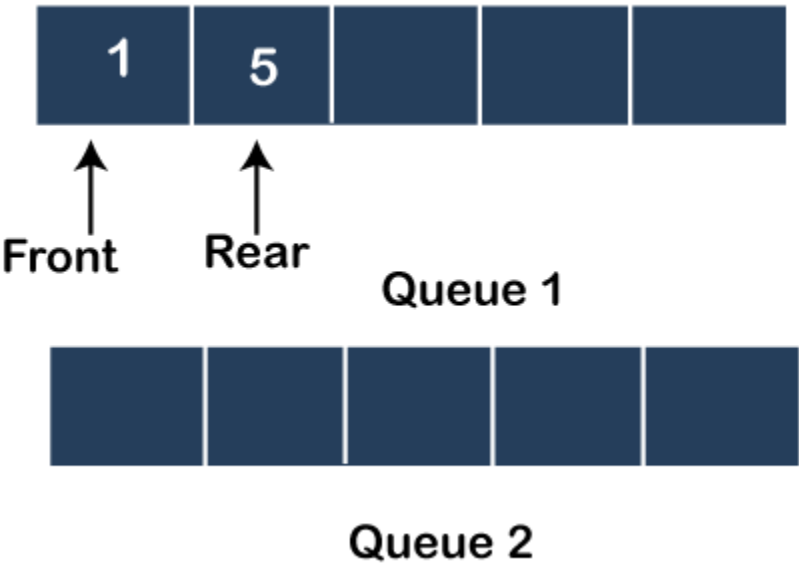
Suppose we have a list given below:

1, 5, 3, P, 2, P

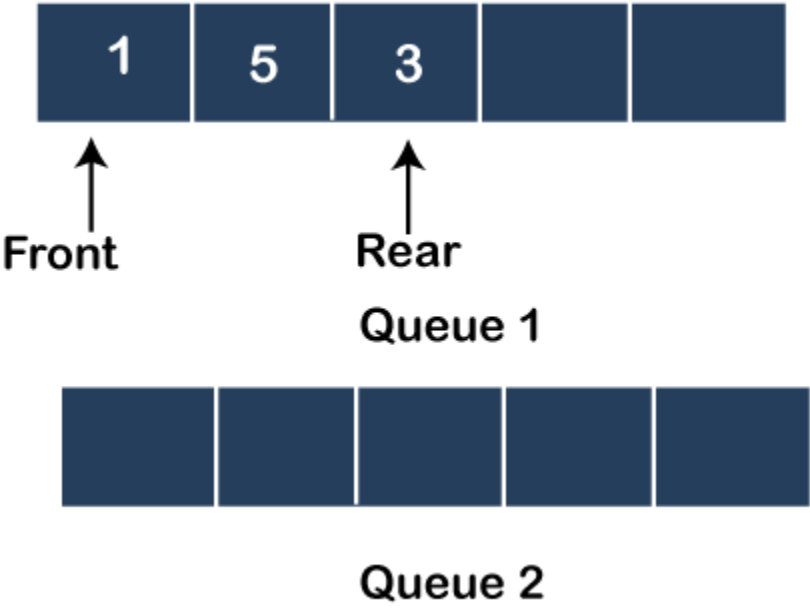
We will consider two Queues, i.e., Queue1 and Queue2 as we have done in the previous approach. First, we will push element 1 into the Queue1 as shown below:



The next element is 5 which will be pushed into the Queue1 as shown below:



The next element is 3 which will also be pushed into the Queue1 as shown below:

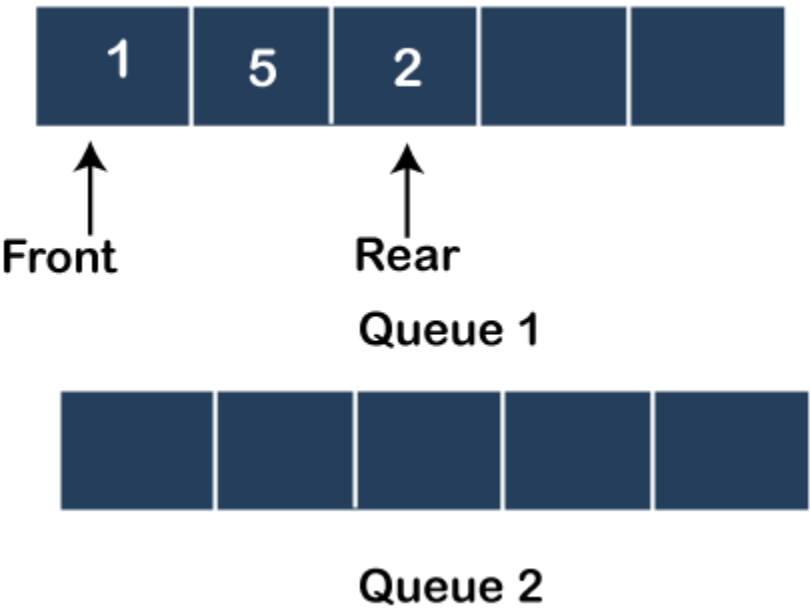


Now we have to implement the pop operation on the Queue1. In this case, we will first pop all the elements except the last pointed by rear and add them into the Queue2. The last element will be removed from the Queue1 and gets printed in the output as shown below:

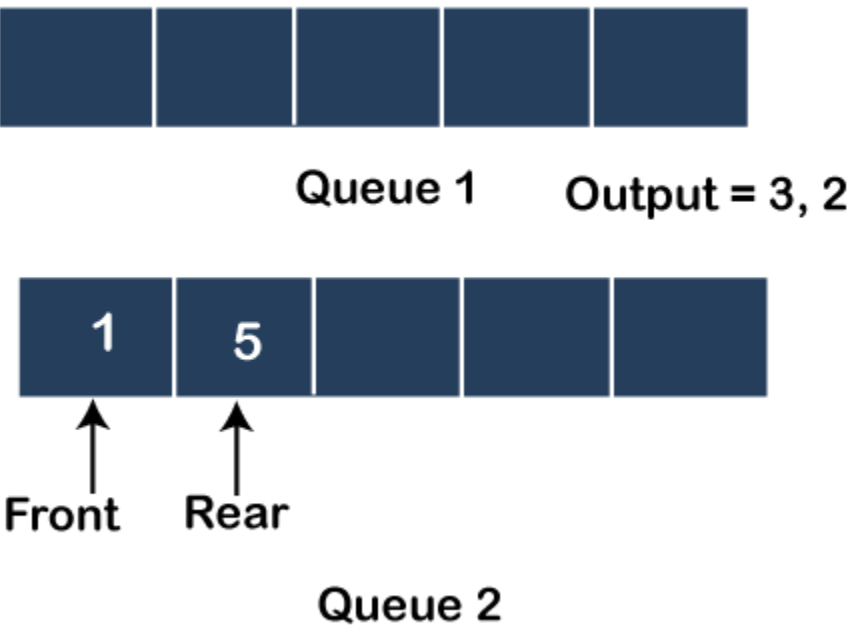


Now we will move the elements of Queue2 back to Queue1.

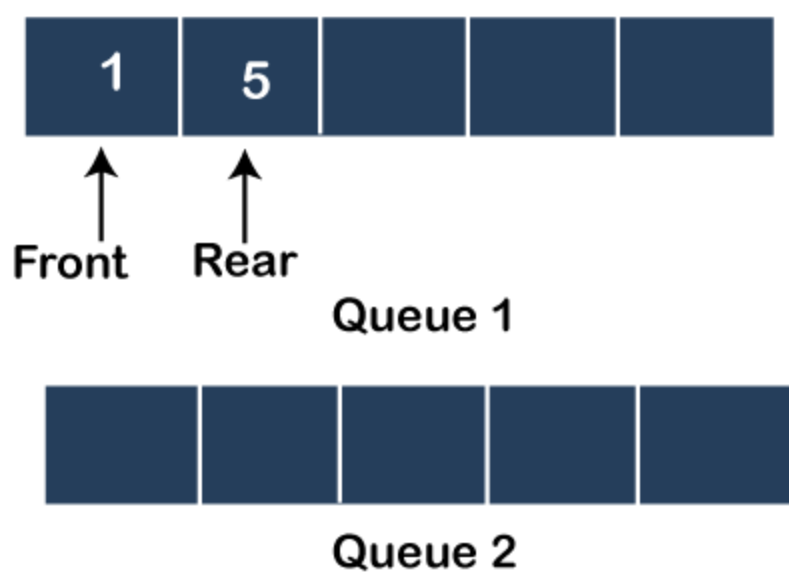
The next element is 2 which will be inserted into the Queue1 as shown below:



The next operation is the pop operation. In this operation, first, we need to pop all the elements from Queue1 except the last element pointed by rear and add it into the Queue2. The last element, i.e., 2 will be removed from the Queue1 and gets printed in the output as shown below:



The elements added in the Queue2 will be moved back to Queue1 as shown below:



As we can observe in the above figure that the output generated as 3, 2 and elements remaining in the Queue are 1, 5.

Time Complexity

In the above case, the push operation takes $O(1)$ time because on each push operation the new element is added at the end of the Queue. On the other hand, pop operation takes $O(n)$ because on each pop operation, all the elements are popped out from the Queue1 except the last element and pushed it into the Queue2. The last element from the Queue1 will be deleted and then all the elements from Queue2 are moved back to the Queue1.

Algorithm (when the pop operation is costly)

Push Algorithm

The following are the steps to perform the push operation:

Step 1: Consider two queues, i.e., Q1 and Q2, and the element to be inserted in the queue is x.

Step 2: element= Q1.enqueue(x);

Step 3: return element;

Pop Algorithm

The following are the steps to delete an element from the queue:

Step 1: Consider two queues, i.e., Q1 and Q2, and we want to remove an element from the queue.

Step 2: if !Q1.isEmpty() then

size:= Q1.size();

for i=0...size-1 do

Q2.enqueue(Q1.dequeue());

end

int item = Q1.dequeue();

for j=0...size-1 do

Q1.enqueue(Q2.dequeue());

end

Binomial Heap

As we have already discussed about the heap data structure which is of two types, i.e., **min heap and max heap**. A binomial heap can be defined as the collection of binomial tree that satisfies the heap properties, i.e., min heap. The min heap is a heap in which each node has a value lesser than the value of its child nodes.

To understand the binomial heap, we first understand about the binomial tree.

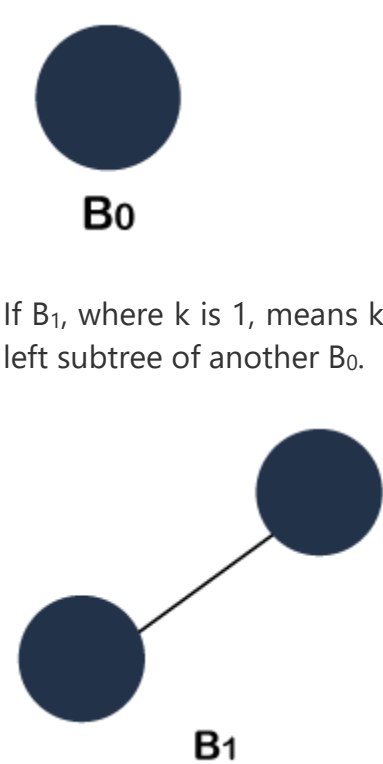
What is Binomial tree?

A Binomial tree is a tree in which B_k is an ordered tree defined recursively, where k is defined as the order of the binomial tree.

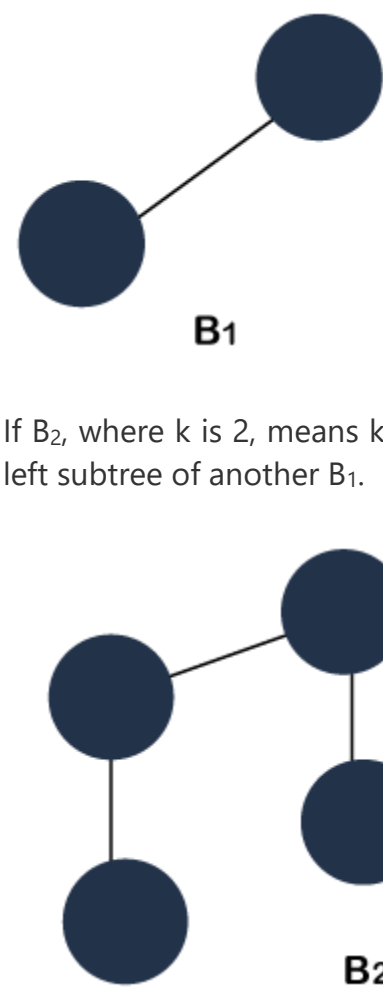
- If the binomial tree is represented as B_0 then the tree consists of a single node.
- In general terms, B_k consists of two binomial trees, i.e., B_{k-1} and B_{k-1} are linked together in which one tree becomes the left subtree of another binomial tree. It can be represented as:

Let's understand through examples.

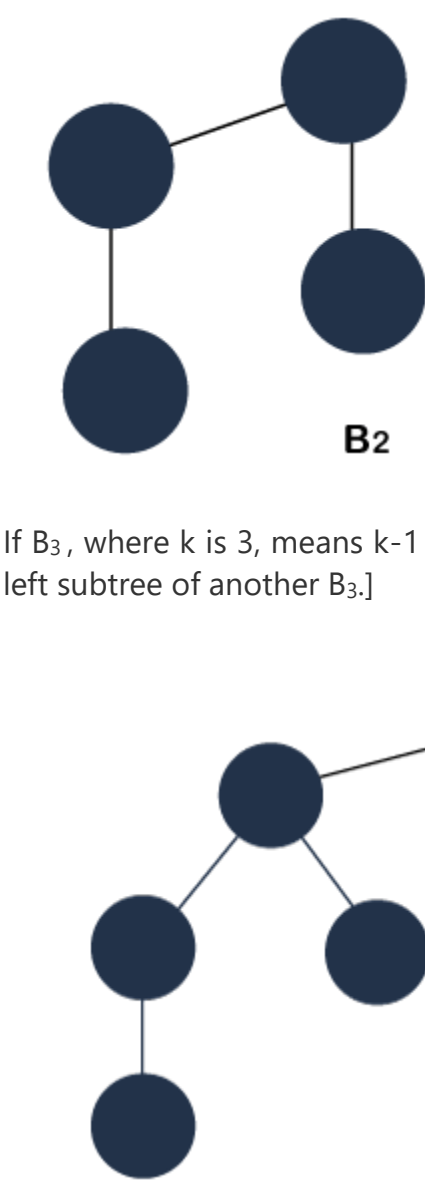
If B_0 , where k is 0, means that there would exist only one node in the tree shown as below:



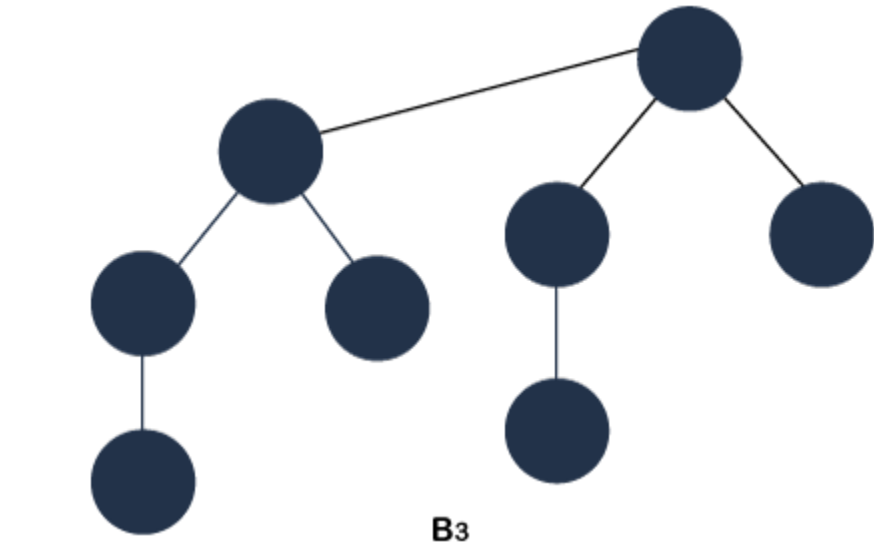
If B_1 , where k is 1, means $k-1$ equal to 0. Therefore, there would be two binomial trees of B_0 in which one B_0 becomes the left subtree of another B_0 .



If B_2 , where k is 2, means $k-1$ equal to 1. Therefore, there would be two binomial trees of B_1 in which one B_1 becomes the left subtree of another B_1 .



If B_3 , where k is 3, means $k-1$ equal to 2. Therefore, there would be two binomial trees of B_3 in which one B_3 becomes the left subtree of another B_3 .]



Properties of Binomial tree

- There are 2^k nodes in a binomial tree of height k .
If $k=1$ then $2^0 = 1$. The number of nodes is 1.
If $k = 2$ then $2^1 = 2$. The number of nodes is 2.
- The height of the tree is k .
If $k=0$ then the height of the tree would be 0.
If $k=1$ then the height of the tree would be 1.
- There are exactly $\binom{k}{i}$ or $k!/i!(k-i)!$ nodes at depth $i = 0, 1..k$.
For example, if k is equal to 4 and at depth $i=2$, we have to determine the number of nodes.
$$\binom{4}{2} = \frac{4!}{2! * (4-2)!} = 6 \text{ nodes}$$

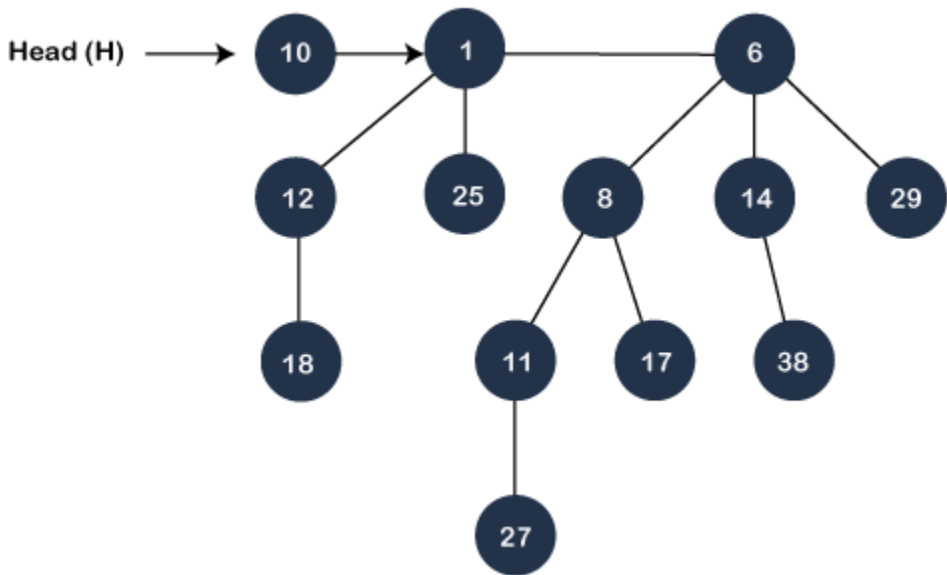
Binomial Heap

A binomial heap is a collection of binomial trees that satisfies the properties of a min- heap.

The following are the two properties of the binomial heap:

- Each binomial heap obeys the min-heap properties.
- For any non-negative integer k , there should be atleast one binomial tree in a heap where root has degree k .

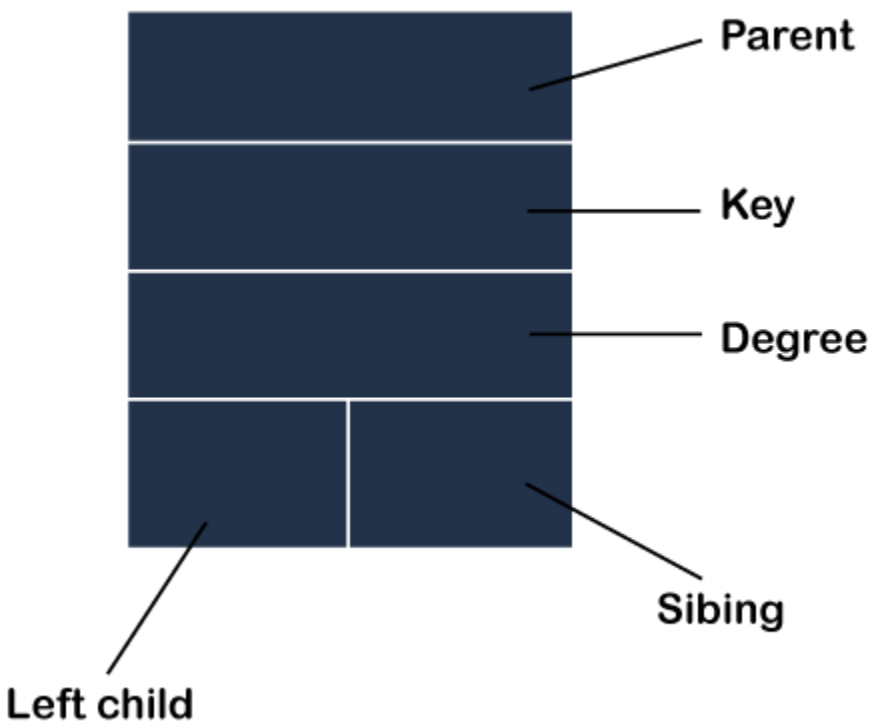
Let's understand the above two properties through an example.



The above figure has three binomial trees, i.e., B_0, B_2, B_3 . The above all three binomial trees satisfy the min heap's property as all the nodes have a smaller value than the child nodes.

The above figure also satisfies the second property of the binomial heap. For example, if we consider the value of k as 3, then we can observe in the above figure that the binomial tree of degree 3 exists in the heap.

Representation of Binomial heap node



The above figure shows the representation of the binomial heap node in the memory. The first block in the memory contains the pointer that stores the address of the parent of the node.

The second block stores the key value of the node.

The third block contains the degree of the node.

The fourth block is divided into two parts, i.e., left child and sibling. The left child contains the address of the left child of a node, whereas the sibling contains the address of the sibling.

Important point

- If we want to create the binomial heap of 'n' nodes, that can be simply defined by the binary number of 'n'. For example: if we want to create the binomial heap of 13 nodes; the binary form of 13 is 1101, so if we start the numbering from the rightmost digit, then we can observe that 1 is available at the 0, 2, and 3 positions; therefore, the binomial heap with 13 nodes will have B_0 , B_2 , and B_3 binomial trees.

Let's consider another example.

If we want to create the binomial heap of 9 nodes. The binary form of 9 is 1001. As we can observe in the binary number that 1 digit is available at 0 and 3 position; therefore, the binomial heap will contain the binomial trees of 0 and 3 degree.

Operations on Binomial Heap

- **Creating a new binomial heap:** When we create a new binomial heap then it simply takes $O(1)$ time because creating a heap will create the head of the heap in which no elements are attached.
- **Finding the minimum key:** As we know that binomial heap is a collection of binomial trees and each binomial tree satisfies the property of min heap means root node contains the minimum value. Therefore, we need to compare only root node of all the binomial trees to find the minimum key. The time complexity for finding a minimum key is $O(\log n)$.
- **Union of two binomial heap:** If we want to combine two binomial heaps, then we can simply find the union of two binomial heaps. The time complexity for finding a union is $O(\log n)$.
- **Inserting a node:** The time complexity for inserting a node is **$O(\log n)$** .
- **Extracting minimum key:** The time complexity for removing a minimum key is $O(\log n)$.
- **Decreasing a key:** When the key value of any node is changed, then the binomial tree does not satisfy the min-heap. We need to perform some rearrangements in order to satisfy the min-heap property. The time complexity would be **$O(\log n)$** .
- **Deleting a node:** The time complexity for deleting a node is $O(\log n)$.

Union of two Binomial Heap

To perform the union to two binomial heap, we will use the following cases:

Case 1: If $\text{degree}[x]$ is not equal to $\text{degree}[\text{next } x]$ then move pointer ahead.

Case 2: if $\text{degree}[x] = \text{degree}[\text{next } x] = \text{degree}[\text{sibling}(\text{next } x)]$ then

Move pointer ahead.

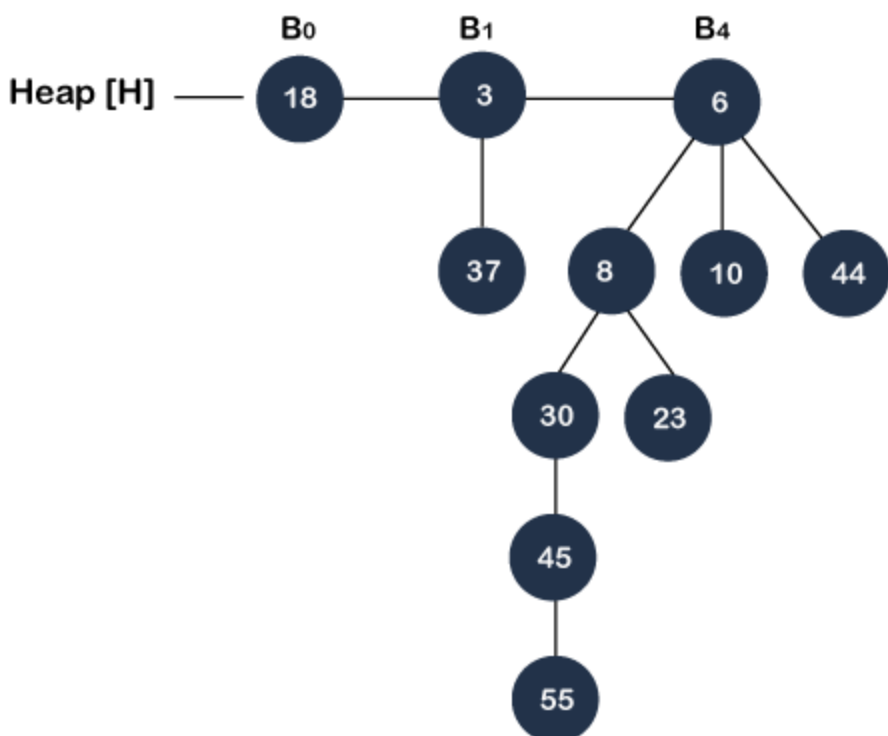
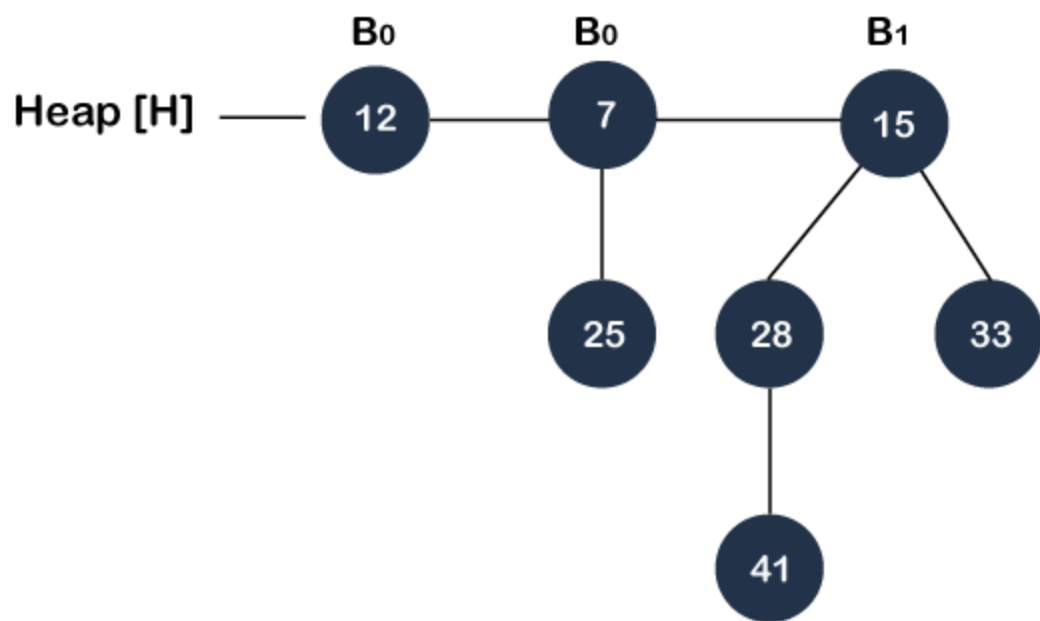
Case 3: If $\text{degree}[x] = \text{degree}[\text{next } x]$ but not equal to $\text{degree}[\text{sibling}[\text{next } x]]$

and $\text{key}[x] < \text{key}[\text{next } x]$ then remove $[\text{next } x]$ from root and attached to x .

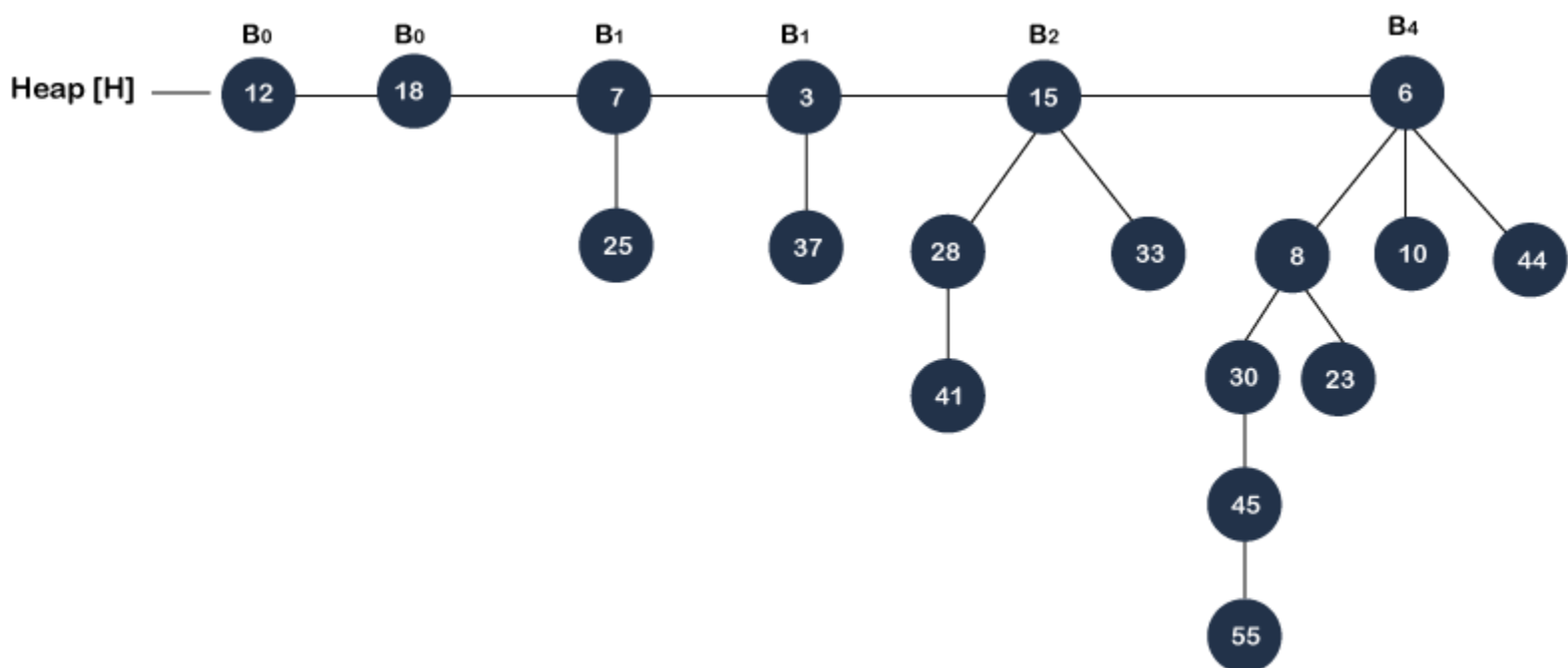
Case 4: If $\text{degree}[x] = \text{degree}[\text{next } x]$ but not equal to $\text{degree}[\text{sibling}[\text{next } x]]$

and $\text{key}[x] > \text{key}[\text{next } x]$ then remove x from root and attached to $[\text{next } x]$.

Let's understand the union of two binomial heaps through an example.



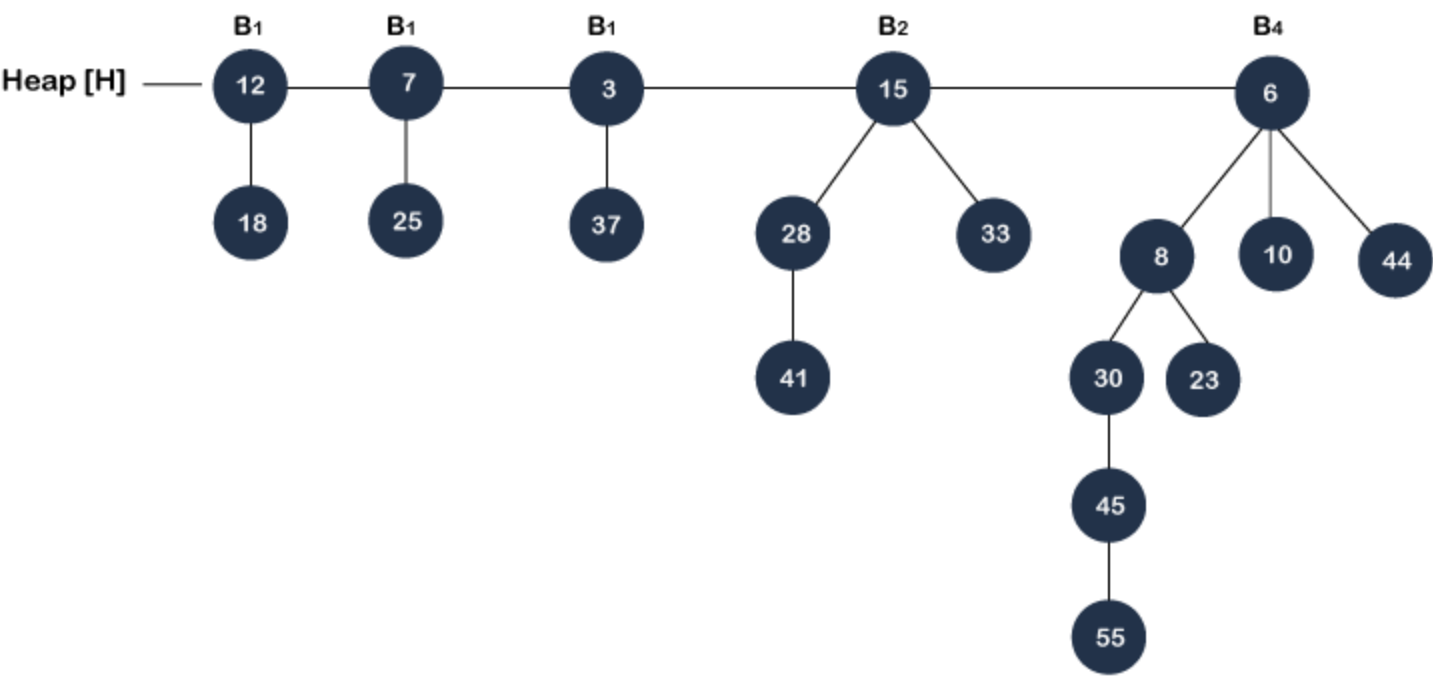
As we can observe in the above figure that there are two binomial heaps. First, we combine these two binomial heaps. To combine these two binomial heaps, we need to arrange them in the increasing order of binomial trees shown as below:



- Initially, x points to the B_0 having value 12, and $\text{next}[x]$ points to B_0 having value 18. The B_1 is the sibling of B_0 having node value 18. Therefore, the sibling B_1 is represented as $\text{sibling}[\text{next } x]$. Now, we will apply case 1. Case 1 says that **'if $\text{degree}[x]$ is not equal to $\text{degree}[\text{next } x]$ then move pointer ahead'** but in the above example, the degree of x is the same as the degree of next x , so this case is not valid.

Now we will apply case 2. The case 2 says that **'if $\text{degree}[x] = \text{degree}[\text{next } x] = \text{degree}[\text{sibling}(\text{next } x)]$ then Move pointer ahead'**. In the above example, the degree of x is same as the degree of the next x but not equal to the degree of a sibling of next x ; therefore, this case is not valid.

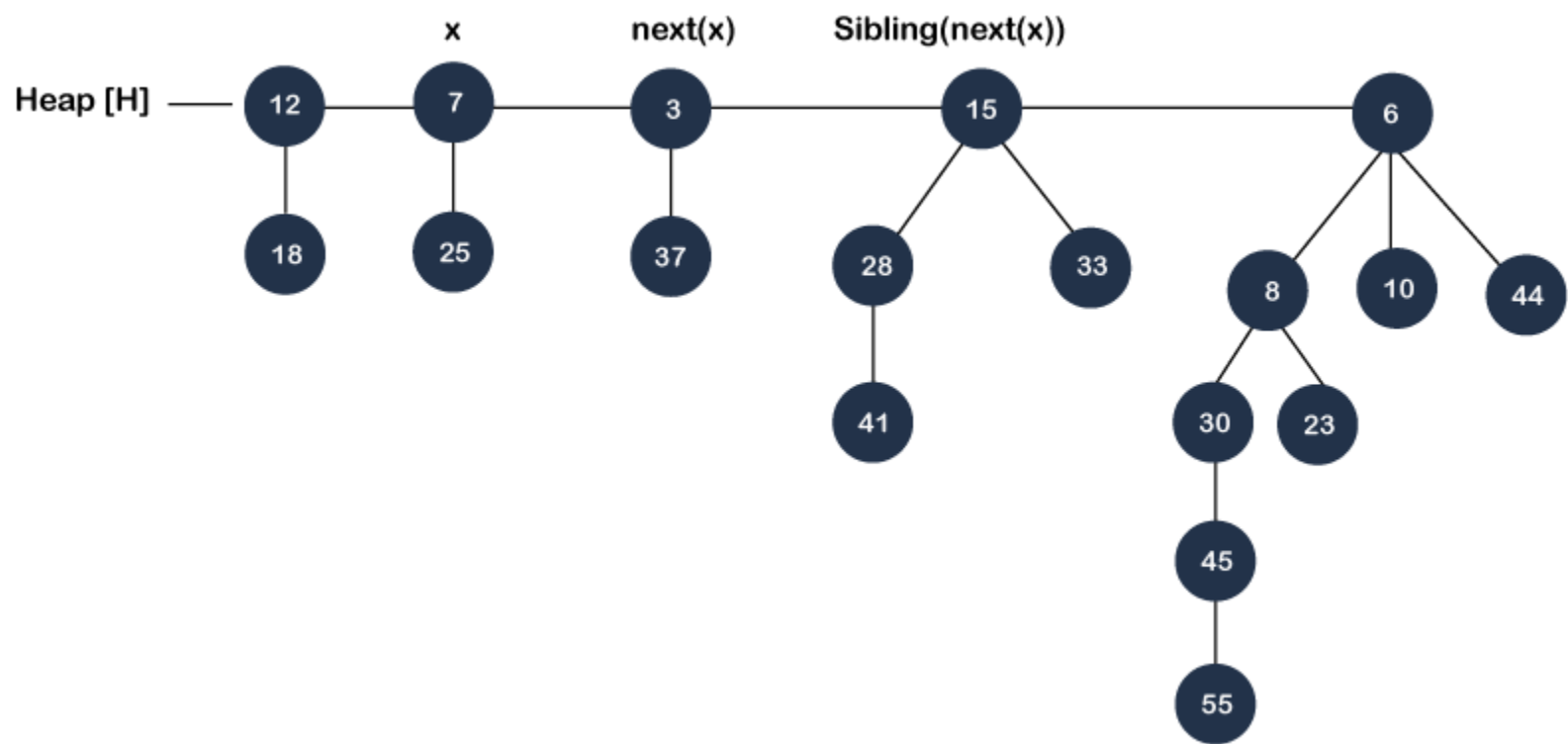
Now we will apply case 3. The case 3 says that **'If $\text{degree}[x] = \text{degree}[\text{next } x]$ but not equal to $\text{degree}[\text{sibling}[\text{next } x]]$ and $\text{key}[x] < \text{key}[\text{next } x]$ then remove $[\text{next } x]$ from root and attached to x '**. In the above example, the degree of x is equal to the degree of next x but equal to the degree of a sibling of next x , and the key value of x , i.e., 12, is less than the value of next x , i.e., 18; therefore, this case is valid. So, we have to remove the next x , i.e., 18, from the root and attach it to the x , i.e., 12, shown as below:



As we can observe in the above binomial heap that node 18 is attached to the node 12.

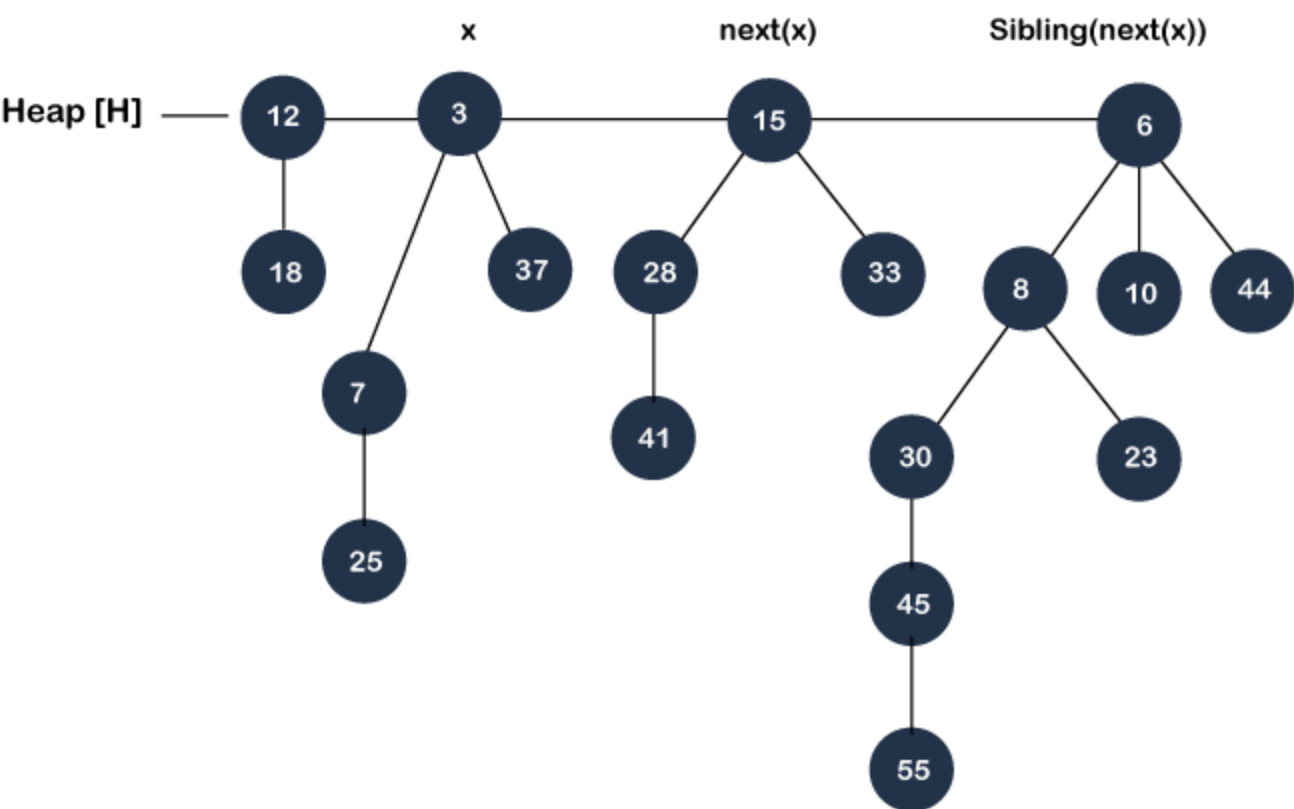
- Now we will reapply the cases in the above binomial heap. First, we will apply case 1. Since x is pointing to node 12 and next x is pointing to node 7, the degree of x is equal to the degree of next x ; therefore, case 1 is not valid.

Now we will apply case 2. Since the degree of x is equal to the degree of next x and also equal to the degree of sibling of next x ; therefore, this case is valid. We will move the pointer ahead shown as below:



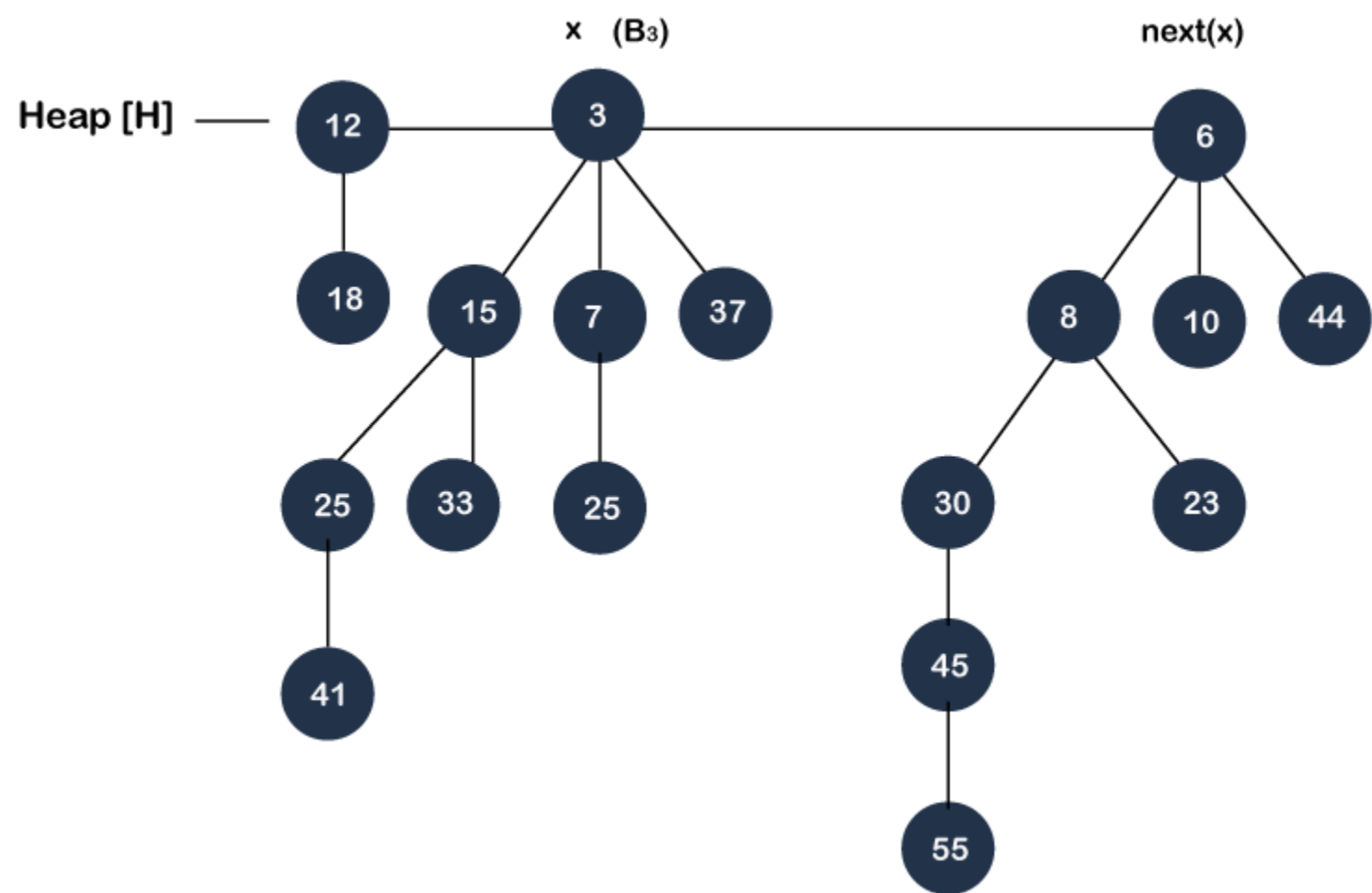
As we can observe in the above figure that ' x ' points to the binomial tree having root node 7, $next(x)$ points to the binomial tree having root node 3 while $prev(x)$ points to the binomial tree having root node 12. The $sibling(next(x))$ points to the binomial tree having root node 15.

- Now we will apply case 3 in the above tree. Since the degree of x is equal to the degree of next x , i.e., 1 but not equal to the degree of a sibling of next x , i.e., 2. Either case 3 or case 4 is valid based on the second condition. The key value of x , i.e., 7 is greater than the key value of $next(x)$, i.e., 3; therefore, we can say that case 4 is valid. Here, we need to remove the x and attach it to the $next(x)$ shown as below:



As we can observe in the above figure that x becomes the left child of $next(x)$. The pointer also gets updated. Now, x will point to the binomial tree having root node 3, and degree is also changed to 2. The $next(x)$ points to the binomial tree having root node as 15, and the $sibling(next(x))$ will point to the binomial tree having root node as 6.

- In the above tree, the degree of x , i.e., B_2 , is the same as the degree of $next(x)$, i.e., B_2 , but not equal to the degree of $sibling(next(x))$, i.e., B_4 . Therefore, either case 3 or case 4 is valid based on the second condition. Since the key value of x is less than the value of $next(x)$ so we need to remove $next(x)$ and attach it to the x shown as below:

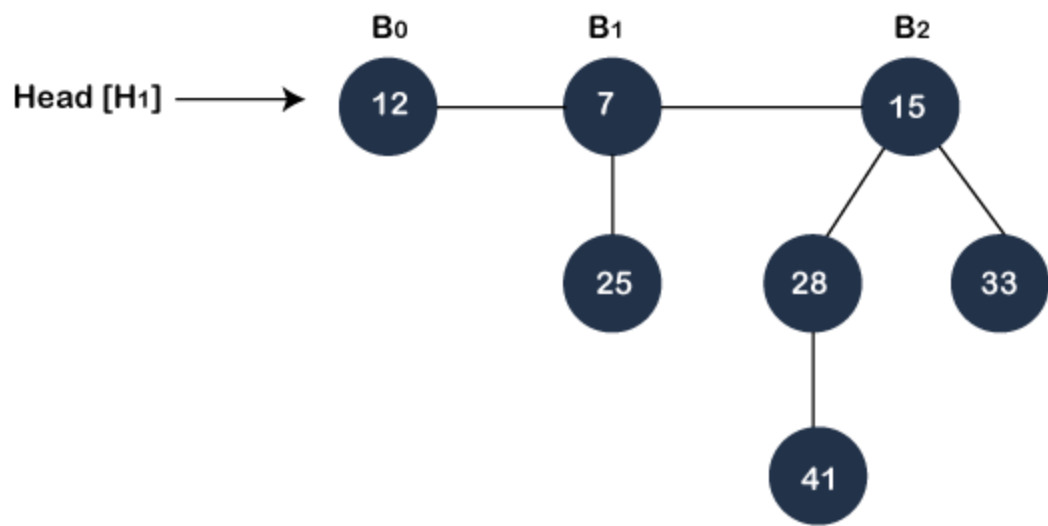


As we can observe in the above figure that $next(x)$ becomes the left child of x , and the degree of x also gets changed to B_3 . The pointer $next(x)$ also gets updated, and it now points to the binomial tree having root node 6. The degree of x is 3, and the degree of $next(x)$ is 4. Since the degrees of x and $next(x)$ are not equal, so case 1 is valid. We will move the pointer ahead, and now x points to node 6.

The B_4 is the last binomial tree in a heap, so it leads to the termination of the loop. The above tree is the final tree after the union of two binomial heaps.

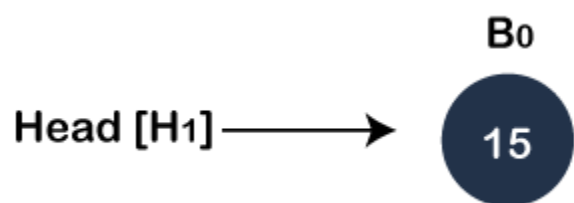
Inserting a new node in a Binomial heap

Now we will see how to insert a new node in a heap. Consider the below heap, and we want to insert a node 15 in a heap.

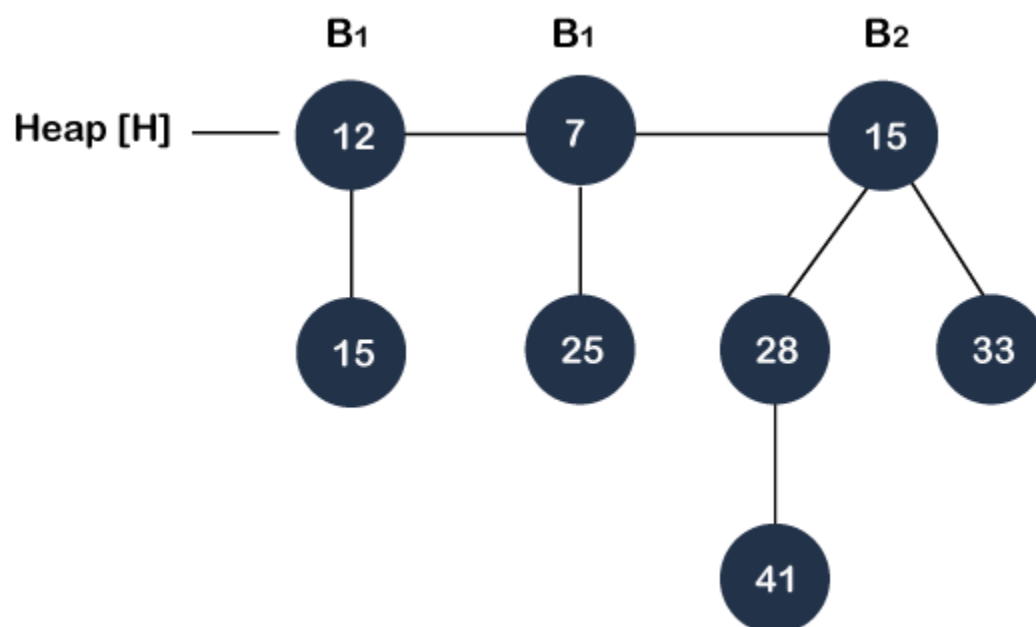
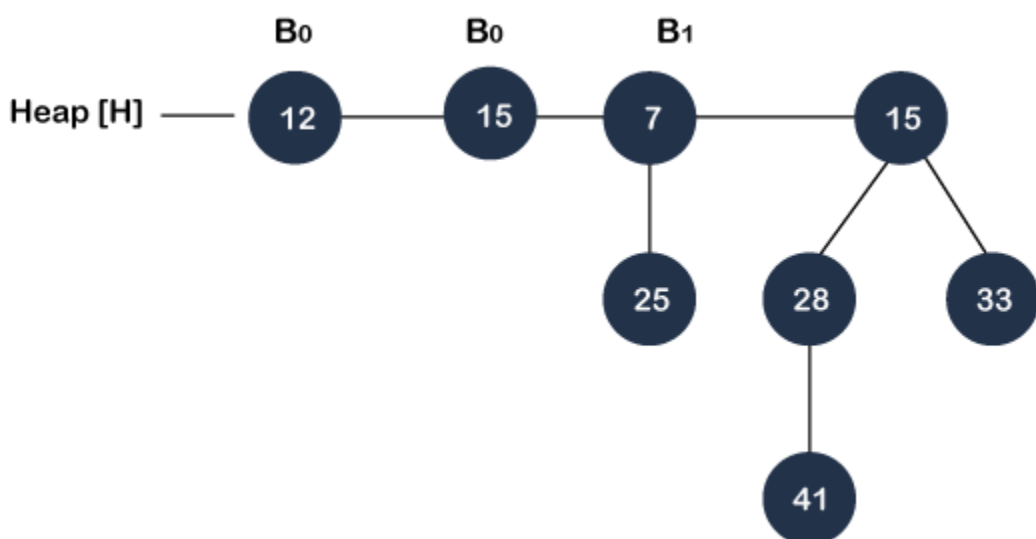


In the above heap, there are three binomial trees of degree B_0 , B_1 , and B_2 , where B_0 is attached to the head of the heap.

Let's assume that node 15 is attached to the head of the heap shown as below:



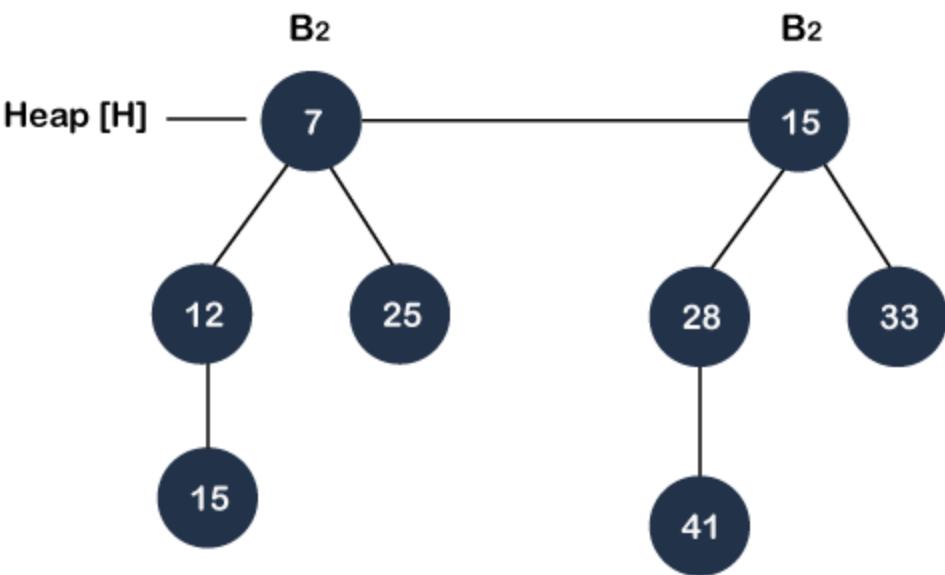
First, we will combine these two heaps. As the degree of node 12 and node 15 is B_0 so node 15 is attached to the node 12 shown in the above figure.



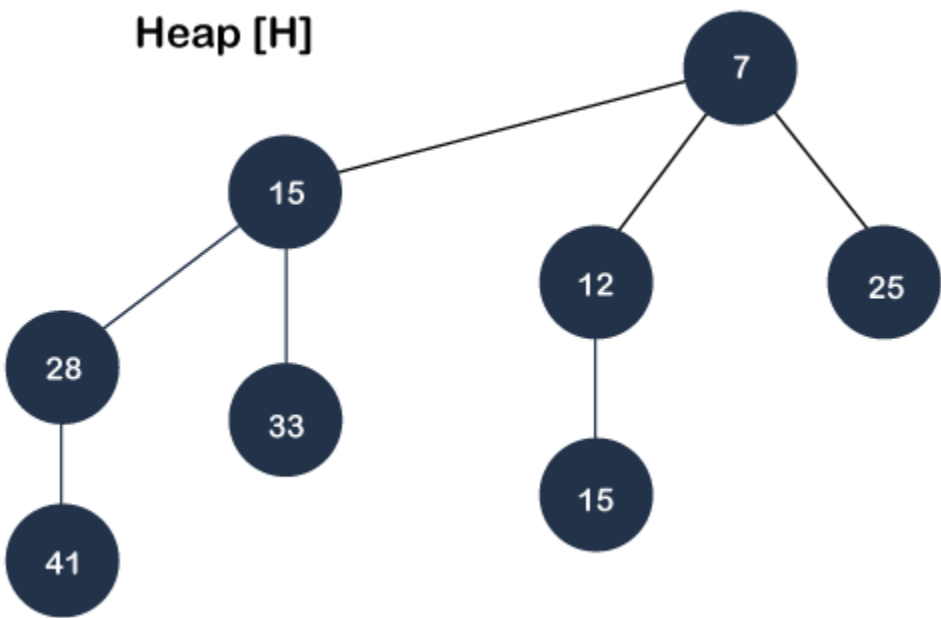
Now we assign 'x' to the B_0 having key value 12, $\text{next}(x)$ to the B_0 having key value 15, and $\text{sibling}(\text{next}(x))$ to the B_1 having key value 7.

Since the degree of x is equal to the degree of $\text{next}(x)$ but not equal to the degree of $\text{sibling}(\text{next}(x))$ so either case 3 or case 4 will be applied to the heap. The key value of x is greater than the key value of $\text{next}(x)$; therefore, $\text{next}(x)$ would be removed and attached it to the x.

Now, we will reapply the cases in the above heap. Now, x points to the node 12 having degree B_1 , $\text{next}(x)$ points to the node 7 having degree B_1 and $\text{sibling}(\text{next}(x))$ points to the node 15 having degree B_2 . Since the degree of x is same as the degree of $\text{next}(x)$ but not equal to the degree of $\text{sibling}(\text{next}(x))$, so either case 3 or case 4 is applied. The key value of x is greater than the key value of $\text{next}(x)$; therefore, the x is removed and attached to the x shown as below:



As we can observe in the above figure that when 'x' is attached to the next(x) then the degree of node 7 is changed to B₂. The pointers are also get updated. Now 'x' points to the node 7 and next(x) points to the node 15. Now we will reapply the cases in the above heap. Since the degree of x is equal to the degree of next(x) and the key value of x is less than the key value of next(x); therefore, next(x) would be removed from the heap and attached it to the x shown as below:

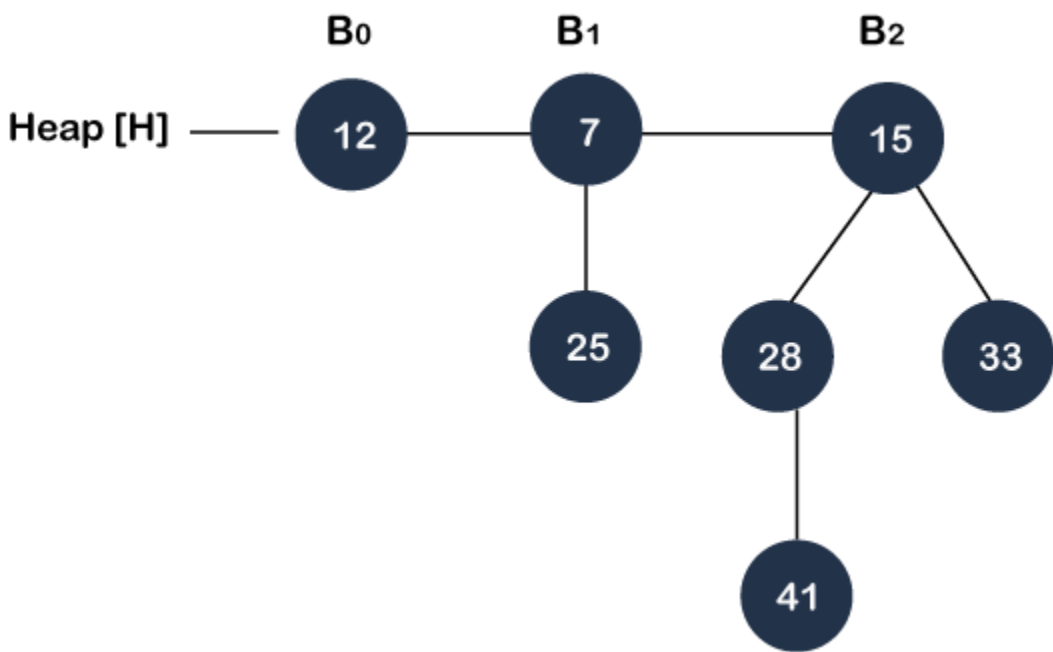


As we can observe in the above heap that the degree of x gets changed to 3, and this is the final binomial heap after inserting the node 15.

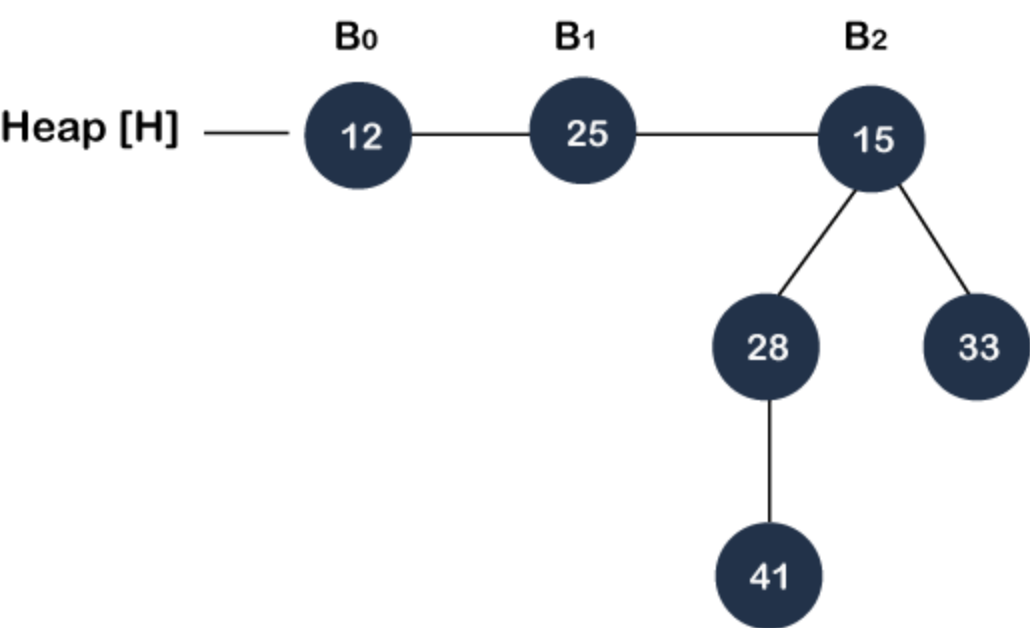
Extracting a minimum key

Here extracting a minimum key means that we need to remove the element which has minimum key value. We already know that in min heap, the root element contains the minimum key value. Therefore, we have to compare the key value of root node of all the binomial trees.

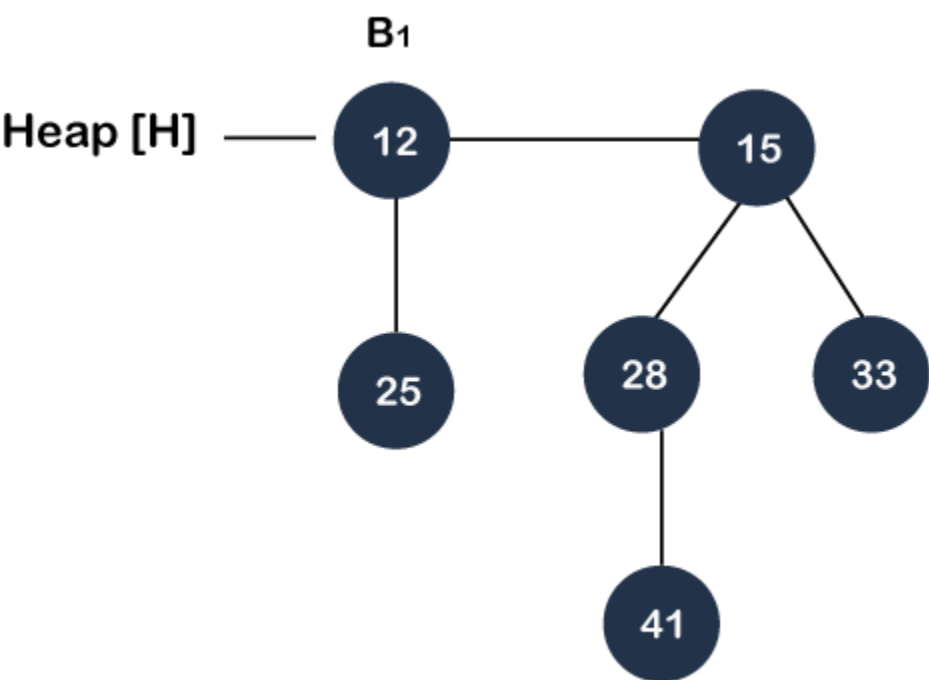
Consider the binomial heap which is given below:



In the above heap, the key values of root node of all the binomial trees are 12, 7 and 15. The key value 7 is the minimum value so we will remove the node 7 from the tree shown as below:



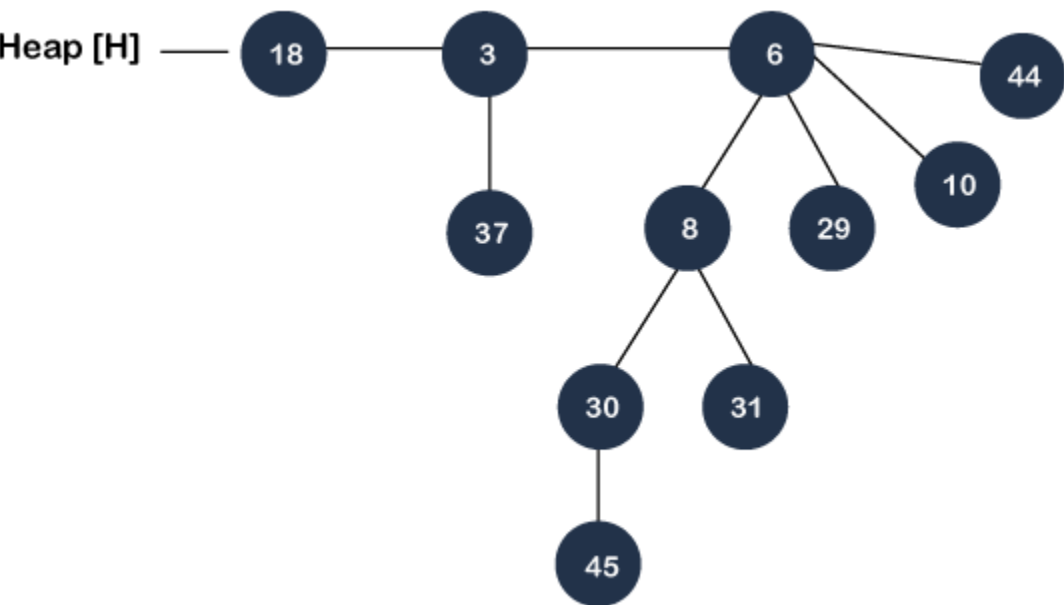
As we can observe in the above binomial heap that the degree of node 12 is B_0 , degree of node 25 is B_0 , and degree of node 15 is B_2 . The pointer x points to the node 12, $\text{next}(x)$ points to the node 25, and $\text{sibling}(\text{next}(x))$ points to the node 15. Since the degree of x is equal to the degree of $\text{next}(x)$ but not equal to the degree of $\text{sibling}(\text{next}(x))$; therefore, either case 3 or case 4 will be applied to the above heap. The key value of x is less than the key value of $\text{next}(x)$, so node 25 will be removed and attached to the node 12 shown as below:



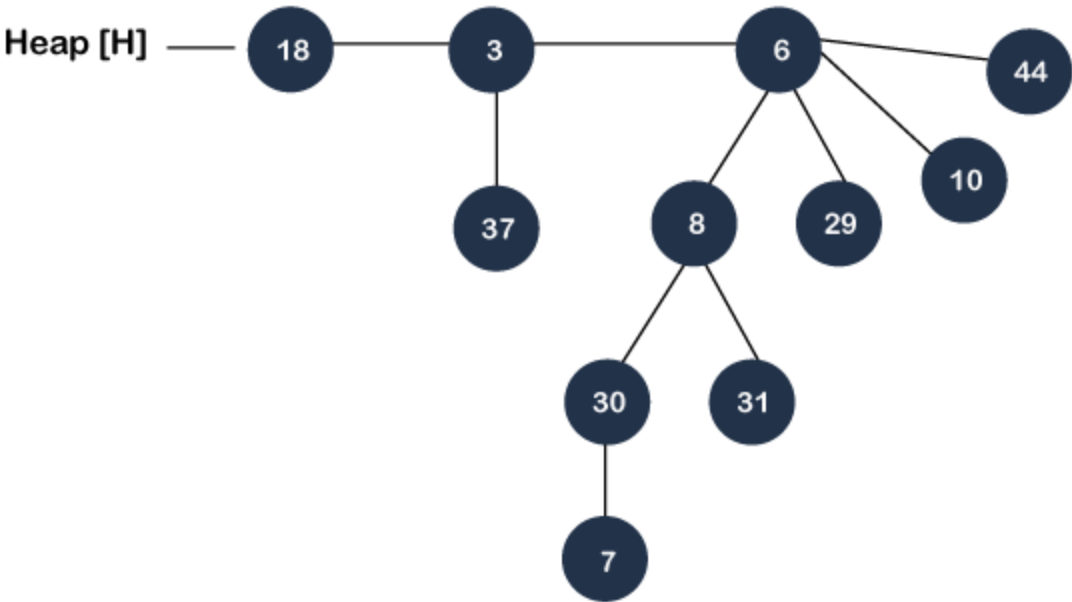
The degree of node 12 is also changed to 1.

Decreasing a key

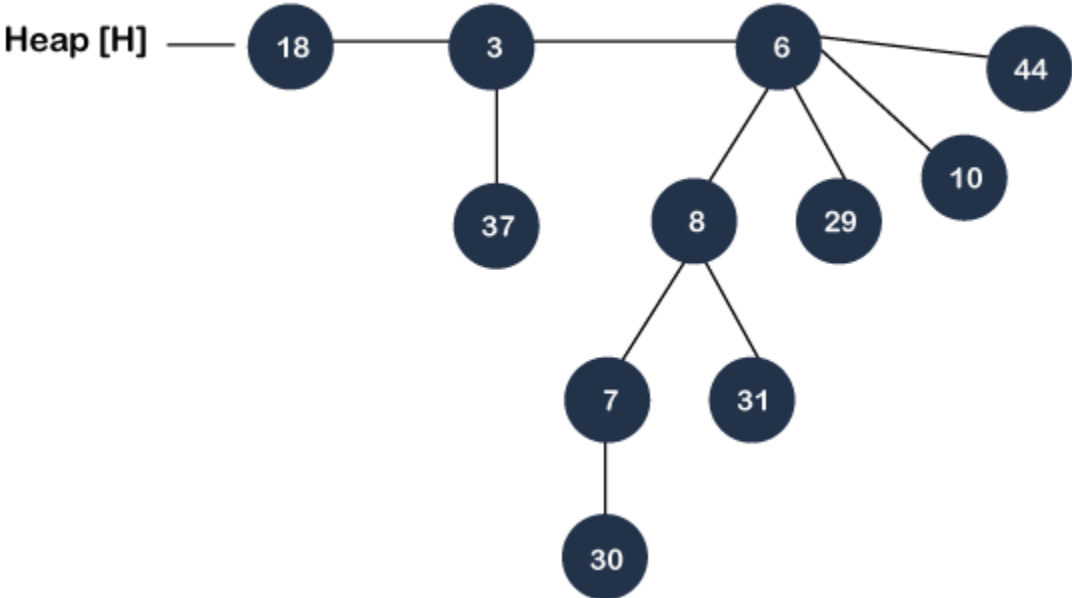
Now we will see how to decrease a key with the help of an example. Consider the below heap, and we will decrease the key 45 by 7:



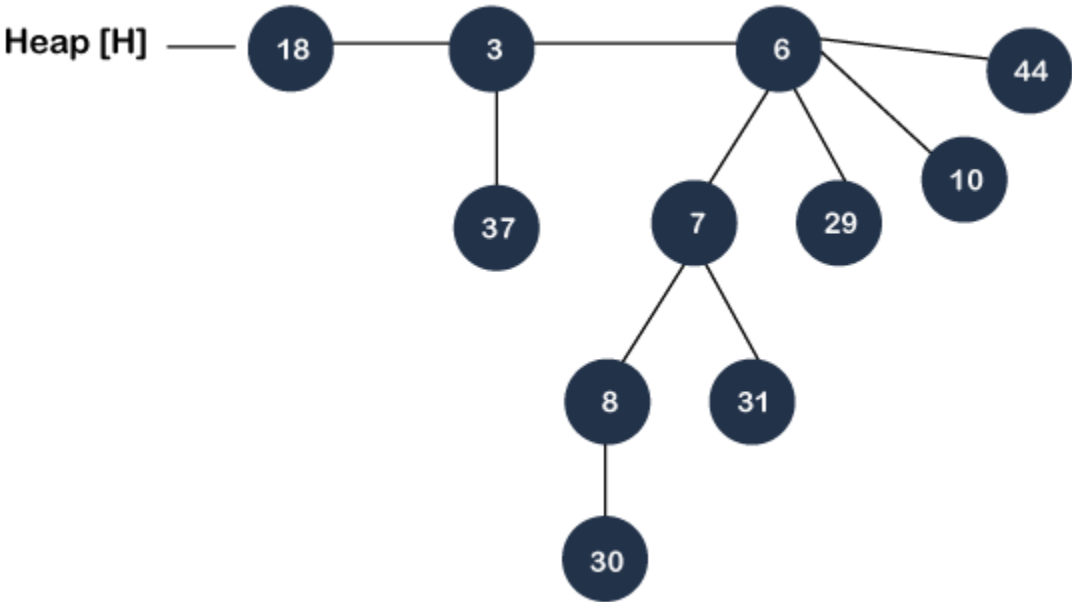
After decreasing the key by 7, the heap would look like:



Since the above heap does not satisfy the min-heap property, element 7 will be compared with an element 30; since the element 7 is less than the element 30 so 7 will be swapped with 30 element shown as below:



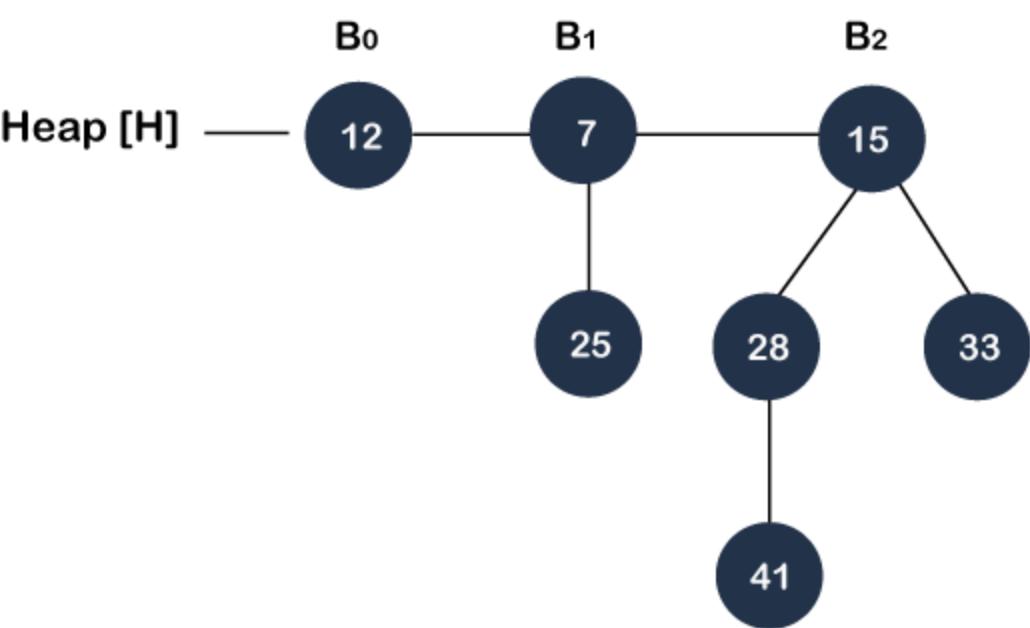
Now we will again compare element 7 with its root element, i.e., 8. Since element 7 is less than element 8 so element 7 will be swapped with an element 8 shown as below:



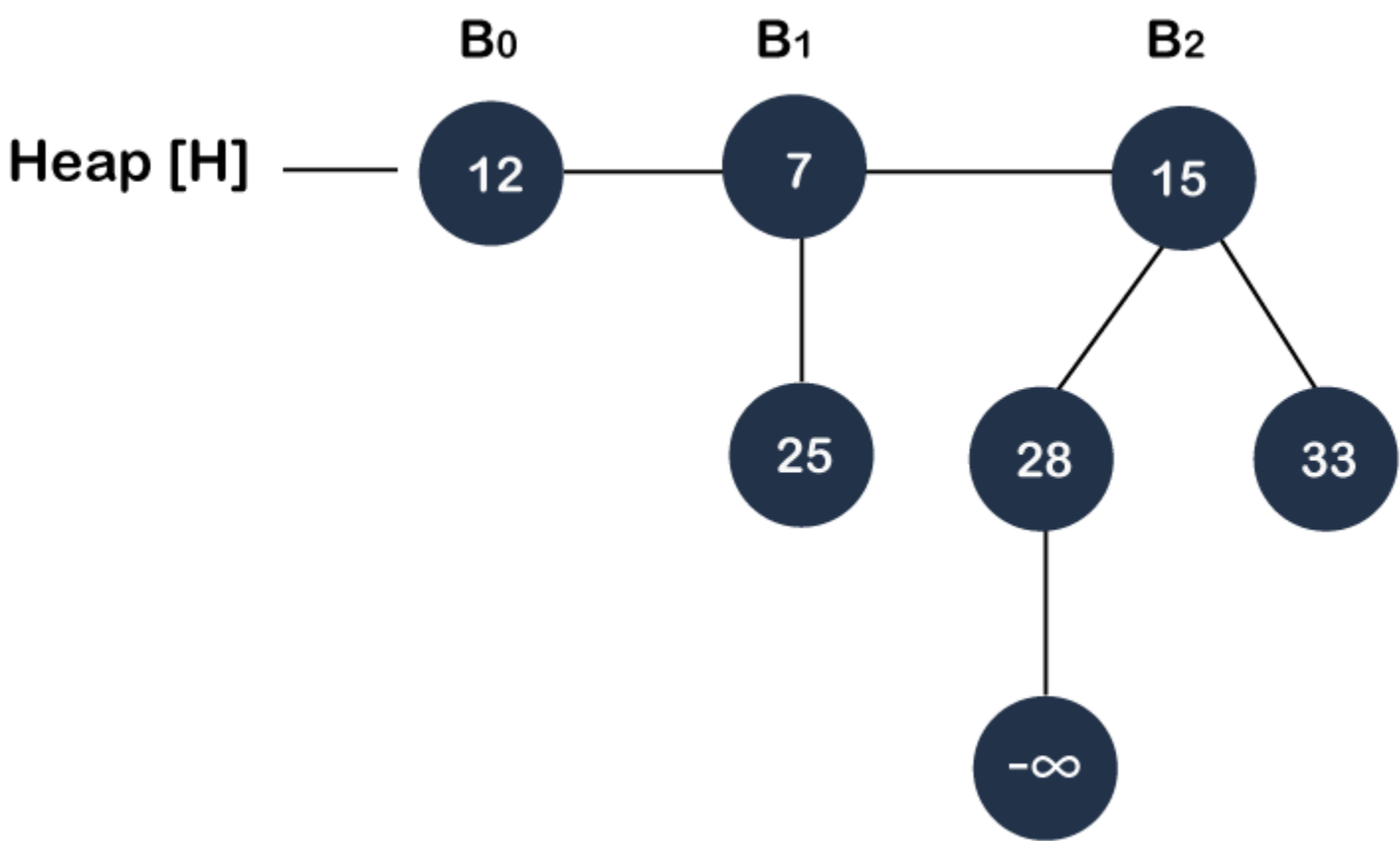
Now, the above heap satisfies the property of the min-heap.

Deleting a node

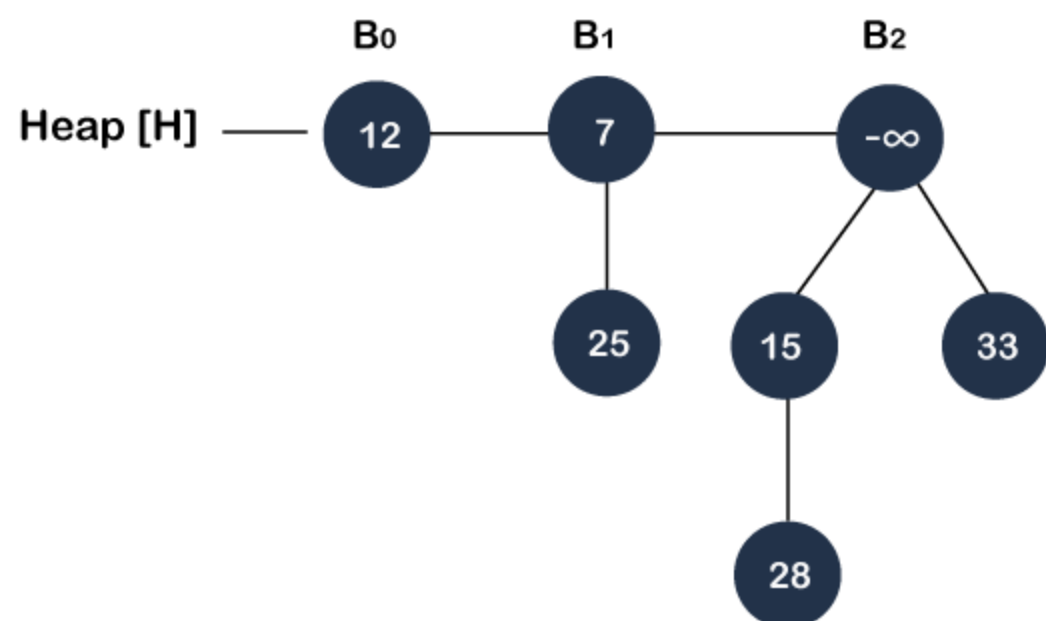
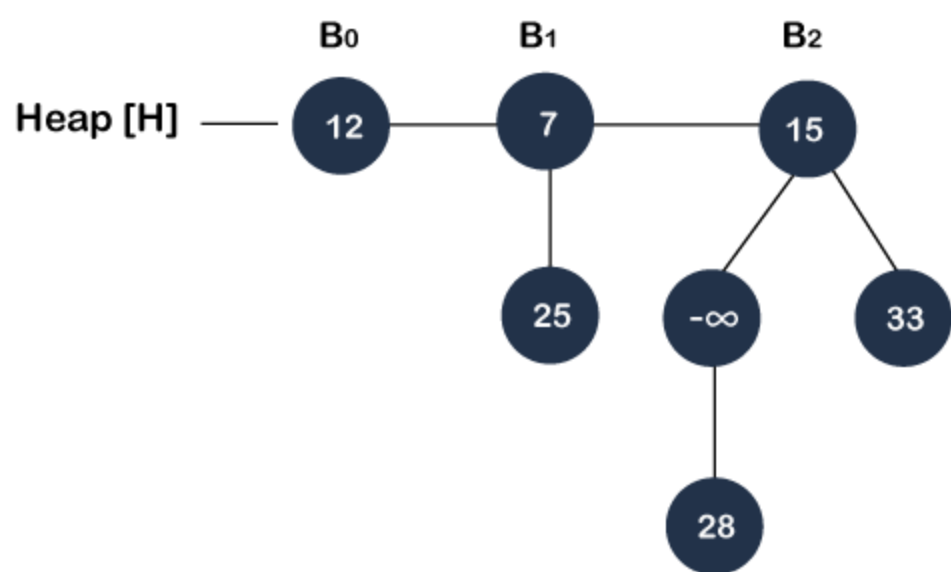
Now we will see how to delete a node with the help of an example. Consider the below heap, and we want to delete node 41:



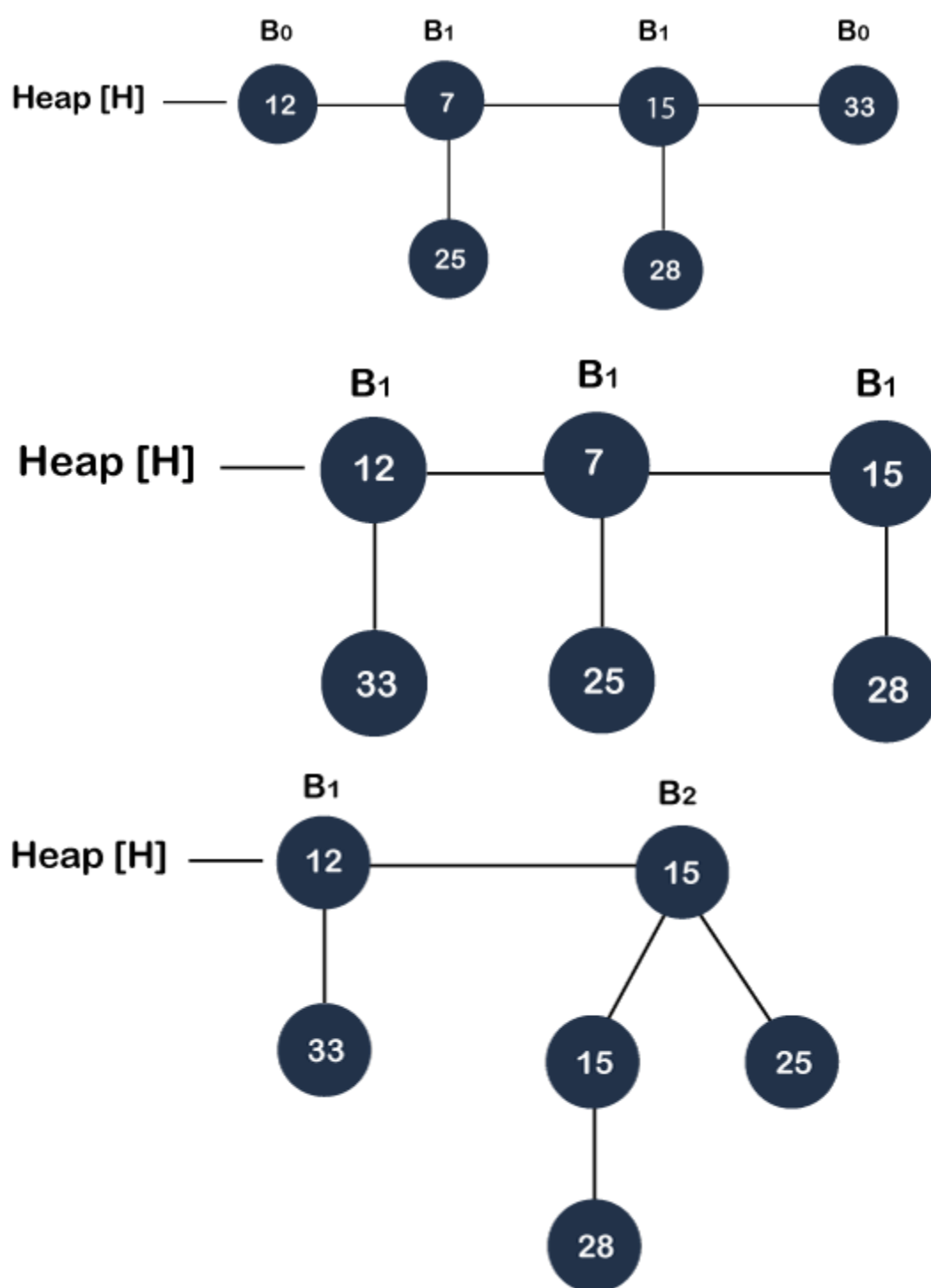
First, we replace node 41 with the smallest value, and the smallest value is -infinity, shown as below:



Now we will swap the -infinity with the root node of the tree shown as below:



The next step is to extract the minimum key. Since the minimum key in a heap is -infinity so we will extract this key, and the heap would be:



Postorder Traversal

The postorder traversal is one of the traversing techniques used for visiting the node in the tree. It follows the principle LRN (Left-right-node). The following are the steps used for the postorder traversal:

- Traverse the left subtree by calling the postorder function recursively.
- Traverse the right subtree by calling the postorder function recursively.
- Access the data part of the current node.

Pseudocode of postorder traversal.

1. **void** postorder(root *r)
2. {
3. **if**(r== NULL)
4. then **return**;
5. **else**
6. postorder(r->left);
7. postorder(r->right);
8. print r->data;

In the above pseudo-code, we have defined a postorder function in which we added the logic of postorder traversal. As we already know that in postorder traversal, we first traverse the left subtree, then the right subtree and finally visit the root node. In the above code, we first have checked whether the value of root is NULL or not. If the value of root is NULL, then the control comes out of the function using the return statement. The 'if' condition is not satisfied, then the else part will be executed.

Implementation of Postorder traversal in C.

```

1. #include<stdio.h>
2. // Creating a node structure
3.     struct node
4.     {
5.         int data;
6.         struct node *left, *right;
7.     };
8.
9. // Creating a tree..
10. struct node* create()
11. {
12.     struct node *temp;
13.     int data;
14.     temp = (struct node *)malloc(sizeof(struct node)); // creating a new node
15.     printf("\nEnter the data: ");
16.     scanf("%d", &data);
17.
18.     if(data== -1)
19.     {
20.         return NULL;
21.     }
22.     temp->data = data;
23.     printf("\nEnter the left child of %d", data);
24.     temp->left = create(); // creating a left child
25.     printf("\nEnter the right child of %d", data);
26.     temp->right = create(); // creating a right child
27.     return temp;
28. }
29.
30. //Postorder traversal
31. void postorder(struct node *root)
32. {
33.     if(root==NULL)
34.         return;
35.     else
36.
37.         postorder(root->left);
38.         postorder(root->right);
39.         printf("%d ", root->data);
40. }
41. void main()
42. {
43.     struct node *root;
44.     root = create(); // calling create function.
45.     postorder(root); // calling postorder function
46. }

```

Output

```
Enter the data: 45

Enter the left child of 45
Enter the data: 34

Enter the left child of 34
Enter the data: 60

Enter the left child of 60
Enter the data: -1

Enter the right child of 60
Enter the data: -1

Enter the right child of 34
Enter the data: -1

Enter the right child of 45
Enter the data: -1
60 34 45
```

Iterative Postorder traversal

The above method that we have used for the Postorder traversal was the Recursion method. As we know that the recursion relies on the system stack, and if we want to convert the recursion into an iterative method, we need to use the external stack. There are two ways of iterative postorder traversal:

- Iterative postorder traversal using two stacks.
- Iterative postorder traversal using one stack

Iterative Postorder traversal using two stacks

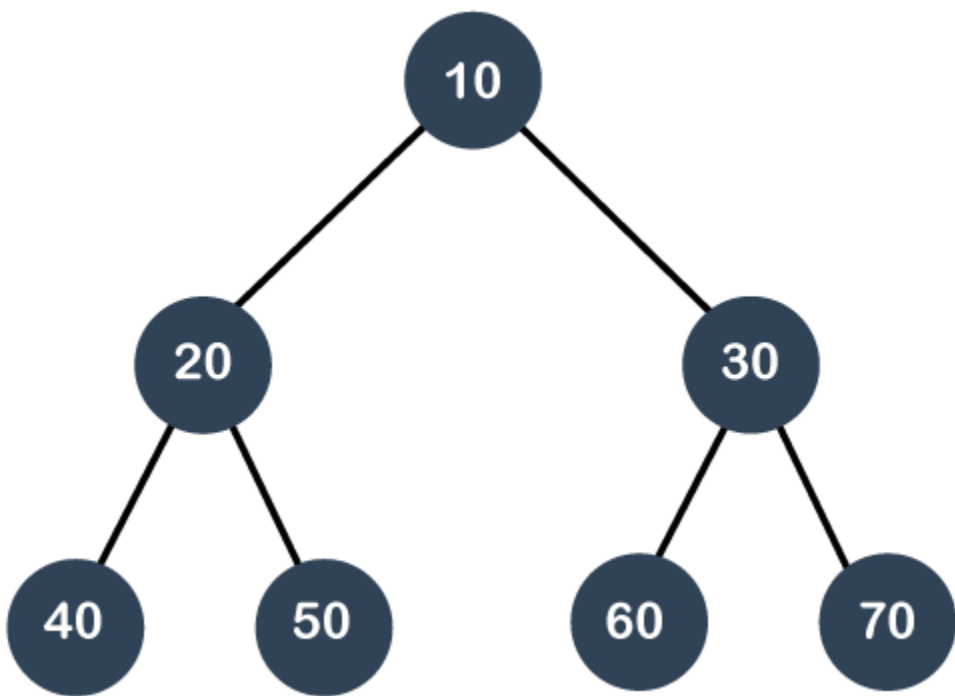
We can also implement the Postorder traversal using two stacks. The idea behind this is to obtain the reversed postorder elements in the stack and then we pop the element one by one from the stack as the stack follows the LIFO principle. The question arises that how can we obtain the reversed order of the postorder elements. The solution to this problem can be obtained by using two stacks. The second stack is used to get the reversed order of the postorder elements.

The following are the steps used to implement postorder using two stacks:

- First, we will push the root to the stack.
- Secondly, we will create a while loop that runs till the first stack is not empty
 1. We pop a node from the first stack and push it into the second stack.
 2. We push the left and right child of a popped node to the first stack.
- We will print the data of the second stack.

Let's understand the postorder traversal using two stacks through an example.

Consider the below tree for the Postorder traversal.



Steps to be followed:

Step 1: First, we push 10 into stack1.

Stack1: 10

Stack2: empty

Step 2: Now we pop element 10 from stack1 and push it into stack2. We will push the left and right child of element 10 into stack1.

Stack1: 20, 30

Stack2: 10

Step 3: Now, we pop element 30 from stack1 and push it into stack2. We will push the left and right child of element 30 into stack1:

Stack 1: 20, 60, 70

Stack 2: 10, 30

Step 4: Now we pop the element 70 from the stack1 and push it into the stack2. We will push the left and right child of element 70 into the stack1. Since there is no left and right child of element 70, so we will not push the element into Stack1.

Stack1: 20, 60

Stack2: 10, 30, 70

Step 5: Now we pop the element 60 from the Stack1 and push it into Stack2. Since there is no left and right of element 60, so we will not push any element into Stack1.

Stack1: 20

Stack2: 10, 30, 70, 60

Step 6: Now, we pop the element 20 from Stack1 and push it into Stack2. We will push the left and right child of element 20 into the Stack1.

Stack1: 40, 50

Stack2: 10, 30, 70, 60, 20

Step 7: Now, we pop the element 50 from Stack1 and push it into Stack2.

Stack1: 40

Stack2: 10, 30, 70, 60, 20, 50

Step 8: Now, we pop element 40 from the Stack1 and push it into Stack2.

Stack1: Empty

Stack2: 10, 30, 70, 60, 20, 50, 40

Since there is no element left in the Stack1 to be popped out and all the elements in Stack2 are in postorder fashion, so we will print them.

Algorithm of iterative postorder traversal (two stacks)

1. Step 1: Create two empty stacks named stack1 and stack2.
2. Step 2: Push the root node on stack1.
3. Step 3: `TreeNode* current = NULL`
4. Step 4: **while**(!stack1.empty())
5. `current = top of stack1`
6. `pop current from stack1`
7. Push current on stack1
8. **if** (current->left!= Null)
9. Push current->left on stack1
10. **if**(current->right!=NULL)

11. Push current->right on stack1
12. Step 5: **while**(!stack2.empty())
13. current = top of stack2
14. print current->value
15. pop current from the stack2

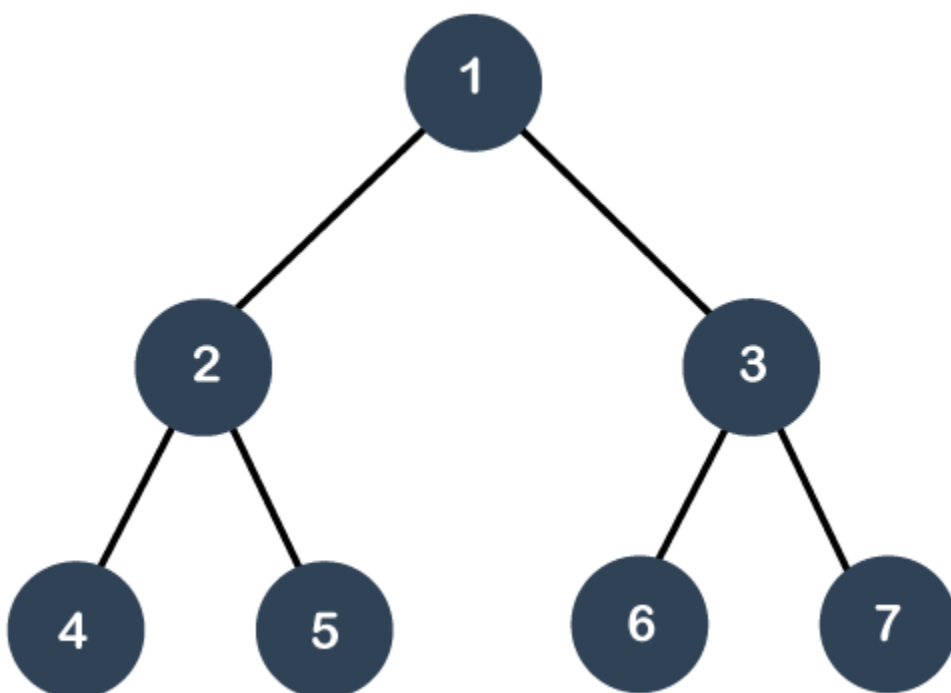
Iterative Postorder Traversal using one stack

In order to implement the Postorder traversal using one stack, first we need to move down till we reach the leftmost node. While moving down to the tree, we need to push the root and then right child of the root into the stack. When we reach the leftmost node having no right child, we need to print the node. If the leftmost node has a right child, we have to update the root so that the right child can be traversed before.

We have to perform three operations:

- If we are moving down the tree, push operation is performed.
- If we are coming up from the left-side, peek operation is performed
- If we are coming from the right-side, pop operation is performed

Let's understand through an example.



Step 1: The right child of node 1 exists. Push node 3 to the stack and then 1 to the stack.

Stack: 3, 1

Step 2: The right child of node 2 exists. Push node 5 to the stack and then 2 to the stack.

Stack: 3, 1, 5, 2

Step 3: The node 4 has no right child, so it gets printed and the current node is set to Null.

Stack: 3, 1, 5, 2

Output: 4

Step 4: Now, peek operation will be performed on the node 2. Since the right child of node 2 is topmost element in the stack, i.e., 5. So, pop the element 5 from the stack.

Stack: 3, 1, 2

Step 5: The right child of node 2 is 5, so we push 5 into the stack.

Stack: 3, 1, 2, 5

Step 6: Since node 5 does not have a right child so pop 5 from the stack and print 5. The current node is now set to NULL.

Stack: 3, 1, 2

Output: 4, 5

Step 7: The current node is NULL. Since the right child of node 2 is not the topmost element in the stack, so we will pop element 2 from the stack and print 2. The current node is now set to NULL.

Stack: 3, 1

Output: 4, 5, 2

Step 8: The current node is NULL. Since the right child of node 1 is 3 so we will perform the peek operation on node 1 and pop 3 from the stack.

Stack: 1

Step 9: The right child of 1 is 3 so we will push element 3 in the stack.

Stack: 1, 3

Step 10: The node 3 has a left child, i.e., 6, so it gets printed and the right child of node 3, i.e., 7 is pushed into the stack.

Stack: 1, 3, 7

Output: 4, 5, 2, 6

Step 11: Since there is no right child of node 7, so pop 7 from the stack.

Stack: 1, 3

Output: 4, 5, 2, 6, 7

Step 12: Since the right child of node 3 is not the topmost element in the stack, so pop 3 from the stack.

Stack: 1

Output: 4, 5, 2, 6, 7, 3

Step 12: Since the right child of node 1 is not the topmost element in the stack, so Pop 1 from the stack.

Stack: empty

Output: 4, 5, 2, 6, 7, 3, 1

The above output generated is the Postorder traversal.

Algorithm of iterative postorder traversal (one stack)

1. Step 1: Create one empty stack.
2. Step 2: Set current_node = root
3. Step 3: **while**(current_node!=NULL)
4. Push current_node->right into the stack
5. Push current_node into the stack
6. Step 4: Pop an element from the stack and set it as a current node.
7. If the popped element has a right child and the right child is top element then remove the right child from the stack
 . Push the current node back to the stack.
8. Otherwise print the current node's data and set the current node as NULL.
9. Step 5: Repeat step 3 and step 4 till the stack is not empty.

Sparse Matrix

A matrix can be defined as a two-dimensional array having 'm' columns and 'n' rows representing m*n matrix. Sparse matrices are those matrices that have the majority of their elements equal to zero. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

Why do we need to use a sparse matrix instead of a simple matrix?

We can also use the simple matrix to store the elements in the memory; then why do we need to use the sparse matrix. The following are the advantages of using a sparse matrix:

- **Storage:** As we know, a sparse matrix that contains lesser non-zero elements than zero so less memory can be used to store elements. It evaluates only the non-zero elements.

- **Computing time:** In the case of searching n sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. The zeroes in the matrix are of no use to store zeroes with non-zero elements. To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.

Sparse Matrix Representation

The non-zero elements can be stored with triples, i.e., rows, columns, and value. The sparse matrix can be represented in the following ways:

- Array representation
- Linked list representation

Array Representation

The 2d array can be used to represent a sparse matrix in which there are three rows named as:

1. Row: It is an index of a row where a non-zero element is located.
2. Column: It is an index of the column where a non-zero element is located.
3. Value: The value of the non-zero element is located at the index (row, column).

Let's understand the sparse matrix using array representation through an example.

Sparse matrix →

	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	3	0	0
4	0	0	0	0

As we can observe above, that sparse matrix is represented using triplets, i.e., row, column, and value. In the above sparse matrix, there are 13 zero elements and 7 non-zero elements. This sparse matrix occupies $5 \times 4 = 20$ memory space. If the size of the sparse matrix is increased, then the wastage of memory space will also be increased. The above sparse matrix can be represented in the tabular form shown as below:

Table Structure

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
3	1	3

In the above table structure, the first column is representing the row number, the second

column is representing the column number and third column represents the non-zero value at index(row, column). The size of the table depends upon the number of non-zero elements in the sparse matrix. The above table occupies $(7 \times 3) = 21$ but it more than the sparse matrix. Consider the case if the matrix is **8*8** and there are only 8 non-zero elements in the matrix then the space occupied by the sparse matrix would be $8 \times 8 = 64$ whereas, the space occupied by the table represented using triplets would be $8 \times 3 = 24$.

In the 0th row and 1nd column, 4 value is available. In the 0th row and 3rd column, value 5 is stored. In the 1st row and 2nd column, value 3 is stored. In the 1st row and 3rd column, value 6 is stored. In 2nd row and 2nd column, value 2 is stored. In the 3rd row and 0th column, value 2 is stored. In the 3rd row and 1st column, value 3 is stored.

Array implementation of sparse matrix in C

```
1. #include <stdio.h>
2. int main()
3. {
4.     // Sparse matrix having size 4*5
5.     int sparse_matrix[4][5] =
6.     {
7.         {0 , 0 , 7 , 0 , 9 },
8.         {0 , 0 , 5 , 7 , 0 },
9.         {0 , 0 , 0 , 0 , 0 },
10.        {0 , 2 , 3 , 0 , 0 }
11.    };
12.    // size of matrix
13.    int size = 0;
14.    for(int i=0; i<4; i++)
15.    {
16.        for(int j=0; j<5; j++)
17.        {
18.            if(sparse_matrix[i][j]!=0)
19.            {
20.                size++;
21.            }
22.        }
23.    }
24.    // Defining final matrix
25.    int matrix[3][size];
26.    int k=0;
27.    // Computing final matrix
28.    for(int i=0; i<4; i++)
29.    {
30.        for(int j=0; j<5; j++)
31.        {
32.            if(sparse_matrix[i][j]!=0)
33.            {
34.                matrix[0][k] = i;
35.                matrix[1][k] = j;
36.                matrix[2][k] = sparse_matrix[i][j];
37.                k++;
38.            }
39.        }
40.    }
41.    // Displaying the final matrix
42.    for(int i=0 ;i<3; i++)
43.    {
44.        for(int j=0; j<size; j++)
45.        {
46.            printf("%d ", matrix[i][j]);
47.            printf("\t");
48.        }
49.        printf("\n");
50.    }
51.    return 0;
52.}
```

Output

```
0 0 1 1 3 3
2 4 2 3 1 2
7 9 5 7 2 3

...Program finished with exit code 0
Press ENTER to exit console.
```

Linked List Representation

In linked list representation, linked list data structure is used to represent a sparse matrix. In linked list representation, each node consists of four fields whereas, in array representation, there are three fields, i.e., row, column, and value. The following are the fields in the linked list:

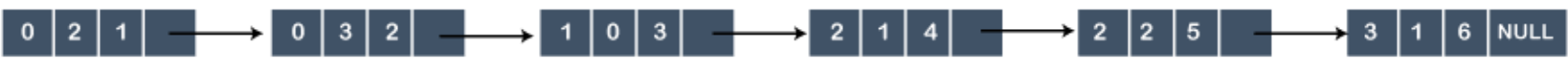
- Row: It is an index of row where a non-zero element is located.
- Column: It is an index of column where a non-zero element is located.
- Value: It is the value of the non-zero element which is located at the index (row, column).
- Next node: It stores the address of the next node.

Let's understand the sparse matrix using linked list representation through an example.

Sparse matrix →

	0	1	2	3
0	0	0	1	2
1	3	0	0	0
2	0	4	5	0
3	0	6	0	0

Linked List Representation

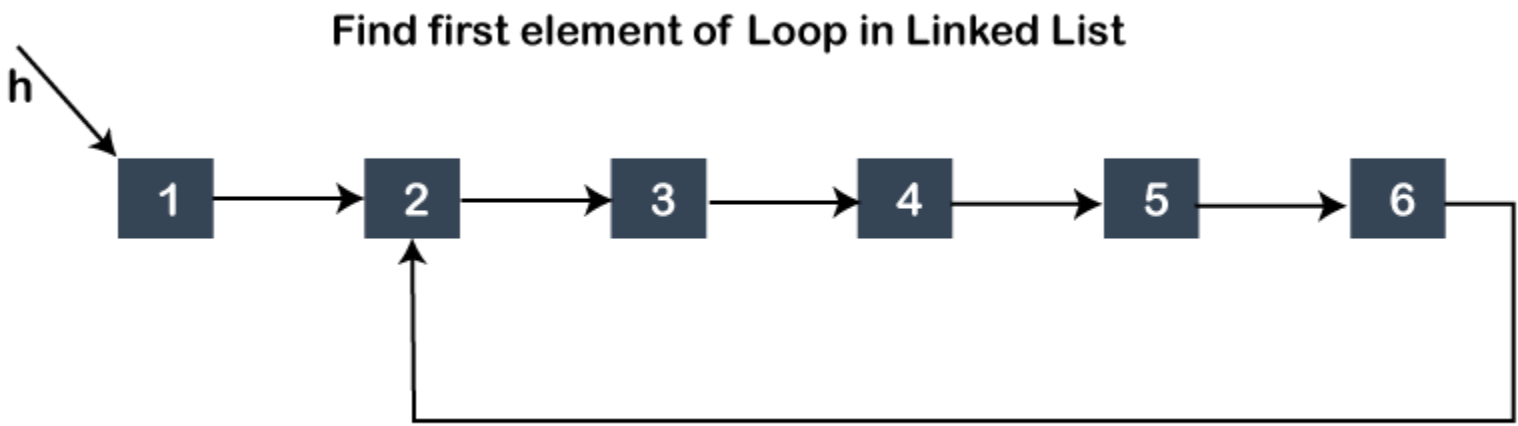


In the above figure, sparse represented in the linked list form. In the node, first field represents the index of row, second field represents the index of column, third field represents the value and fourth field contains the address of the next node.

Detect loop in a Linked list

A loop in a linked list is a condition that occurs when the linked list does not have any end. When the loop exists in the linked list, the last pointer does not point to the Null as observed in the singly linked list or doubly linked list and to the head of the linked list observed in the **circular linked list**. When the loop exists, it points to some other node, also known as the linked list cycle.

Let's understand the loop through an example.



In the above figure, we can observe that the loop exists in the linked list. Here, the problem statement is that we have to detect the node, which is the start of the loop. The solution to solve this problem is:

- First, we detect the loop in the linked list.
- Detect the start node of the loop.

Detecting a loop

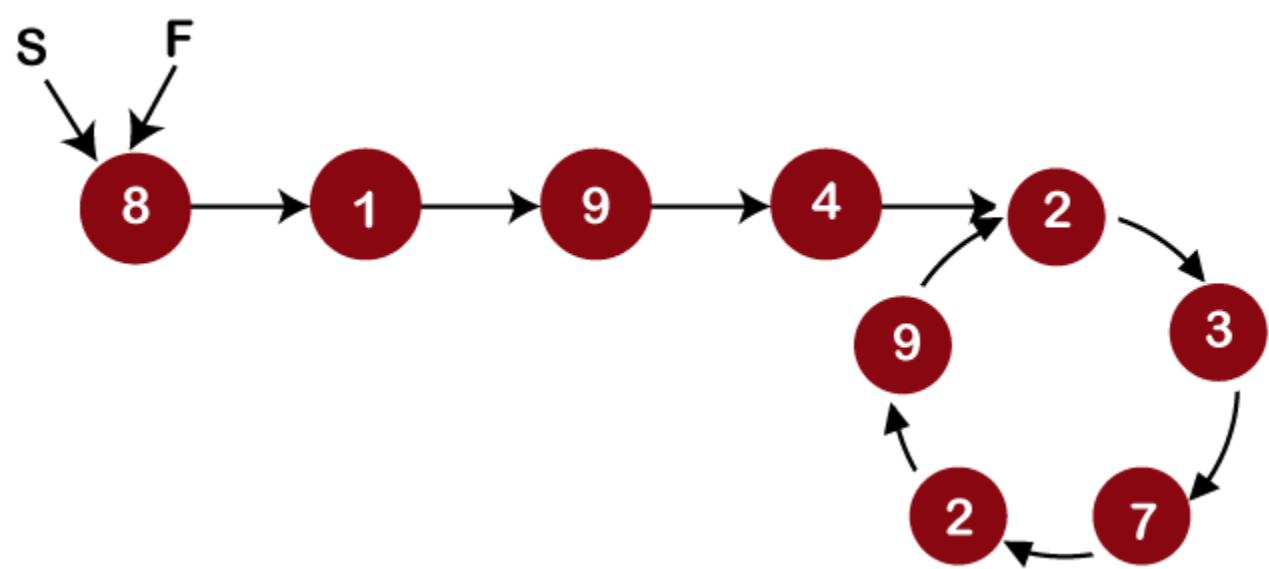
First, we will detect a loop in the linked list. To understand this, we will look at the algorithm for detecting a loop.

Step 1: First, we will initialize two pointers, i.e., S as a slow pointer and F as a fast pointer. Initially, both the pointers point to the first node in the linked list.

Step 2: Move the 'S' pointer one node at a time while move the 'F' pointer two nodes at a time.

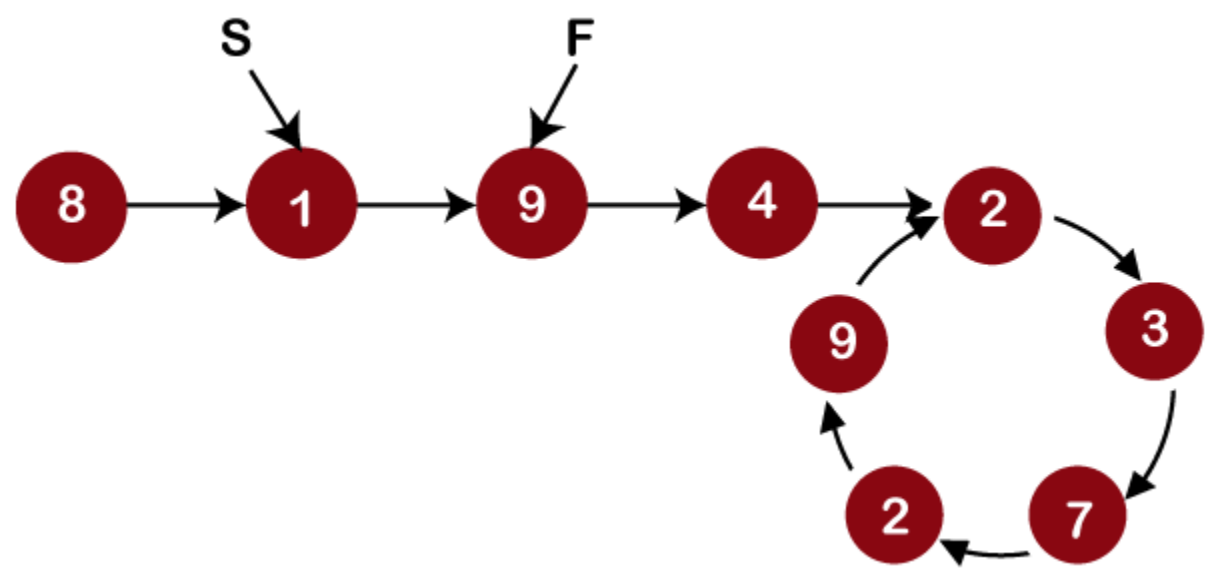
Step 3: If at some point that both the pointers, i.e., 'S' and 'F', point to the same node, then there is a loop in the linked list; otherwise, no loop exists.

Let's visualize the above algorithm for more clarity.

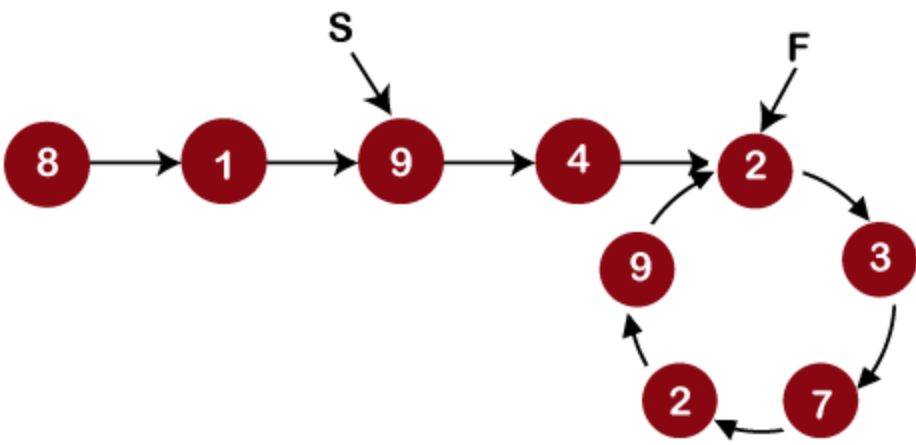


As we can observe in the above figure that both the pointers, i.e., S and F point to the first node. Now, we will move the 'S' pointer by one and the 'F' pointer by two until they meet. If the 'F' pointer reaches the end node means that there is no loop in the linked list.

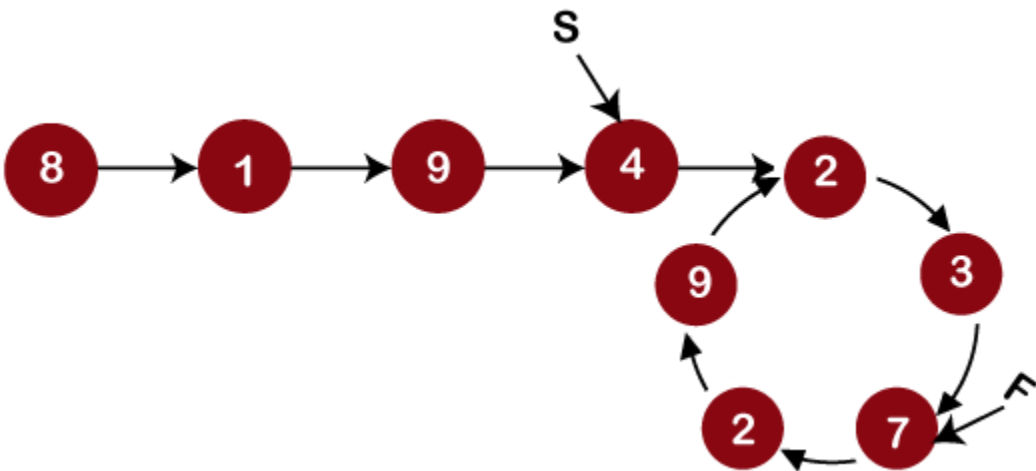
The 'S' pointer moves by one whereas the 'F' pointer moves by two, so 'S' pointer points to node 1, and the 'F' pointer points to node 9 shown as below:



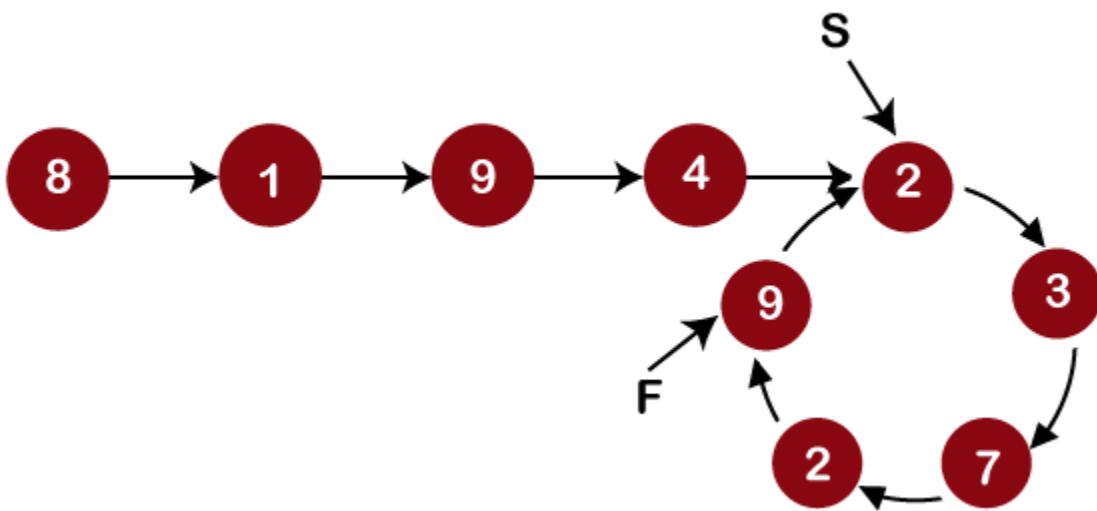
Since both the pointers do not point to the same node and 'F' pointer does not reach the end node so we will again move both the pointers. Now, pointer 'S' will move to the node 9, and pointer 'F' will move to node 2, shown as below:



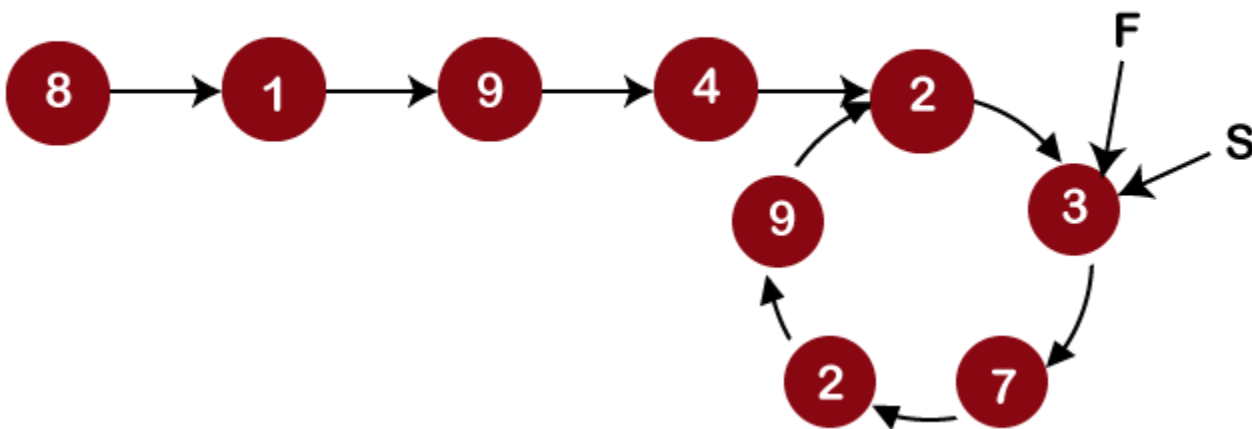
Since both the pointers do not point to the same node, so again, we will increment the pointers. Now, 'S' will point to node 4, and 'F' will point to the node 7 shown as below:



Since both the pointers do not point to the same node, so again, we will increment the pointers. Now, 'S' will point to the node 2, and 'F' will point to node 9 shown as below:



Since both the pointers do not point to the same node, so again, we will increment the pointers. Now, 'S' will point to node 3, and 'F' will also point to node 3 shown as below:



As we can observe in the above figure, both pointers point to the same node, i.e., 3; therefore, the loop exists in the linked list.

Detecting start of the loop

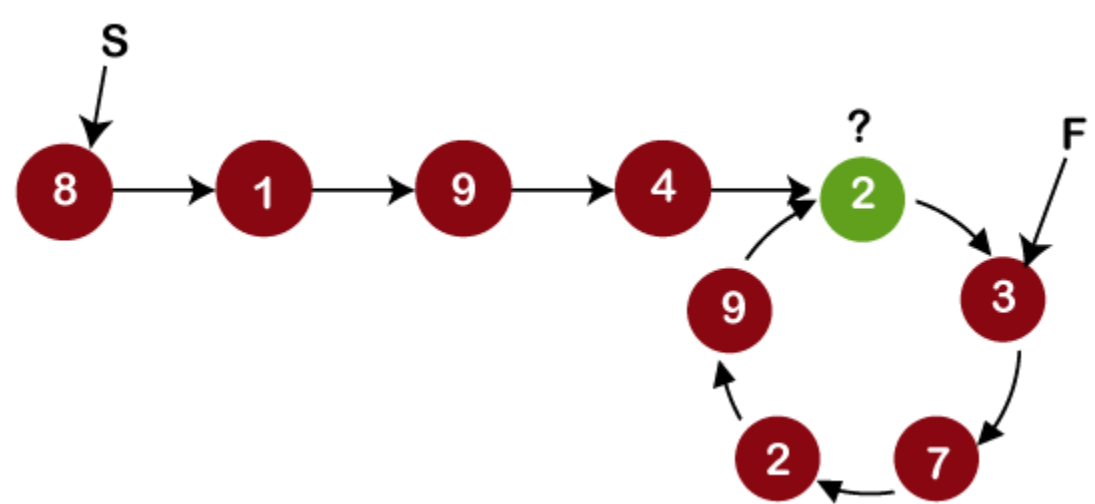
Here, we need to detect the origination of the loop. We will consider the same example which we discussed in detecting the loop. To detect the start of the loop, consider the below algorithm.

Step 1: Move 'S' to the start of the list, but 'F' would remain point to node 3.

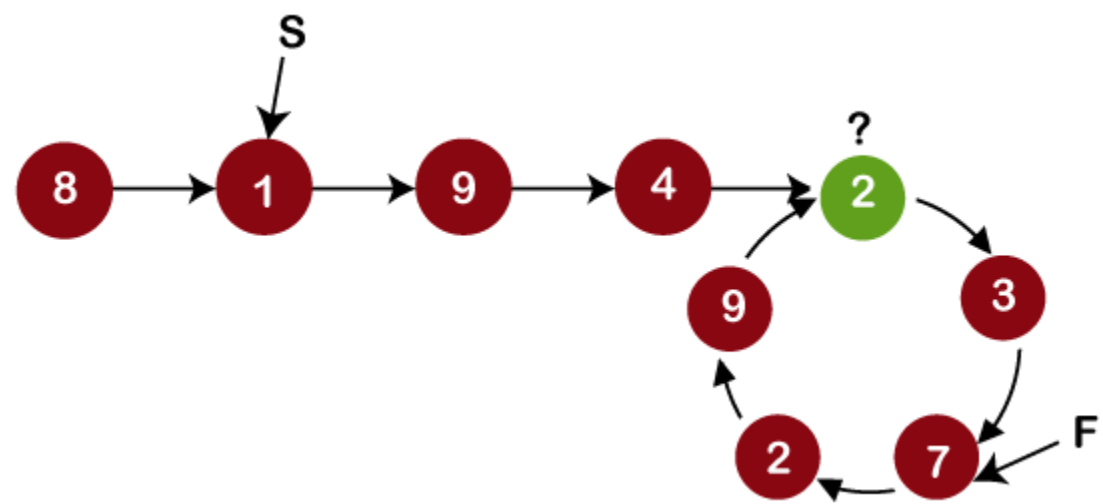
Step 2: Move 'S' and 'F' forward one node at a time until they meet.

Step 3: The node where they meet is the start of the loop.

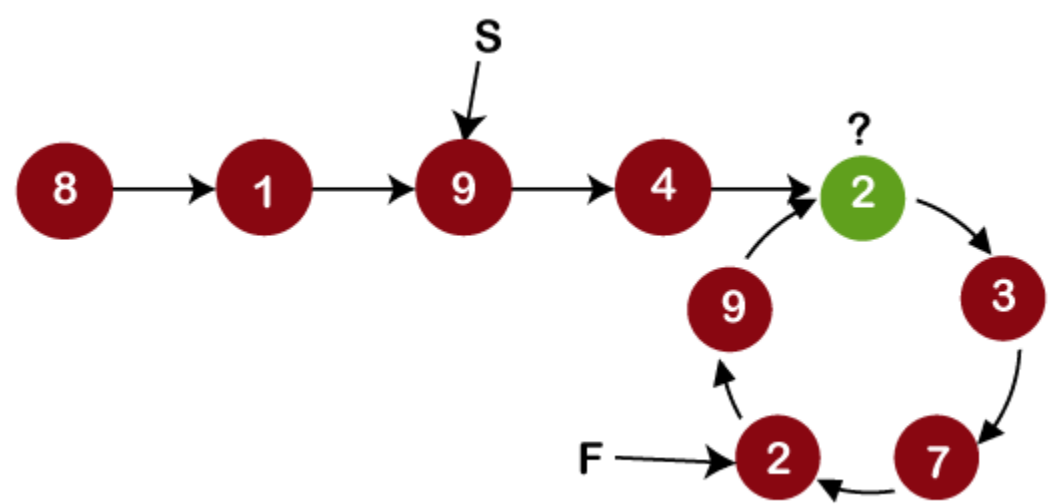
Let's visualize the above algorithm for more clarity.



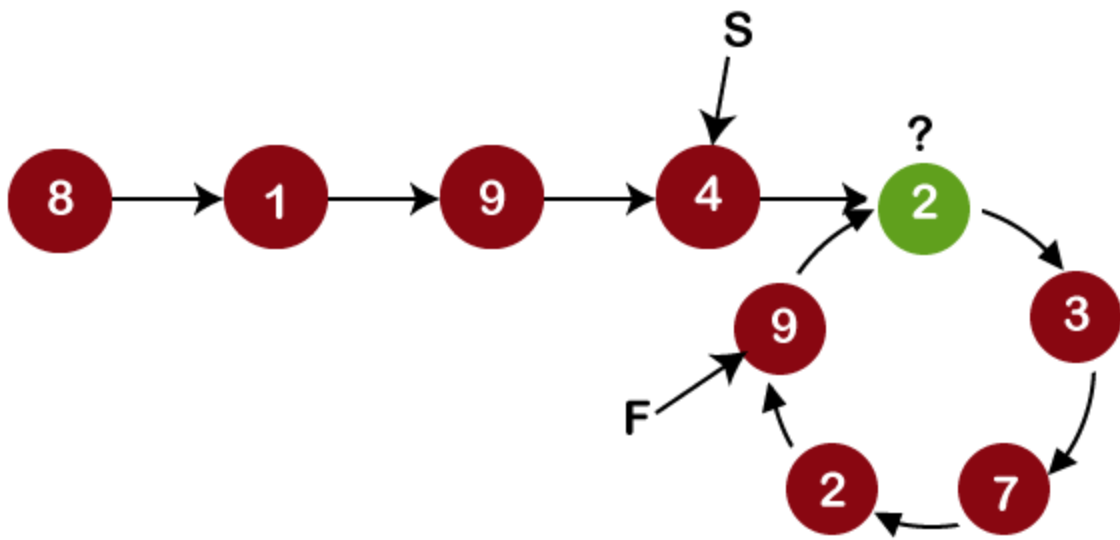
First, we increment the pointer 'S' and 'F' by one; 'S' and 'F' would point to node 1 and node 7, respectively, shown as below:



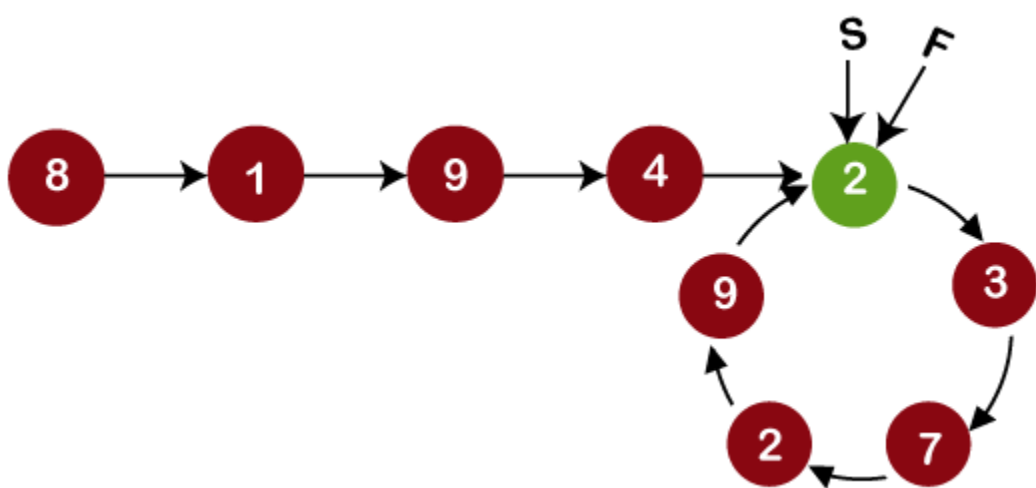
Since both the node are not met, so we again increment the pointers by one node shown as below:



As we can observe in the above figure that the 'S' pointer points to node 9 and 'F' pointer points to the node 2. So again, we increment both the pointers by one node. Now, 'S' would point to the node 4, and 'F' would point to the node 9, shown as below:



As we can observe in the above figure that both the pointers do not point to the same node, so again, we will increment both the pointers by one node. Now, pointer 'S' and 'F' point to node 2 shown as below:



Since both the pointers point to the same node, i.e., 2; therefore, we conclude that the starting node of the loop is node 2.

Why this algorithm works?

Consider 'l' that denotes the length of the loop, which will measure the number in the links.

L: length of the loop

As we can observe in the above figure that there are five links.

Consider 'm' as the distance of the start of the loop from the beginning of the list. In other words, 'm' can be defined as the distance from the starting node to the node from where the loop gets started. So, in the above figure, m is 4 because the starting node is 8 and the node from the loop gets started is 2.

Consider 'k' is the distance of the meeting point of 'S' and 'F' from the start of the loop when they meet for the first time while detecting the loop. In the above linked list, the value of 'k' would be 1 because the node where both fast and slow pointers meet the first time is 3, and the node from where the loop has been started is 2.

When 'S' and 'F' meet for the first time then,

Let's assume that the total distance covered by the slow pointer is **distance_S**, and the total distance covered by the fast pointer is **distance_F**.

$$\text{distance_S} = m + p \cdot l + k$$

where the total distance covered by S is the sum of the distance from the beginning of the list to the start of the loop, the distance covered by the slow pointer in the loop, and the distance from the start of the loop to the node where both the pointers meet.

$$\text{distance_F} = m + q \cdot l + k \quad (q > p \text{ since the speed of 'S' is greater than the speed of 'F'})$$

As we know that when 'S' and 'F' met the first time, then 'F' traverses twice as faster as the 'S' pointer; therefore, the distance covered by 'F' would be two times the distance covered by 'S'. Mathematically, it can be represented as:

$$\text{distance_F} = 2\text{distance_S}$$

The above equation can be written as:

$$m + q \cdot l + k = 2(m + p \cdot l + k)$$

Solving the above equation, we would get:

$m+k = (q-2p) \cdot l$, which implies that $m+k$ is an integer multiple of l (length of the loop).

Implementation of detecting loop in C.

```

1. int detectloop(struct node *list)
2. {
3.     struct node *S = list, *F=list;
4.     while(S && F && F->next)
5.     {
6.         S=S->next;
7.         F=F->next->next;
8.         if(F==S)
9.         {
10.            printf("loop exists");
11.            return 1;
12.        }
13.    }
14.    return 0;
15.}
```

In the above code, detectloop() is the name of the function which will detect the loop in the [linked list](#). We have passed the list pointer of type struct node pointing to the head node in the linked list. Inside the detectloop() function, we have declared two pointers, i.e., 'S' and 'F' of type struct node and assign the reference of the head node to these pointers. We define the while loop in which we are checking whether the 'S', 'F' and F->next are NULL or not. If they are not Null, then the control will go inside the while loop. Within the while loop, 'S' pointer is incremented by one node and 'F' pointer is incremented by two. If both 'F' and 'S' gets equal; means that the loop exists in the linked list.

[Next →](#) [← Prev](#)

Inorder Traversal

Here, we will see how to traverse the node in an inorder fashion in a Binary search tree. In a [binary search tree](#), the left subtree consists of nodes having a smaller value than the root node, while the right subtree consists of nodes having a greater value than the root node.

If we want to traverse the nodes in ascending order, then we use the inorder traversal. The following are the steps required for the inorder traversal:

- Visit all the nodes in the left subtree
- Visit the root node
- Visit all the nodes in the right subtree

There are two approaches used for the inorder traversal:

- Inorder traversal using Recursion
- Inorder traversal using an Iterative method

Inorder Traversal using Recursion

First, we look at the inorder traversal using recursion. Recursion means the function calls itself. In order to access each node, we first need to define the structure of the node.

```

1. struct BinaryTreeNode
2. {
3.     int info;
```

```
4.  struct BinaryTreeNode *left;
5.  struct BinaryTreeNode *right;
6. }
```

The above code defines the structure of the node. The name of the structure is **BinaryTreeNode** that contains three fields, i.e., info, left pointer, and right pointer. The info field contains the value of the node, the left pointer contains the address of the left node, and the right pointer contains the address of the right node.

Recursive approach of Inorder traversal

```
1. void inorder_traversal(struct BinaryTreeNode *root)
2. {
3.     if(root)
4.     {
5.         inorder_traversal(root->left);
6.         printf("%d", root->info);
7.         inorder_traversal(root->right);
8.     }
9. }
```

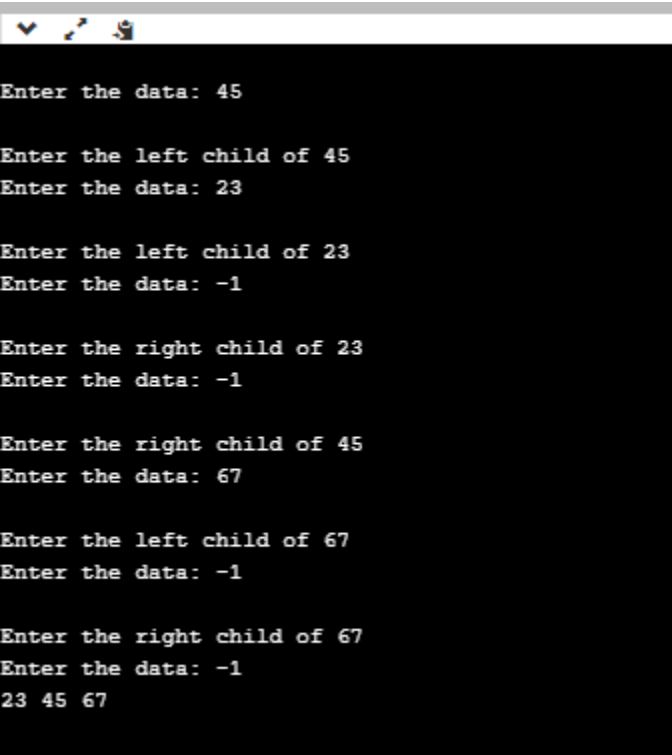
The above approach is the recursive approach. Inside the parenthesis of inorder_traversal function, we have passed the root pointer of type **struct BinaryTreeNode**, which is pointing to the root node of the tree. First, we will check whether the root is NULL or not. If the root is null means that the tree is empty. If the root is not empty, then the inorder_traversal(root_left) function will be called recursively to print all the left subtree nodes and then the root node. Once the left subtree is traversed, and the root node, the inorder_traversal(root_right) will be called recursively to print all the right subtree nodes.

Implementation of Inorder traversal in C.

```
1. #include<stdio.h>
2. // Creating a node structure
3. struct BinaryTreeNode
4. {
5.     int info;
6.     struct BinaryTreeNode *left;
7.     struct BinaryTreeNode *right;
8. };
9.
10. // Creating a tree..
11. struct BinaryTreeNode* create()
12. {
13.     struct BinaryTreeNode *temp;
14.     int data;
15.     temp = (struct BinaryTreeNode *)malloc(sizeof(struct BinaryTreeNode)); // creating a new node
16.     printf("\nEnter the data: ");
17.     scanf("%d", &data);
18.
19.     if(data == -1)
20.     {
21.         return NULL;
22.     }
23.     temp->info = data;
24.     printf("\nEnter the left child of %d", data);
25.     temp->left = create(); // creating a left child
26.     printf("\nEnter the right child of %d", data);
27.     temp->right = create(); // creating a right child
28.     return temp;
29. }
30.
31. //inorder traversal
32. void inorder(struct BinaryTreeNode *root)
```

```
33. {
34.   if(root==NULL)
35.     return;
36.   else
37.
38.     inorder(root->left);
39.     printf("%d ", root->info);
40.     inorder(root->right);
41. }
42. void main()
43. {
44.   struct BinaryTreeNode *root;
45.   root = create(); // calling create function.
46.   inorder(root); // calling inorder function
47. }
```

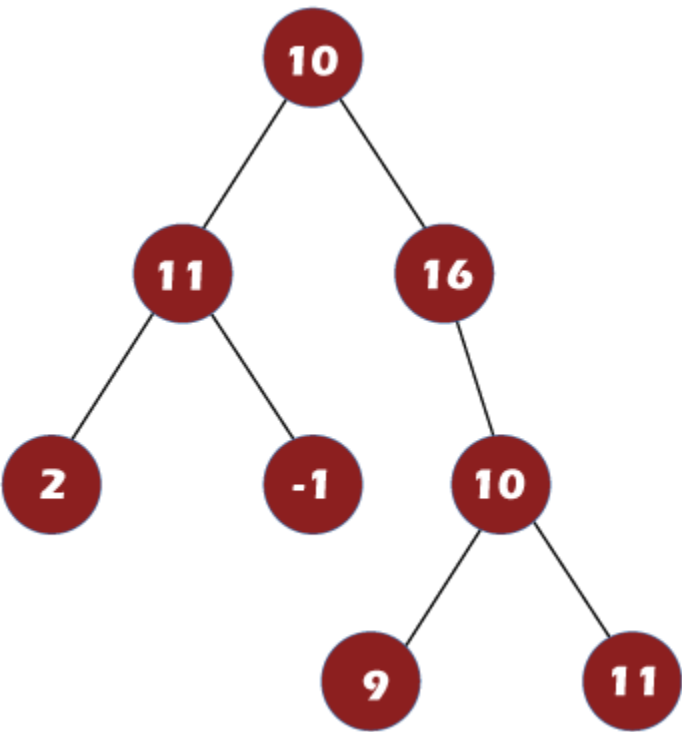
Output



Iterative Inorder Traversal

The second approach is the iterative approach for the inorder traversal. When we were using the recursive approach, we used the system stack for the inorder traversal. For the iterative approach, we will use the external stack. Let's understand through an example.

Consider the below tree:



Now we have to perform the inorder traversal using an external stack. The following are the steps required to perform the inorder traversal:

Step 1: First we push the root node, i.e., 10, into the stack:

Stack: 10

Step 2: Node 10 has a left child, i.e., 11, so we will push node 11 into the stack.

Stack: 10, 11

Step 3: Node 11 has a left child, i.e., 2, so we will push element 2 into the stack.

Stack: 10, 11, 2

Step 4: Since node 2 does not have a left child, we will pop node 2 from the stack and get printed in the output. Set the root node as NULL.

Stack: 10, 11

Output: 2

Step 5: Since the root node is NULL, so pop node 11 from the stack.

Stack: 10

Output: 2, 11

Step 6: Node 11 has a right child, i.e., -1, so set the root as -1. Push -1 into the stack.

Stack: 10, -1

Step 7: Since node -1 does not have a left or a right child, so set the root as NULL. Pop -1 from the stack.

Stack: 10

Output: 2, 11, -1

Step 8: Since the root is NULL, pop 10 from the stack. Node 10 has a right child, i.e., 16, so set root to 16.

Stack: empty

Output: 2, 11, -1, 10

Step 9: The root is 16, so push 16 into the stack.

Stack: 16

Step 10: Node 16 has also left child, i.e., 10, so push 10 into the stack. Set root to 10.

Stack: 16, 10.

Step 11: The node 10 has a left child, i.e., 9, so push 9 into the stack. Set root to 9.

Stack: 16, 10, 9

Step 12: Since the node 9 has no left and right child so set root as **NULL**. Pop element 9 from the stack.

Stack: 16, 10

Output: 2, 11, -1, 10, 9

Step 13: The root is set to node 9 and the topmost element is 10, so pop element 10 from the stack. The node 10 has a right child, i.e., node 11. Set root to node 11.

Stack: 16

Output: 2, 11, -1, 10, 9, 10

Step 14: The root is set to 11 so push element 11 into the stack.

Stack: 16, 11

Step 14: Since the root is set to 11 and it does not have any left and right child, pop element 11 from the stack. Set root to NULL.

Stack: 16

Output: 2, 11, -1, 10, 9, 10, 11

Step 15: The root is set to NULL, so pop element 16 from the stack. Since the node 16 does not have a right child, set root node to NULL.

Stack: empty

Output: 2, 11, -1, 10, 9, 10, 11, 16

Algorithm of iterative inorder traversal

1. Step 1: Create an empty stack named as 'S'.
2. Step 2: Initialize current node to root.
3. Step 3: Push current node to the stack 'S' and then set current = current->left until the current node is NULL.
4. Step 4: If current node is NULL and stack 'S' is not empty:
 5. i) Pop topmost item of stack and print it.
 6. ii) Set current= popped item->right
 7. iii) Repeat step 3
8. Step 5: If both current node and stack are empty means that the inorder traversal is completed.

Convert Infix to Postfix notation

Before understanding the conversion from infix to postfix notation, we should know about the infix and postfix notations separately.

An infix and postfix are the expressions. An expression consists of constants, variables, and symbols. Symbols can be operators or parenthesis. All these components must be arranged according to a set of rules so that all these expressions can be evaluated using the set of rules.

Examples of expressions are:

5 + 6

A - B

(P * 5)

All the above expressions have a common structure, i.e., we have an operator between the two operands. An Operand is an object or a value on which the operation is to be performed. In the above expressions, 5, 6 are the operands while '+', '-', and '*' are the operators.

What is infix notation?

When the operator is written in between the operands, then it is known as **infix notation**. Operand does not have to be always a constant or a variable; it can also be an expression itself.

For example,

$(p + q) * (r + s)$

In the above expression, both the expressions of the multiplication operator are the operands, i.e., **(p + q)**, and **(r + s)** are the operands.

In the above expression, there are three operators. The operands for the first plus operator are p and q, the operands for the second plus operator are r and s. While performing the **operations on the expression, we need to follow some set of rules to evaluate the result. In the** above expression, addition operation would be performed on the two expressions, i.e., p+q and r+s, and then the multiplication operation would be performed.

Syntax of infix notation is given below:

<operand> <operator> <operand>

If there is only one operator in the expression, we do not require applying any rule. For example, 5 + 2; in this expression, addition operation can be performed between the two operands (5 and 2), and the result of the operation would be 7.

If there are multiple operators in the expression, then some rule needs to be followed to evaluate the expression.

If the expression is:

4 + 6 * 2

If the plus operator is evaluated first, then the expression would look like:

10 * 2 = 20

If the multiplication operator is evaluated first, then the expression would look like:

4 + 12 = 16

The above problem can be resolved by following the operator precedence rules. In the algebraic expression, the order of the operator precedence is given in the below table:

Operators	Symbols
Parenthesis	(), { }, []
Exponents	^
Multiplication and Division	*, /
Addition and Subtraction	+, -

The first preference is given to the parenthesis; then next preference is given to the exponents. In the case of multiple exponent operators, then the operation will be applied from right to left.

For example:

2^2^3 = 2 ^ 8

= 256

After exponent, multiplication, and division operators are evaluated. If both the operators are present in the expression, then the operation will be applied from left to right.

The next preference is given to addition and subtraction. If both the operators are available in the expression, then we go from left to right.

The operators that have the same precedence termed as **operator associativity**. If we go from left to right, then it is known as left-associative. If we go from right to left, then it is known as right-associative.

Problem with infix notation

To evaluate the infix expression, we should know about the **operator precedence** rules, and if the operators have the same precedence, then we should follow the **associativity** rules. The use of parenthesis is very important in infix notation to control the order in which the operation to be performed. Parenthesis improves the readability of the expression. An infix expression is the most common way of writing expression, but it is not easy to parse and evaluate the infix expression without ambiguity. So, mathematicians and logicians studied this problem and discovered two other ways of writing expressions which are prefix and postfix. Both expressions do not require any parenthesis and can be parsed without ambiguity. It does not require operator precedence and associativity rules.

Postfix Expression

The postfix expression is an expression in which the operator is written after the operands. For example, the postfix expression of infix notation (2+3) can be written as 23+.

Some key points regarding the postfix expression are:

- In postfix expression, operations are performed in the order in which they have written from left to right.
- It does not any require any parenthesis.

- We do not need to apply operator precedence rules and associativity rules.

Algorithm to evaluate the postfix expression

- Scan the expression from left to right until we encounter any operator.
- Perform the operation
- Replace the expression with its computed value.
- Repeat the steps from 1 to 3 until no more operators exist.

Let's understand the above algorithm through an example.

Infix expression: $2 + 3 * 4$

We will start scanning from the left most of the expression. The multiplication operator is an operator that appears first while scanning from left to right. Now, the expression would be:

Expression = $2 + 34^*$

= $2 + 12$

Again, we will scan from left to right, and the expression would be:

Expression = $2\ 12\ +$

= 14

Evaluation of postfix expression using stack.

- Scan the expression from left to right.
- If we encounter any operand in the expression, then we push the operand in the stack.
- When we encounter any operator in the expression, then we pop the corresponding operands from the stack.
- When we finish with the scanning of the expression, the final value remains in the stack.

Let's understand the evaluation of postfix expression using stack.

Example 1: Postfix expression: $2\ 3\ 4\ *\ +$

Input	Stack	
2 3 4 * +	empty	Push 2
3 4 * +	2	Push 3
4 * +	3 2	Push 4
* +	4 3 2	Pop 4 and 3, and perform $4*3 = 12$. Push 12 into the stack.
+	12 2	Pop 12 and 2 from the stack, and perform $12+2 = 14$. Push 14 into the stack.

The result of the above expression is 14.

Example 2: Postfix expression: $3\ 4\ *\ 2\ 5\ *\ +$

Input	Stack	
3 4 * 2 5 * +	empty	Push 3
4 * 2 5 * +	3	Push 4
*2 5 * +	4 3	Pop 3 and 4 from the stack and perform $3*4 = 12$. Push 12 into the stack.
2 5 * +	12	Push 2

5 * +	2 12	Push 5
*+	5 2 12	Pop 5 and 2 from the stack and perform $5*2 = 10$. Push 10 into the stack.
+	10 12	Pop 10 and 12 from the stack and perform $10+12 = 22$. Push 22 into the stack.

The result of the above expression is 22.

Algorithm to evaluate postfix expression

1. Read a character
2. If the character is a digit, convert the character into int and push the integer into the stack.
3. If the character is an operator,
 - o Pop the elements from the stack twice obtaining two operands.
 - o Perform the operation
 - o Push the result into the stack.

Conversion of infix to postfix

Here, we will use the stack data structure for the conversion of infix expression to prefix expression. Whenever an operator will encounter, we push operator into the stack. If we encounter an operand, then we append the operand to the expression.

Rules for the conversion from infix to postfix expression

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8. At the end of the expression, pop and print all the operators of the stack.

Let’s understand through an example.

Infix expression: $K + L - M*N + (O^P) * W/U/V * T + Q$

Input Expression	Stack	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L+
M	-	K L+ M
*	- *	K L+ M
N	- *	K L + M N
+	+	K L + M N* K L + M N* -
(+ (K L + M N *-

O	+ (K L + M N * - O
^	+ (^	K L + M N* - O
P	+ (^	K L + M N* - O P
)	+	K L + M N* - O P ^
*	+ *	K L + M N* - O P ^
W	+ *	K L + M N* - O P ^ W
/	+ /	K L + M N* - O P ^ W *
U	+ /	K L + M N* - O P ^W*U
/	+ /	K L + M N* - O P ^W*U/
V	+ /	KL + MN*-OP^W*U/V
*	+ *	KL+MN*-OP^W*U/V/
T	+ *	KL+MN*-OP^W*U/V/T
+	+	KL+MN*-OP^W*U/V/T* KL+MN*-OP^W*U/V/T*+
Q	+	KL+MN*-OP^W*U/V/T*Q
		KL+MN*-OP^W*U/V/T*+Q+

The final postfix expression of infix expression(K + L - M*N + (O^P) * W/U/V * T + Q) is KL+MN*-OP^W*U/V/T*+Q+.

Convert infix to prefix notation

What is infix notation?

An infix notation is a notation in which an expression is written in a usual or normal format. It is a notation in which the operators lie between the operands. The examples of infix notation are A+B, A*B, A/B, etc.

As we can see in the above examples, all the operators exist between the operands, so they are infix notations. Therefore, the syntax of infix notation can be written as:

<operand> <operator> <operand>

Parsing Infix expressions

In order to parse any expression, we need to take care of two things, i.e., **Operator precedence** and **Associativity**. Operator precedence means the precedence of any operator over another operator. For example:

$$A + B * C \rightarrow A + (B * C)$$

As the multiplication operator has a higher precedence over the addition operator so B * C expression will be evaluated first. The result of the multiplication of B * C is added to the A.

Precedence order

Operators	Symbols
Parenthesis	{ }, (), []
Exponential notation	^
Multiplication and Division	*, /
Addition and Subtraction	+, -

Associativity means when the operators with the same precedence exist in the expression. For example, in the expression, i.e., $A + B - C$, '+' and '-' operators are having the same precedence, so they are evaluated with the help of associativity. Since both '+' and '-' are left-associative, they would be evaluated as $(A + B) - C$.

Associativity order

Operators	Associativity
^	Right to Left
*, /	Left to Right
+, -	Left to Right

Let's understand the associativity through an example.

$1 + 2 * 3 + 30 / 5$

Since in the above expression, * and / have the same precedence, so we will apply the associativity rule. As we can observe in the above table that * and / operators have the left to right associativity, so we will scan from the leftmost operator. The operator that comes first will be evaluated first. The operator * appears before the / operator, and multiplication would be done first.

$1 + (2 * 3) + (30 / 5)$

$1 + 6 + 6 = 13$

What is Prefix notation?

A prefix notation is another form of expression but it does not require other information such as precedence and associativity, whereas an infix notation requires information of precedence and associativity. It is also known as **polish notation**. In prefix notation, an operator comes before the operands. The syntax of prefix notation is given below:

<operator> <operand> <operand>

For example, if the infix expression is $5 + 1$, then the prefix expression corresponding to this infix expression is $+51$.

If the infix expression is:

$a * b + c$

↓

$*ab + c$

↓

$+*abc$ (Prefix expression)

Consider another example:

$A + B * C$

First scan: In the above expression, multiplication operator has a higher precedence than the addition operator; the prefix notation of $B * C$ would be $(*BC)$.

$A + *BC$

Second scan: In the second scan, the prefix would be:

$+A *BC$

In the above expression, we use two scans to convert infix to prefix expression. If the expression is complex, then we require a greater number of scans. We need to use that method that requires only one scan, and provides the desired result. If we achieve the desired output through one scan, then the algorithm would be efficient. This is possible only by using a stack.

Conversion of Infix to Prefix using Stack

$K + L - M * N + (O^P) * W/U/V * T + Q$

If we are converting the expression from infix to prefix, we need first to reverse the expression.

The Reverse expression would be:

$Q + T * V/U/W *) P^O(+ N*M - L + K$

To obtain the prefix expression, we have created a table that consists of three columns, i.e., input expression, stack, and prefix expression. When we encounter any symbol, we simply add it into the prefix expression. If we encounter the operator, we will push it into the stack.

Input expression	Stack	Prefix expression
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW
P	+*//*)	QTVUWP
^	+*//*)^	QTVUWP
O	+*//*)^	QTVUWPO
(+*//*	QTVUWPO^
+	++	QTVUWPO^*//*
N	++	QTVUWPO^*//*N
*	++*	QTVUWPO^*//*N
M	++*	QTVUWPO^*//*NM
-	++-	QTVUWPO^*//*NM*
L	++-	QTVUWPO^*//*NM*L
+	++-+	QTVUWPO^*//*NM*L
K	++-+	QTVUWPO^*//*NM*LK
		QTVUWPO^*//*NM*LK+-++

The above expression, i.e., QTVUWPO^*//*NM*LK+-++, is not a final expression. We need to reverse this expression to obtain the prefix expression.

Rules for the conversion of infix to prefix expression:

- First, reverse the infix expression given in the problem.
- Scan the expression from left to right.
- Whenever the operands arrive, print them.

- If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
- If the incoming operator has higher precedence than the TOP of the stack, push the incoming operator into the stack.
- If the incoming operator has the same precedence with a TOP of the stack, push the incoming operator into the stack.
- If the incoming operator has lower precedence than the TOP of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the stack again and pop the operator from the stack till it finds the operator of a lower precedence or same precedence.
- If the incoming operator has the same precedence with the top of the stack and the incoming operator is ^, then pop the top of the stack till the condition is true. If the condition is not true, push the ^ operator.
- When we reach the end of the expression, pop, and print all the operators from the top of the stack.
- If the operator is ')', then push it into the stack.
- If the operator is '(', then pop all the operators from the stack till it finds) opening bracket in the stack.
- If the top of the stack is ')', push the operator on the stack.
- At the end, reverse the output.

Pseudocode of infix to prefix conversion

1. Function InfixtoPrefix(stack, infix)
2. infix = reverse(infix)
3. loop i = 0 to infix.length
4. **if** infix[i] is operand → prefix+= infix[i]
5. **else if** infix[i] is '(' → stack.push(infix[i])
6. **else if** infix[i] is ')' → pop and print the values of stack till the symbol ')' is not found
7. **else if** infix[i] is an operator(+, -, *, /, ^) →
- 8.
9. **if** the stack is empty then push infix[i] on the top of the stack.
10. Else →
11. If precedence(infix[i] > precedence(stack.top))
12. → Push infix[i] on the top of the stack
13. **else if**(infix[i] == precedence(stack.top) && infix[i] == '^')
14. → Pop and print the top values of the stack till the condition is **true**
15. → Push infix[i] into the stack
16. **else if**(infix[i] == precedence(stack.top))
17. → Push infix[i] on to the stack
18. Else **if**(infix[i] < precedence(stack.top))
19. → Pop the stack values and print them till the stack is not empty and infix[i] < precedence(stack.top)
20. → Push infix[i] on to the stack
21. End loop
22. Pop and print the remaining elements of the stack
23. Prefix = reverse(prefix)
24. **return**

Conversion of Prefix to Postfix expression

Before understanding the conversion of prefix to postfix conversion, we should know about the prefix and postfix expressions separately.

What is Prefix conversion?

An infix expression is an expression in which the operators are written between the two operands. If we move the operator before the operands then it is known as a prefix expression. In other words, prefix expression can be defined as an expression in which all the operators precede the two operands.

For example:

If the infix expression is given as: A + B * C

As we know that the multiplication operator * has a higher precedence than the addition operator. First, multiplication operator will move before operand B shown as below:

A + * B C

Once the multiplication operator is moved before 'B' operand, addition operator will move before the operand 'A' shown as below:

+ A * B C

Evaluation of Prefix Expression using Stack

- Step 1:** Initialize a pointer 'S' pointing to the end of the expression.
- Step 2:** If the symbol pointed by 'S' is an operand then push it into the stack.
- Step 3:** If the symbol pointed by 'S' is an operator then pop two operands from the stack. Perform the operation on these two operands and stores the result into the stack.
- Step 4:** Decrement the pointer 'S' by 1 and move to step 2 as long as the symbols left in the expression.
- Step 5:** The final result is stored at the top of the stack and return it.
- Step 6:** End

Let's understand the evaluation of prefix expression through an example.

Expression: +, -, *, 2, 2, /, 16, 8, 5

First, we will reverse the expression given above.

Expression: 5, 8, 16, /, 2, 2, *, -, +

We will use the stack data structure to evaluate the prefix expression.

Symbol Scanned	Stack
5	5
8	5, 8
16	5, 8, 16
/	5, 2
2	5, 2, 2
2	5, 2, 2, 2
*	5, 2, 4
-	5, 2
+	7

The final result of the above expression is 7.

What is Postfix expression?

If we move the operators after the operands then it is known as a postfix expression. In other words, postfix expression can be defined as an expression in which all the operators are present after the operands.

For example:

If the infix expression is A + B * C

As we know that the multiplication operator has a higher precedence than the addition operator, so multiplication operator will move after the operands B and C shown as below:

A + B C *

Once the multiplication operator is moved after the operand C, then the addition operator will come after the multiplication operator shown as below:

A B C * +

Evaluation of Postfix expression using Stack

Algorithm for the evaluation of postfix expression using stack:

Step 1: Create an empty stack used for storing the operands.

Step 2: Scan each element of an expression one by one and do the following:

- If the element is an operand then push it into the stack.
- If the element is an operator then pop two operands from the stack. Perform operation on these operands. Push the final result into the stack.

Step 3: When the expression is scanned completely, the value available in the stack would be the final output of the given expression.

Let's understand the evaluation of postfix expression using stack through an example.

If the expression is: 5, 6, 2, +, *, 12, 4, /, -

Symbol Scanned	Stack
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
12	40, 12
4	40, 12, 4
/	40, 3
-	37

The result of the above expression is 37.

Conversion of Prefix to Postfix Expression

Here, we will see the conversion of prefix to postfix expression using a stack data structure.

Rules for prefix to postfix expression using stack data structure:

- Scan the prefix expression from right to left, i.e., reverse.
- If the incoming symbol is an operand then push it into the stack.
- If the incoming symbol is an operator then pop two operands from the stack. Once the operands are popped out from the stack, we add the incoming symbol after the operands. When the operator is added after the operands, then the expression is pushed back into the stack.
- Once the whole expression is scanned, pop and print the postfix expression from the stack.

Pseudocode for prefix to postfix conversion

1. Function PrefixToPostfix(string prefix)
2. 1. Stack s
3. 2. Loop: i = prefix.length-1 to 0
4. • if prefix[i] is operand ->

5. s.push(prefix[i])
6. • **else if** prefix[i] is operator->
7. op1 = s.top()
8. s.pop()
9. op2 = s.top()
- 10.s.pop()
- 11.exp = op1 + op2 + prefix[i]
- 12.s.push(exp)
- 13.End Loop
- 14.3. Return s.top

Let's understand the conversion of Prefix to Postfix expression using Stack through an example.

If the expression is: * - A / B C - / A K L

Symbols to be scanned	Action	Stack	Description
L	Push L into the stack	L	
K	Push K into the stack	L, K	
A	Push A into the stack	L, K, A	
/	Pop A from the stack Pop K from the stack Push A K / into the stack	L, A K /	Pop two operands from the stack, i.e., A and K. Add '/' operator after K operand, i.e., AK/. Push AK/ into the stack.
-	Pop A K / and L from the stack. Push (A K / L -) into the stack	A K / L -	Pop two operands from the stack, i.e., AK/ and L. Add '-' operator after 'L' operand.
C	Push C into the stack	AK/L-, C	
B	Push B into the stack	AK/L-, C, B	
/	Pop B and C from the stack. Push BC/ into the stack.	AK/L-, BC/	Pop two operands from the stack, i.e., B and C. Add '/' operator after C operator, i.e., BC/. Push BC/ into the stack.
A	Push A into the stack	AK/L-, BC/, A	
-	Pop BC/ and A from the stack. Push ABC/- into the stack.	AK/L-, ABC/-	Pop two operands from the stack, i.e., A and BC/. Add '-' operator after '/'.
*	Pop ABC/- and AK/L- from the stack. Push ABC/AK/L-* into the stack.	ABC/- AK/L-*	Pop two operands from the stack, i.e., ABC/-, and AK/L- . Add '*' operator after L and '-' operator, i.e., ABC/-AK/L-*.

Conversion of Postfix to Prefix expression

What is Postfix expression?

A postfix expression is said to be an expression in which the operator appears after the operands. It can be written as:

(operand) (operand) (operator)

For example:

If the expression is:

$(A+B) * (C+D)$

Firstly, operator precedence rules will be applied to the above expression. Since the parenthesis has higher precedence than the multiplication operator; therefore '+' will be resolved first, and the + operator will come after AB and CD shown as below:

$(AB+) * (CD+)$

Now, the multiplication operator will move after CD+ shown as below:

$AB+ CD+*$

What is Prefix Expression?

A prefix expression is said to be an expression in which the operator appears before the operands.

For example:

If the expression is given as:

$(A+B) * (C+D)$

Firstly, operator precedence rules will be applied to the above expression. Since the parenthesis has higher precedence than the multiplication operator; therefore, the '+' operator will be resolved first, and the '+' operator will move before the operands AB and CD shown as below:

$(+AB) * (+CD)$

Now, the multiplication operator will move before the +AB shown as below:

$*+AB+CD$

Conversion of Postfix to Prefix expression

There are two ways of converting a postfix into a prefix expression:

1. Conversion of Postfix to Prefix expression manually.
2. Conversion of Postfix to Prefix expression using stack.

Conversion of Postfix to Prefix expression manually

The following are the steps required to convert postfix into prefix expression:

- Scan the postfix expression from left to right.
- Select the first two operands from the expression followed by one operator.
- Convert it into the prefix format.
- Substitute the prefix sub expression by one temporary variable
- Repeat this process until the entire postfix expression is converted into prefix expression.

Let's understand through an example.

a b - c +

First, we scan the expression from left to right. We will move '-' operator before the operand ab.

-abc+

The next operator '+' is moved before the operand -abc is shown as below:

+ -abc

Conversion of Postfix to Prefix expression using Stack

The following are the steps used to convert postfix to prefix expression using stack:

- Scan the postfix expression from left to right.

- If the element is an operand, then push it into the stack.
- If the element is an operator, then pop two operands from the stack.

Create an expression by concatenating two operands and adding operator before the operands.

Push the result back to the stack.

- Repeat the above steps until we reach the end of the postfix expression.

Pseudocode for the conversion of Postfix to Prefix

1. Function PostfixToPrefix(string postfix)
2. 1. Stack s
3. 2. Loop: i = 0 to postfix.length
4. 2.1 **if** postfix[i] is operand ->
5. s.push(postfix[i])
- 6.
7. 2.2 **else if** postfix[i] is operator->
8. op1 = s.top()
9. s.pop()
10. op2 = s.top()
11. s.pop()
12. expression = postfix[i] + op2 + op1
13. s.push(expression)
14. end loop
15. **return** s.top

Let's understand the conversion of postfix to prefix expression using stack.

If the Postfix expression is given as:

AB + CD - *

Symbol Scanned	Action	Stack	Description
A	Push A into the stack	A	
B	Push B into the stack	AB	
+	Pop B from the stack Pop A from the stack Push +AB into the stack.	+AB	Pop two operands from the stack, i.e., A and B. Add '+' operator before the operands AB, i.e., +AB.
C	Push C into the stack	+ABC	
D	Push D into the stack	+ABCD	
-	Pop D from the stack. Pop C from the stack. Push -CD into the stack	+AB - CD	Pop two operands from the stack, i.e., D and C. Add '-' operator before the operands CD, i.e., -CD.
*	Pop from the -CD	*+AB - CD	Pop two operands from the stack, i.e., -CD and +AB. Add '*' operator before +AB

	stack. Pop +AB from the stack. Push *+AB - CD into the stack.		then the expression would become *+AB- CD.
--	---	--	---

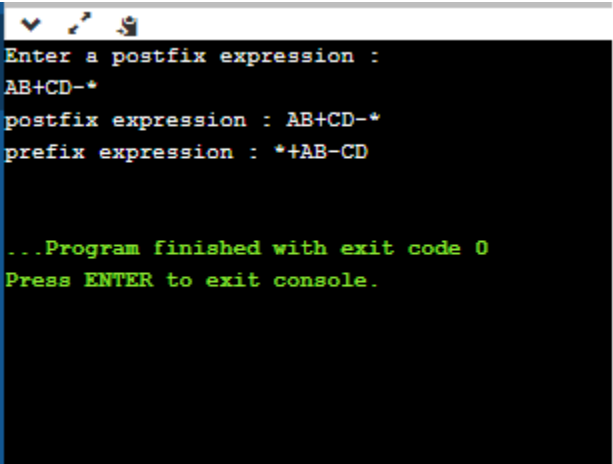
The prefix expression of the above postfix expression is *+AB-CD.

Implementation of Postfix to Prefix conversion in C++

```
1. // C++ program to convert postfix to prefix expression.
2. #include <iostream>
3. #include<stack>
4. using namespace std;
5. // Checking whether the symbol is operand or not..
6. bool isOperand(char c)
7. {
8.     if((c>='a' && c<='z') || (c>='A' && c<='Z'))
9.     {
10.         return true;
11.     }
12.     else
13.     {
14.         return false;
15.     }
16. }
17. // Converting postfix to prefix expression in C++
18. string postfixtoprefix(string postfix)
19. {
20.     stack<string> s; // using predefined stack data structure in stl library
21.     // executing the loop from 0 till the length of the expression..
22.     for(int i=0; i< postfix.length(); i++)
23.     {
24.         if(isOperand(postfix[i])) // calling the isOperand() function
25.         {
26.             string op(1, postfix[i]); // converting the char type variable into string type.
27.             s.push(op); // Pushing the operand into the stack..
28.         }
29.         else
30.         {
31.             string op1 = s.top(); // declaration of op1 variable of type stri
32.             s.pop(); // pop the operand from the stack.
33.             string op2 = s.top(); // declaration of op2 variable of type string
34.             s.pop(); // pop the operand from the stack.
35.             string expression = postfix[i] + op2 + op1; // concatenating the operands and operator
36.             s.push(expression); // push the expression into the stack.
37.
38.         }
39.
40.     }
41.     return s.top(); // returning the top of the stack.
42. }
43. int main()
44. {
45.     string postfix, prefix; // declaration of two variables of type string
46.     std::cout << "Enter a postfix expression : " << std::endl;
```

```
47. std::cin >> postfix;
48. std::cout << "postfix expression : " << postfix<<std::endl;
49. prefix = postfixtoprefix(postfix); // calling the postfixtoprefix() function
50. std::cout << "prefix expression : " << prefix<< std::endl;
51. return 0;
52.}
```

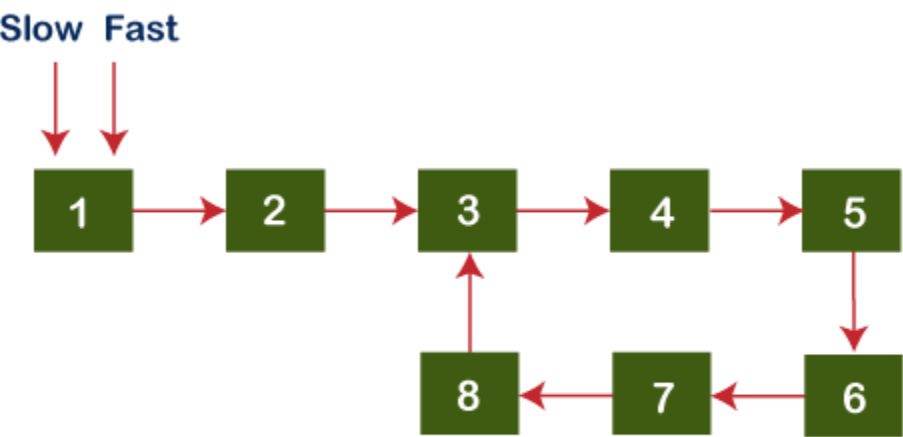
Output



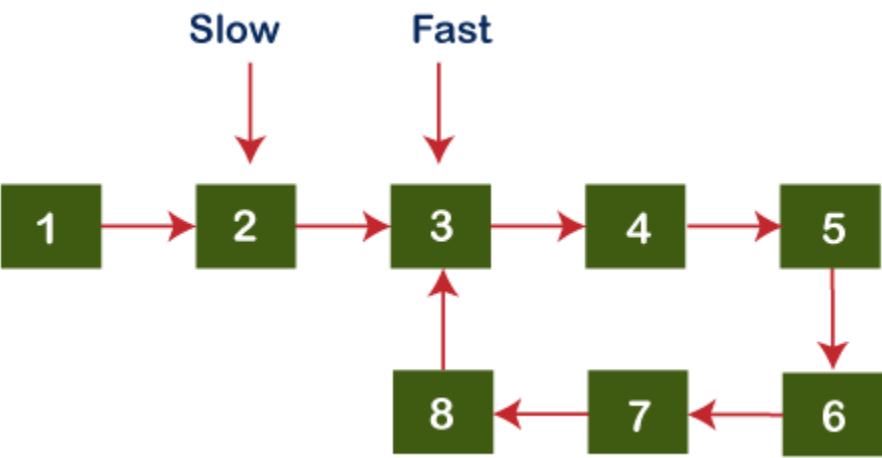
Remove the loop in a Linked List

In this topic, we will learn how to remove the loop from the linked list. Till now, we have learnt how to detect and start of the loop by using **Floyd's** algorithm. The **Floyd's** algorithm will also be used to remove the loop from the linked list.

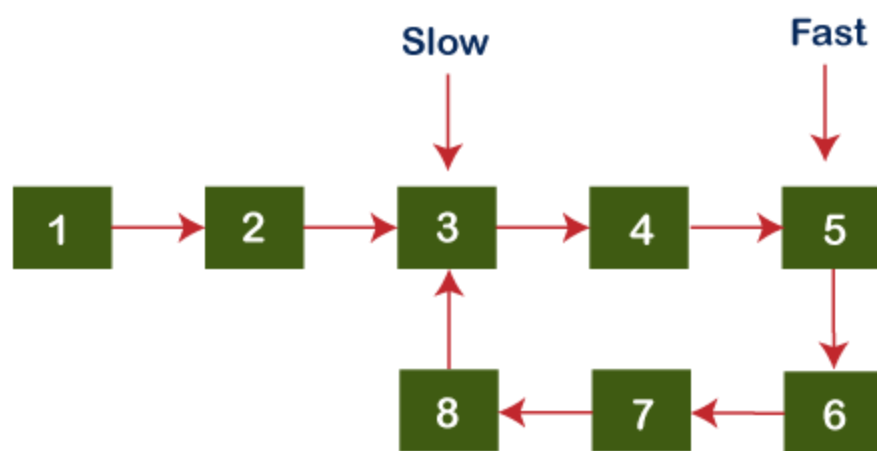
Let's understand through an example.



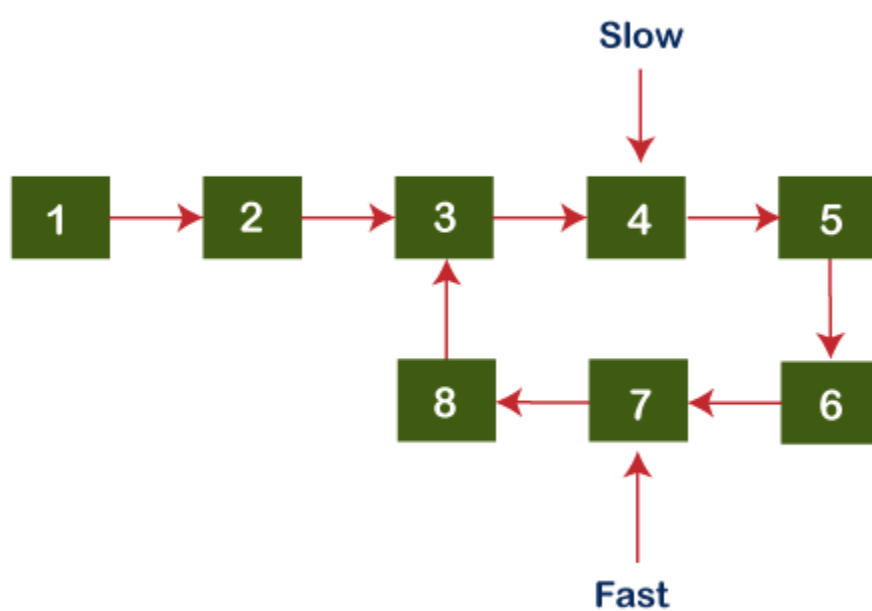
As we know that slow pointer increments by one and fast pointer increments by two. In the above example, initially, both slow and fast pointer point to the first node, i.e., node 1. The slow pointer gets incremented by one, and fast pointer gets incremented by two, and slow and fast pointers point to nodes 2 and 3, respectively, as shown as below:



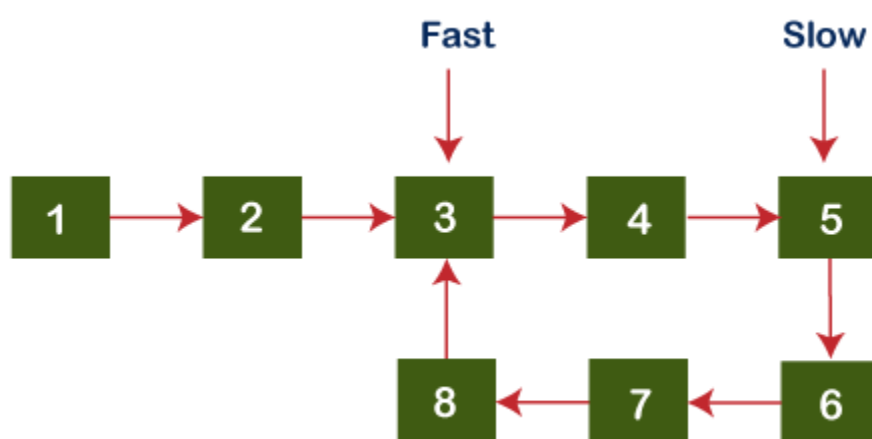
Since both the pointers do not point to the same node, we will increment both fast and slow pointers again. Now, slow pointer points to the node 3 while the fast pointer points to node 5 shown as below:



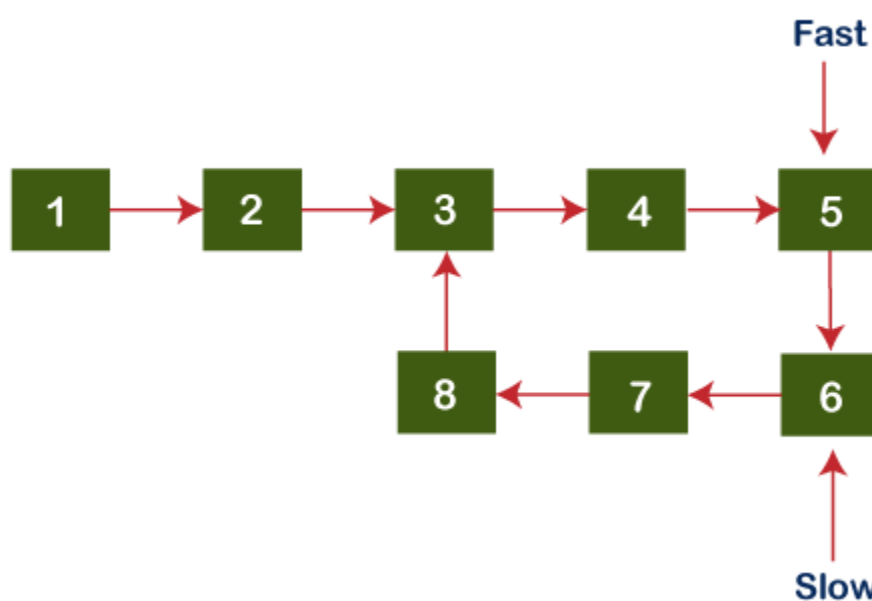
Since both the pointers do not point to the same node, we will increment both fast and slow pointers again. Now, the slow pointer points to node 4 while the fast pointer points to node 7 shown as below:



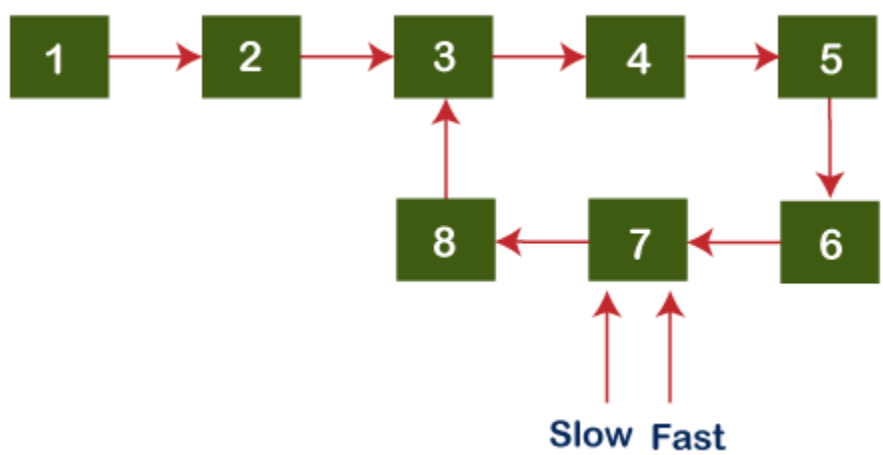
Since both the pointers do not point to the same node, we will increment both fast and slow pointers again. Now, the slow pointer points to node 5, and the fast pointer points to node 3 shown as below:



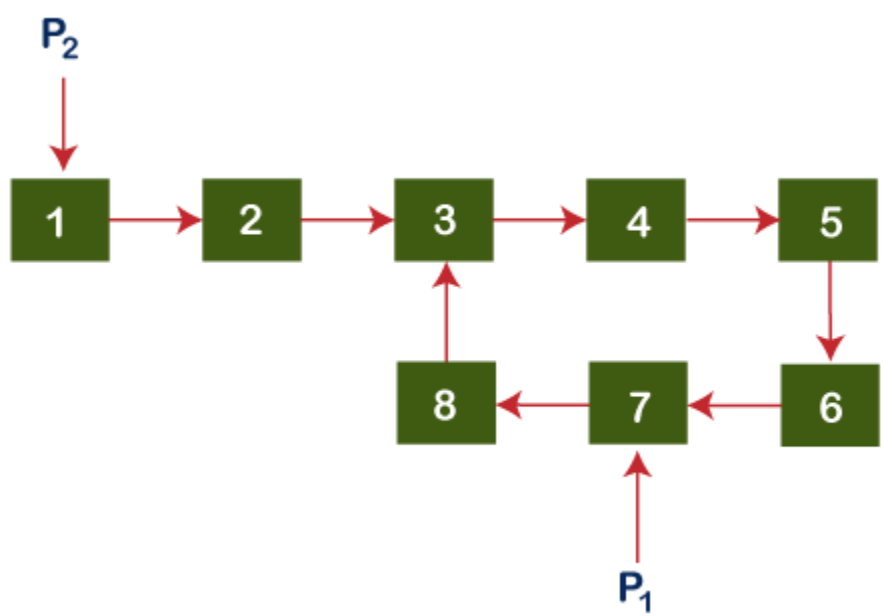
Since both the pointers do not point to the same node, we will increment both fast and slow pointers again. Now, slow pointer points to node 6 while the fast pointer points to the node 5 shown as below:



Again, both the pointers (slow and fast) are not pointing to the same node, so we will increment both fast and slow pointers again. Now, the slow pointer points to node 7, and the fast pointer also points to node 7 shown as below:



As we can observe in the above example both slow and fast pointers meet at node 7. We create one pointer named p_1 that points to the node 7 where both the pointers meet, and we also create one more pointer named p_2 that points to the first node as shown in the below figure:



To remove the loop, we will define the following logic:

1. **while**($p_1.next \neq p_2.next$)
2. {
3. $P_1 = P_1.next$;
4. $P_2 = P_2.next$;
5. }
6. $P_1.next = \text{NULL}$;

The above logic is defined to remove a loop from the linked list. The while loop will execute till the $p_1.next$ is not equal to $p_2.next$. When $p_1.next$ is equal to $p_2.next$, the control will come out of the while loop, and we set the $p_1.next$ equal to Null. This statement breaks the link, which is creating the loop in the linked list.

Implementation of removing a loop in C

1. *// C program to remove a loop from the linked list*
2. `#include <stdio.h>`
3. `#include <stdlib.h>`
4. *// creating a structure of a node*
5. `struct node`
6. {
7. `int data;`
8. `struct node *next;`
9. };
10. `struct node *head;`
11. *// creating a node in a linked list..*
12. `struct node* create_node(int value)`

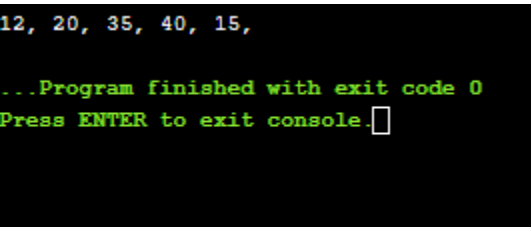
```

13. {
14.  // creating a temp variable of type node..
15.  struct node *temp = (struct node*)malloc(sizeof(struct node));
16.      temp->data = value;
17.      temp->next = NULL;
18.      return temp;
19.
20.
21. }
22. // detecting a node that creates a loop in the linked list..
23. int detect_loop(struct node *ptr)
24. {
25.  struct node *fast; // declaration of fast pointer of type node..
26.  struct node *slow; // declaration of slow pointer of type node..
27.  fast = slow = ptr;
28.  while (slow && fast && fast->next)
29.  {
30.      slow = slow->next;
31.      fast = fast->next->next;
32.      // checking whether slow is equal to fast or not.
33.      if(slow==fast)
34.      {
35.          removeloop(slow, head); // calling removeloop() function.
36.          return 1;
37.      }
38.  }
39. }
40.
41. // Removing a loop from the linked list..
42. void removeloop(struct node *slow, struct node *head)
43. {
44.  struct node *p1;
45.  struct node *p2;
46.  p1 = slow;
47.  p2 = head;
48.
49. // while loop will execute till the p1.next is not equal to p2.next..
50. while(p1->next!=p2->next)
51. {
52.     p1 = p1->next;
53.     p2 = p2->next;
54. }
55. p1->next = NULL;
56.
57. }
58. // display the linked list..
59. void display(struct node *temp)
60. {
61.  while (temp !=NULL)
62.  {
63.      printf("%d, ", temp->data);
64.      temp = temp->next;
65.  }
66. }
67. int main()
68. {
69.     head = create_node(12);

```

```
70. head->next = create_node(20);
71. head->next->next = create_node(35);
72. head->next->next->next = create_node(40);
73. head->next->next->next->next = create_node(15);
74. /* Create a loop for testing */
75. head->next->next->next->next->next = head->next->next;
76. detect_loop(head);
77. display(head);
78. return 0;
79.}
```

Output



Implement two stacks in an array

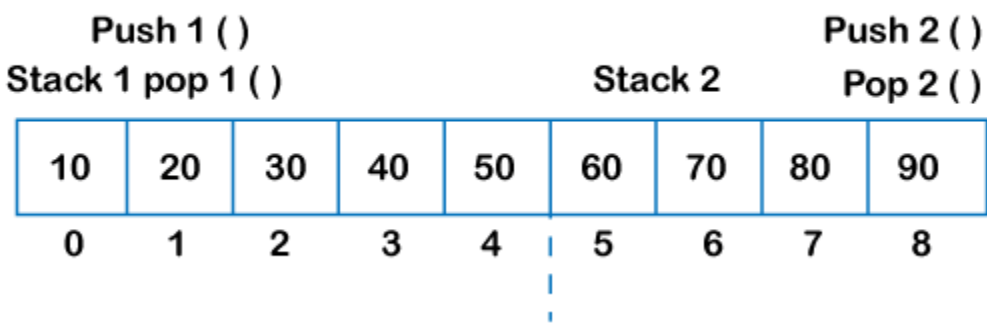
Here, we will create two stacks, and we will implement these two stacks using only one array, i.e., both the stacks would be using the same array for storing elements.

There are two approaches to implement two stacks using one array:

First Approach

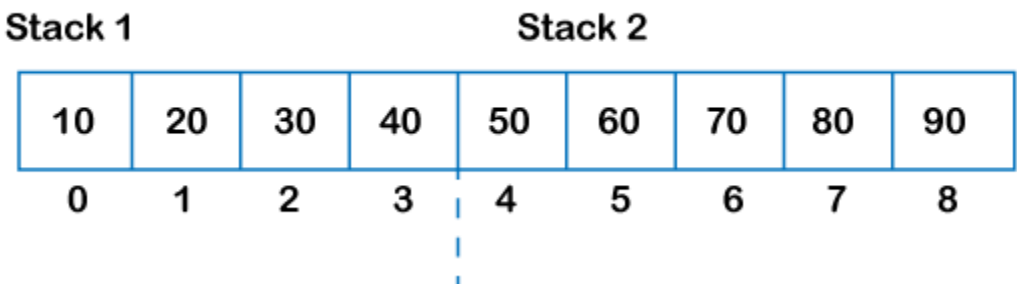
First, we will divide the array into two sub-arrays. The array will be divided into two equal parts. First, the sub-array would be considered stack1 and another sub array would be considered stack2.

For example, if we have an array of n equal to 8 elements. The array would be divided into two equal parts, i.e., 4 size each shown as below:



The first subarray would be stack 1 named as st1, and the second subarray would be stack 2 named as st2. On st1, we would perform push1() and pop1() operations, while in st2, we would perform push2() and pop2() operations. The stack1 would be from 0 to n/2, and stack2 would be from n/2 to n-1.

If the size of the array is odd. For example, the size of an array is 9 then the left subarray would be of 4 size, and the right subarray would be of 5 size shown as below:



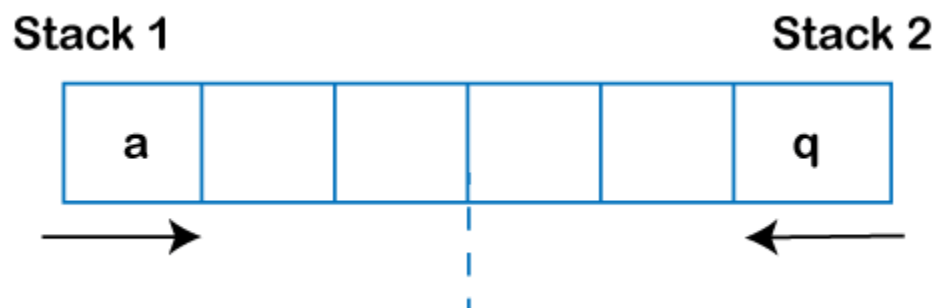
Disadvantage of using this approach

Stack overflow condition occurs even if there is a space in the array. In the above example, if we are performing push1() operation on the stack1. Once the element is inserted at the 3rd index and if we try to insert more elements, then it leads to the overflow error even there is a space left in the array.

Second Approach

In this approach, we are having a single array named as 'a'. In this case, stack1 starts from 0 while stack2 starts from n-1. Both the stacks start from the extreme corners, i.e., Stack1 starts from the leftmost corner (at index 0), and Stack2 starts from the rightmost corner (at index n-1). Stack1 extends in the right direction, and stack2 extends in the left direction, shown as below:

If we push 'a' into stack1 and 'q' into stack2 shown as below:



Therefore, we can say that this approach overcomes the problem of the first approach. In this case, the stack overflow condition occurs only when **top1 + 1 = top2**. This approach provides a space-efficient implementation means that when the array is full, then only it will show the overflow error. In contrast, the first approach shows the overflow error even if the array is not full.

Implementation in C

// C Program to Implement two Stacks using a Single Array & Check for Overflow & Underflow

```
1. #include <stdio.h>
2. #define SIZE 20
3. int array[SIZE]; // declaration of array type variable.
4. int top1 = -1;
5. int top2 = SIZE;
6.
7. //Function to push data into stack1
8. void push1 (int data)
9. {
10. // checking the overflow condition
11. if (top1 < top2 - 1)
12. {
13.     top1++;
14.     array[top1] = data;
15. }
16. else
17. {
18.     printf ("Stack is full");
19. }
20.}
21. // Function to push data into stack2
22. void push2 (int data)
23. {
24. // checking overflow condition
25. if (top1 < top2 - 1)
26. {
27.     top2--;
28.     array[top2] = data;
29. }
30. else
31. {
32.     printf ("Stack is full.\n");
33. }
34.}
```



```
35.
36. //Function to pop data from the Stack1
37. void pop1 ()
38. {
39. // Checking the underflow condition
40. if (top1 >= 0)
41. {
42.     int popped_element = array[top1];
43.     top1--;
44.
45.     printf ("%d is being popped from Stack 1\n", popped_element);
46. }
47. else
48. {
49.     printf ("Stack is Empty \n");
50. }
51.}
52. // Function to remove the element from the Stack2
53. void pop2 ()
54. {
55. // Checking underflow condition
56. if (top2 < SIZE)
57. {
58.     int popped_element = array[top2];
59.     top2--;
60.
61.     printf ("%d is being popped from Stack 1\n", popped_element);
62. }
63. else
64. {
65.     printf ("Stack is Empty!\n");
66. }
67.}
68.
69. //Functions to Print the values of Stack1
70. void display_stack1 ()
71. {
72.     int i;
73.     for (i = top1; i >= 0; --i)
74.     {
75.         printf ("%d ", array[i]);
76.     }
77.     printf ("\n");
78.}
79. // Function to print the values of Stack2
80. void display_stack2 ()
81. {
82.     int i;
83.     for (i = top2; i < SIZE; ++i)
84.     {
85.         printf ("%d ", array[i]);
86.     }
87.     printf ("\n");
88.}
89.
90. int main()
91. {
```

```

92. int ar[SIZE];
93. int i;
94. int num_of_ele;
95.
96. printf ("We can push a total of 20 values\n");
97.
98. //Number of elements pushed in stack 1 is 10
99. //Number of elements pushed in stack 2 is 10
100.
101. // loop to insert the elements into Stack1
102. for (i = 1; i <= 10; ++i)
103. {
104.     push1(i);
105.     printf ("Value Pushed in Stack 1 is %d\n", i);
106. }
107. // loop to insert the elements into Stack2.
108. for (i = 11; i <= 20; ++i)
109. {
110.     push2(i);
111.     printf ("Value Pushed in Stack 2 is %d\n", i);
112. }
113.
114. //Print Both Stacks
115. display_stack1 ();
116. display_stack2 ();
117.
118. //Pushing on Stack Full
119. printf ("Pushing Value in Stack 1 is %d\n", 11);
120. push1 (11);
121.
122. //Popping All Elements from Stack 1
123. num_of_ele = top1 + 1;
124. while (num_of_ele)
125. {
126.     pop1 ();
127.     --num_of_ele;
128. }
129.
130. // Trying to Pop the element From the Empty Stack
131. pop1 ();
132.
133. return 0;
134. }

```

Output

```
We can push a total of 20 values
Value Pushed in Stack 1 is 1
Value Pushed in Stack 1 is 2
Value Pushed in Stack 1 is 3
Value Pushed in Stack 1 is 4
Value Pushed in Stack 1 is 5
Value Pushed in Stack 1 is 6
Value Pushed in Stack 1 is 7
Value Pushed in Stack 1 is 8
Value Pushed in Stack 1 is 9
Value Pushed in Stack 1 is 10
Value Pushed in Stack 2 is 11
Value Pushed in Stack 2 is 12
Value Pushed in Stack 2 is 13
Value Pushed in Stack 2 is 14
Value Pushed in Stack 2 is 15
Value Pushed in Stack 2 is 16
Value Pushed in Stack 2 is 17
Value Pushed in Stack 2 is 18
Value Pushed in Stack 2 is 19
Value Pushed in Stack 2 is 20
10 9 8 7 6 5 4 3 2 1
20 19 18 17 16 15 14 13 12 11
Pushing Value in Stack 1 is 11
Stack Full! Cannot Push
10 is being popped from Stack 1
9 is being popped from Stack 1
8 is being popped from Stack 1
7 is being popped from Stack 1
6 is being popped from Stack 1
5 is being popped from Stack 1
4 is being popped from Stack 1
3 is being popped from Stack 1
2 is being popped from Stack 1
```

Reverse a stack using recursion

Here, we are going to reverse a stack using recursion. We are not supposed to use any loop constructs like for loop, while loop, do-while loop, etc. We should use the recursion method to reverse a stack.

For example: input: s = [10, 20, 30, 40, 50]

Output: [50, 40, 30, 20, 10]

Explanation: When the stack s is reversed then the output would be [50, 40, 30, 20, 10].

There are various ways to reverse a stack using recursion. The most common way of reversing a stack is to use an auxiliary stack. First, we will pop all the elements from the stack and push them into the auxiliary stack. Once all the elements are pushed into the auxiliary stack, then it contains the elements in the reverse order and we simply print them. But, here, we will not use the auxiliary stack. We will use a recursion method to reverse a stack where recursion means calling the function itself again and again.

In the recursion method, we first pop all the elements from the input stack and push all the popped items into the function call stack until the stack becomes empty. When the stack becomes empty, all the items will be pushed at the stack. Let's understand this scenario through an example.

For example:

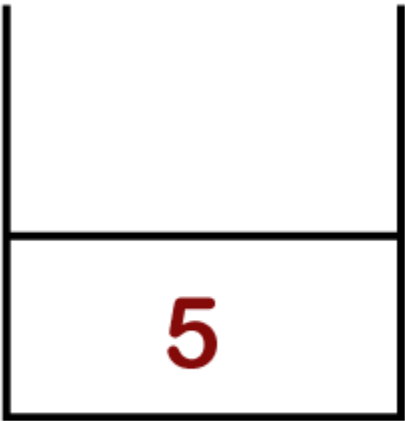
Input stack: 1, 2, 3, 4, 5



Output: 5, 4, 3, 2, 1

Solution: Firstly, all the elements from the input stack are pushed into the function call stack

Step 1: Element 5 is pushed at the bottom of the stack shown as below:



Step 2: Element 4 is pushed at the bottom of the stack shown as below:



Step 3: Element 3 is pushed at the bottom of the stack shown as below:



Step 4: Element 2 is pushed at the bottom of the stack shown as below:



Step 5: Element 1 is pushed at the bottom of the stack shown as below:



Implementation in C

```
1. #include <stdio.h>
2. #define MAXSIZE 10
3. #define TRUE 1
4. #define FALSE 0
5. // Defining the structure of stack type
6. struct Stack {
7.     int top;
8.     int array[MAXSIZE];
```

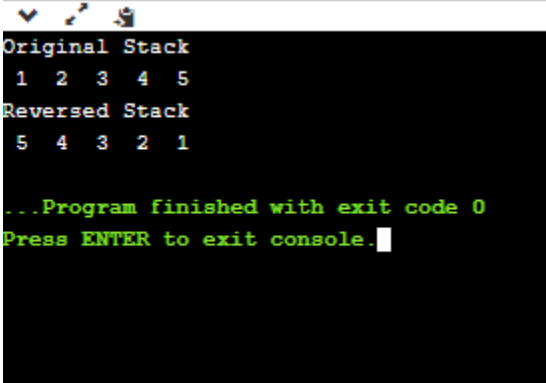
```
9. } st;
10.
11. // initialization of top variable
12. void initialize() {
13.     st.top = -1;
14. }
15. // checking whether the stack is full or not
16. int isFull() {
17.     if(st.top >= MAXSIZE-1)
18.         return TRUE;
19.     else
20.         return FALSE;
21. }
22. // checking whether the stack is empty or not
23. int isEmpty() {
24.     if(st.top == -1)
25.         return TRUE;
26.     else
27.         return FALSE;
28. }
29.
30. // function to push the element into the stack.
31. void push(int num) {
32.     if (isFull())
33.         printf("Stack is Full...\n");
34.     else {
35.         st.array[st.top + 1] = num;
36.         st.top++;
37.     }
38. }
39.
40. // function to pop the element from the stack
41. int pop() {
42.     if (isEmpty())
43.         printf("Stack is Empty...\n");
44.     else {
45.         st.top = st.top - 1;
46.         return st.array[st.top+1];
47.     }
48. }
49.
50. // function to print the elements of stack.
51. void printStack(){
52. // condition to check whether the stack is empty or not.
53. if(!isEmpty())
54. {
55.     int temp = pop();
56.     printStack();
57.     printf(" %d ", temp);
58.     push(temp);
59. }
60. }
61. // function to insert the element at the bottom of the stack.
62. void insertAtBottom(int item) {
63.     if (isEmpty()) {
64.         push(item);
65.     } else {
```

```

66.     int top = pop();
67.     insertAtBottom(item);
68.     push(top);
69. }
70.}
71. // function to reverse the order of the stack.
72. void reverse() {
73.     if (!isEmpty()) {
74.         int top = pop();
75.         reverse();
76.         insertAtBottom(top);
77.     }
78.}
79.
80. int getSize(){
81.     return st.top+1;
82.}
83. // Definition of main() method
84. int main() {
85. initialize(st); // calling initialize() method
86. push(1); // calling push() method
87. push(2); // calling push() method
88. push(3); // calling push() method
89. push(4); // calling push() method
90. push(5); // calling push() method
91. printf("Original Stack\n");
92. printStack();
93. reverse();
94. printf("\nReversed Stack\n");
95. printStack(); // calling printStack() method
96. return 0;
97.}

```

Output



```

Original Stack
1 2 3 4 5
Reversed Stack
5 4 3 2 1

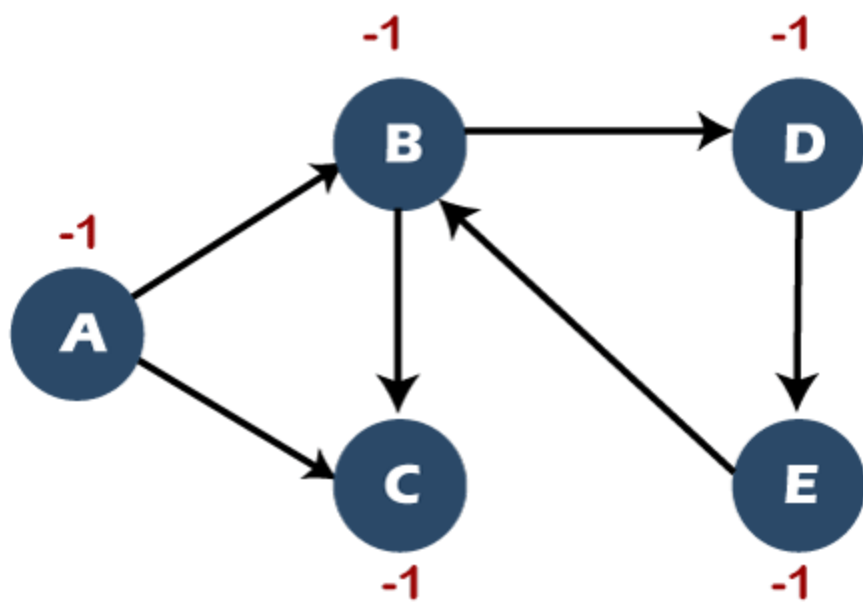
...Program finished with exit code 0
Press ENTER to exit console.

```

Detect cycle in a directed graph

In a directed graph, we will check whether the graph contains cycle or not. A directed graph is a set of vertices or nodes connected by edges, and each edge is associated with some direction.

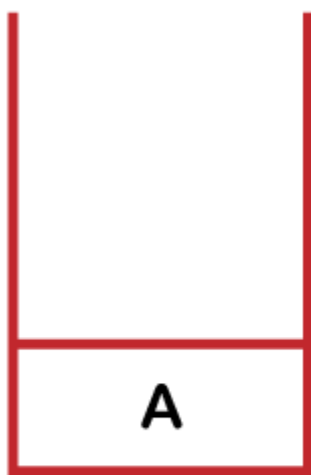
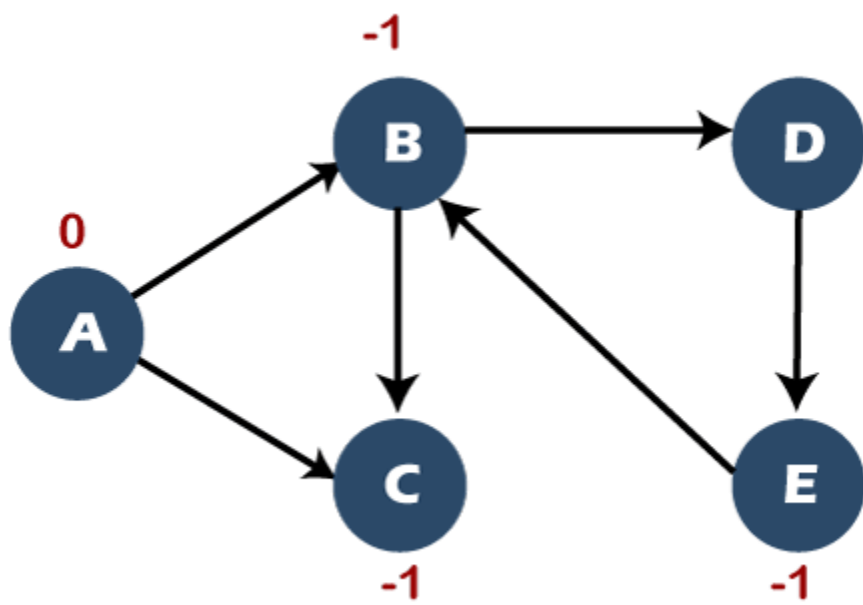
Consider the below directed graph to detect the cycle.



Now, we will use the [DFS](#) technique to detect cycles in a directed graph. As we are using the DFS technique, so we will use the [stack data structure](#) for the implementation. Here, we will use the flag variable that contains three values, i.e., 0, 1, and -1. Here, 0 means that the node is visited and available in the stack, -1 means that the node is unvisited, and 1 means that the node is visited and has been popped out from the stack.

Initially, all the vertices are marked with -1 as they all are not visited.

Step 1: First, we will visit vertex A and will be marked as 0. Since node A has been visited so it will be marked as 0, and node A is pushed into the stack shown as below:



Stack

The visited set contains the node A shown as below:

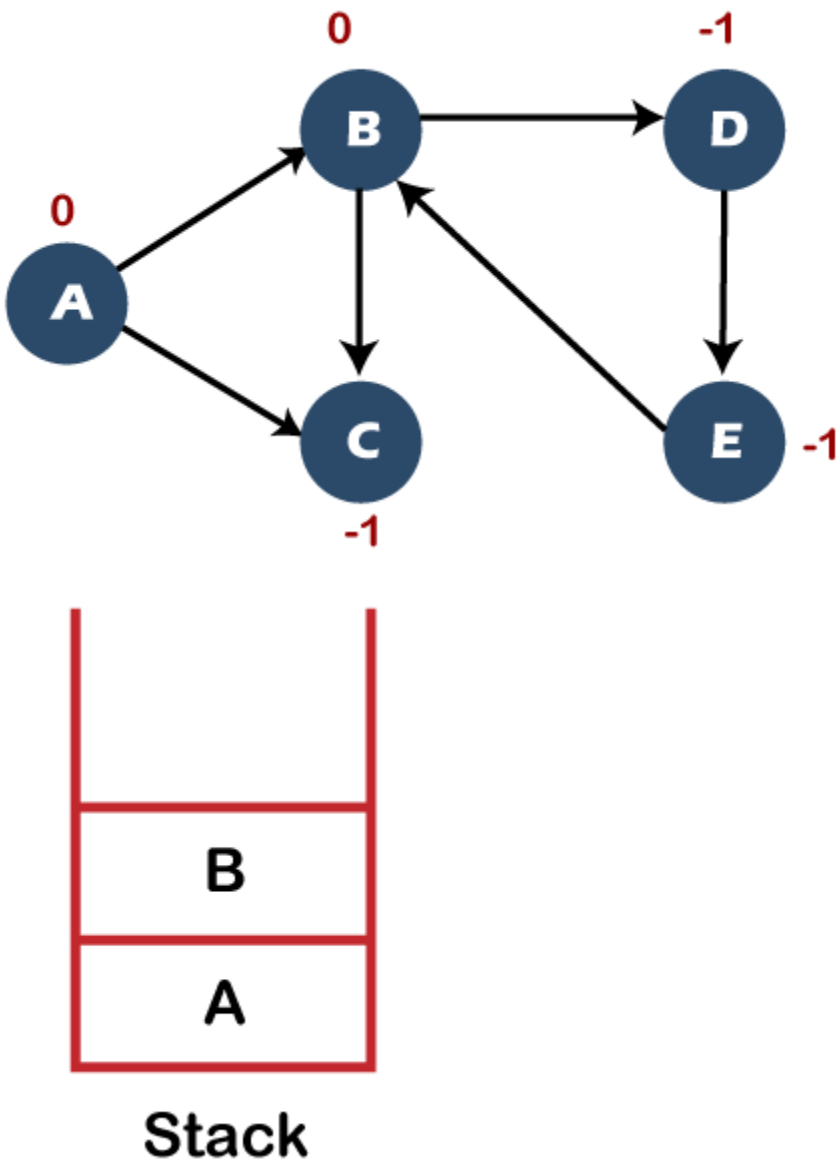
Visited set: A

The parent map table is given below:

Vertex	Parent
A	-

Since node A has been visited, A comes under the vertex column and the parent column is left blank as node A is a source vertex.

Step 2: The next vertex is B. Now, we will visit vertex B and will be marked as 0. Since the node B has been visited so it will be marked as 0, and node B is pushed into the stack shown as below:



The visited node contains the nodes A and B shown as below:

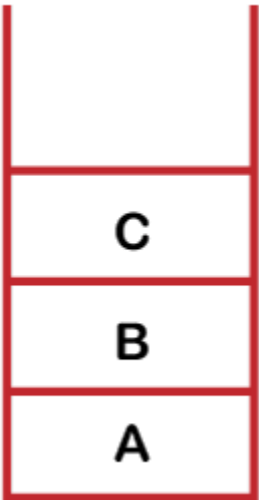
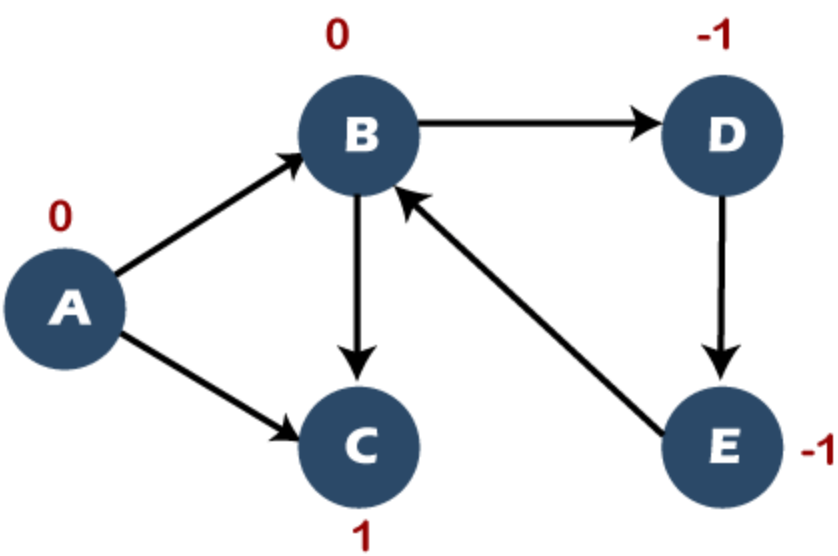
Visited set: A, B

The parent map table is shown below:

Vertex	Parent
A	-
B	A

Since node B has been visited, so B comes under the vertex column, and A comes under the parent column as B comes from node A.

Step 3: The adjacent vertices of B are C and D means we can either visit C or D. Suppose we visit vertex C, so vertex C will be marked as 0 and node C is pushed into the stack shown as below:



Now, the visited set contains the nodes A, B, and C shown as below:

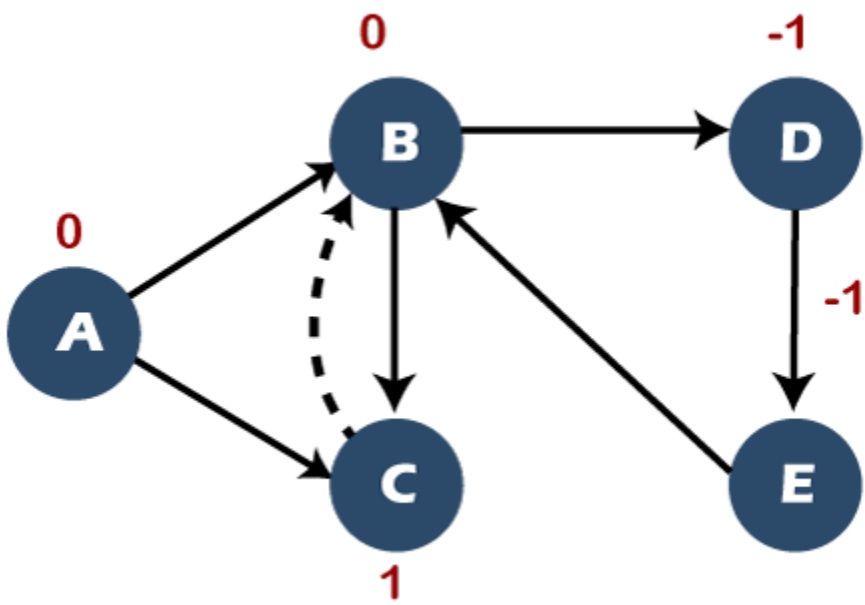
Visited set: A, B, C

The parent map table is shown below:

Vertex	Parent
A	-
B	A
C	B

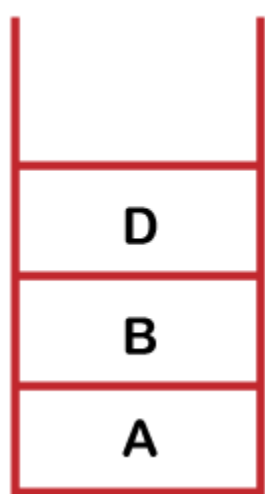
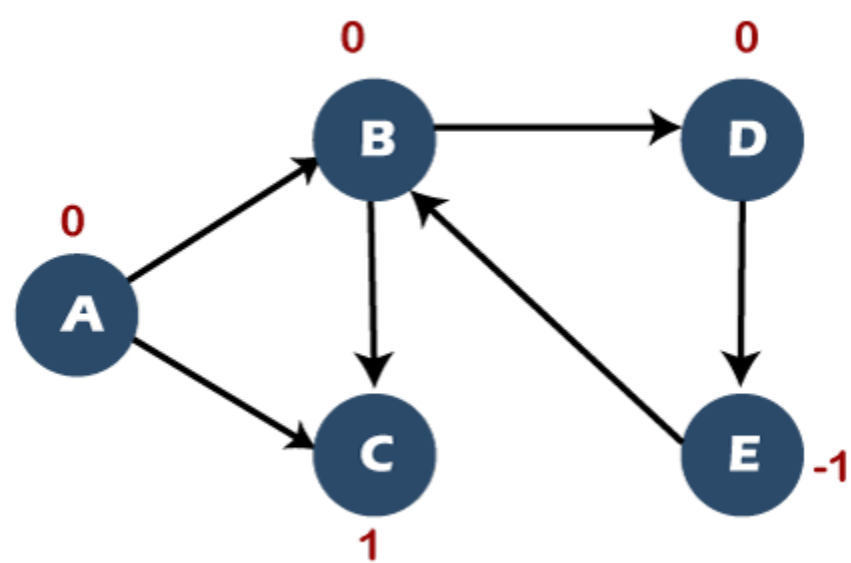
Since node C has been visited, so C comes under the vertex column, and B comes under the parent column.

Step 4: There are no further vertices to be visited from C, so we will perform backtracking. In order to perform backtracking, we will pop the node. First, we will pop node C from the stack. Since node C has been popped out, so we will mark the node C as 1 shown as below:



The next topmost node in the stack is B, so we will backtrack to the vertex B shown as below:

Step 5: Now, we will see 'is there any adjacent vertex left to be visited'. We can observe in the above graph that the vertex D is left unvisited. So, now we will move from the vertex B to the vertex D. The flag value of vertex D is now changed to 0, and the vertex D is pushed into the stack shown as below:



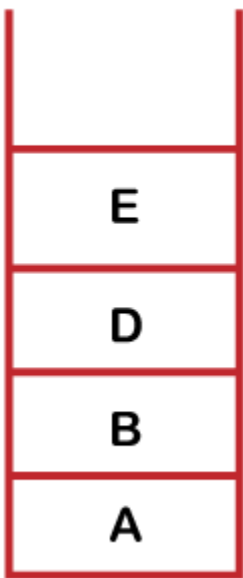
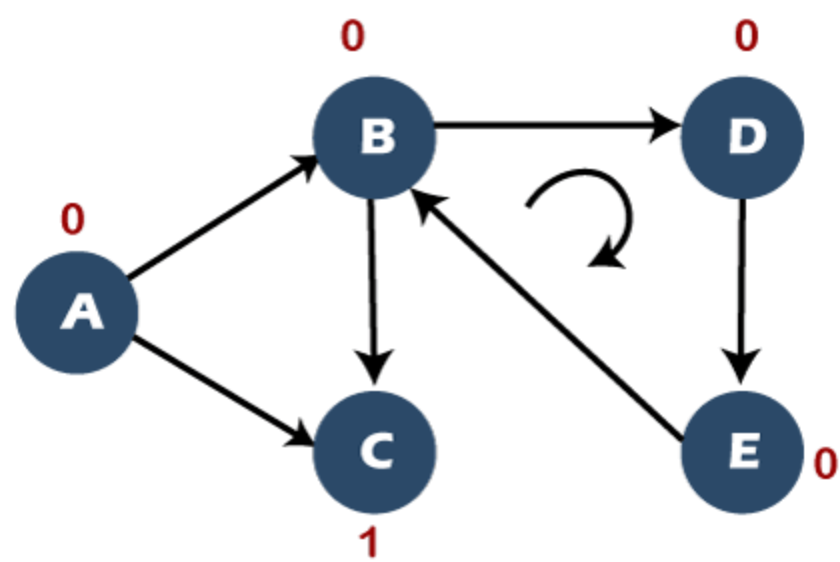
Now the visited set contains the nodes A, B, C, D

The parent map table is shown below:

Vertex	Parent
A	-
B	A
C	B
D	B

Since node D has been visited, it comes under the vertex column, and the node D has arrived from vertex B, so vertex B comes under the parent column.

Step 6: The adjacent vertex of node D is node E which is left unvisited. Now we will visit the vertex E and will mark its flag value as 0. The node E is pushed into the stack shown as below:



Now, the visited set contains the nodes A, B, C, D, E.

The parent map table is shown below:

Vertex	Parent
A	-
B	A
C	B
D	B
E	D

Since node E has been visited, it comes under the vertex column, and node E has arrived from the vertex D, so D comes under the parent column.

Condition

There is one condition that determines whether the graph contains a cycle or not. If the adjacent vertex of any vertex is having a 0 flag value means that the graph contains a cycle.

In the above graph, the adjacent vertex of E is B, and its flag value is 0; therefore, the graph contains a cycle.

Now we will determine the nodes that create a cycle in a graph.

The adjacent vertex of E is B;

E->B

The parent of E is D so;

D->E->B

The parent of D is B so,

B->D->E->B (creates a cycle)

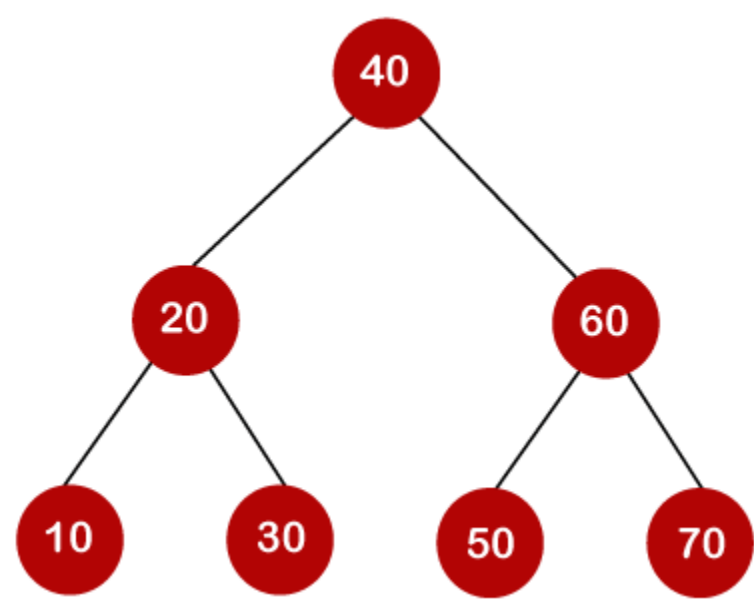
Optimal Binary Search Tree

As we know that in binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node.

We know the key values of each node in the tree, and we also know the frequencies of each node in terms of searching means how much time is required to search a node. The frequency and key-value determine the overall cost of searching a node. The cost of searching is a very important factor in various applications. The overall cost of searching a node should be less. The time required to search a node in BST is more than the balanced binary search tree as a balanced binary search tree contains a lesser number of levels than the BST. There is one way that can reduce the cost of a **binary search tree** is known as an **optimal binary search tree**.

Let's understand through an example.

If the keys are 10, 20, 30, 40, 50, 60, 70



In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node. The maximum time required to search a node is equal to the minimum height of the tree, equal to $\log n$.

Now we will see how many binary search trees can be made from the given number of keys.

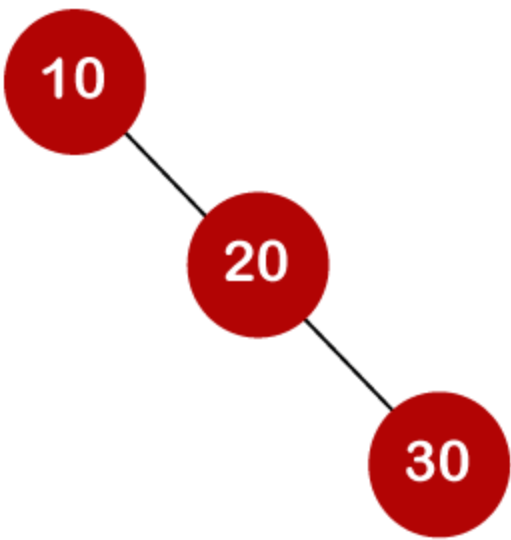
For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.

The Formula for calculating the number of trees:

$$\frac{2n}{n+1} C_n$$

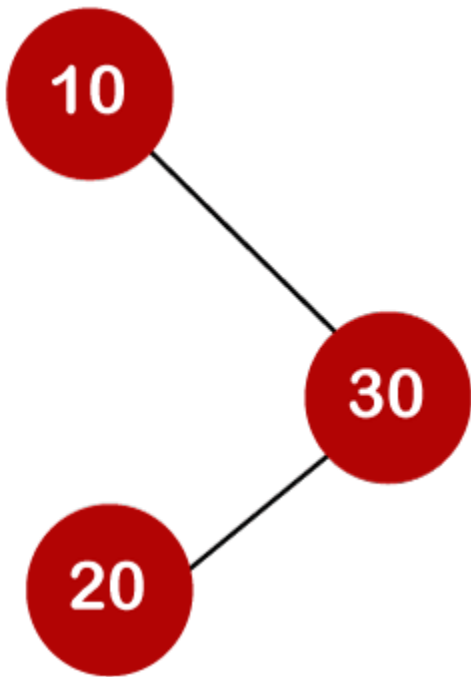
When we use the above formula, then it is found that total 5 number of trees can be created.

The cost required for searching an element depends on the comparisons to be made to search an element. Now, we will calculate the average cost of time of the above binary search trees.



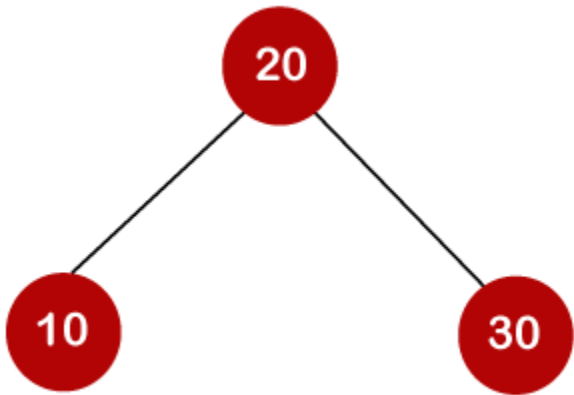
In the above tree, total number of 3 comparisons can be made. The average number of comparisons can be made as:

average number of comparisons = $\frac{1+2+3}{3} = 2$



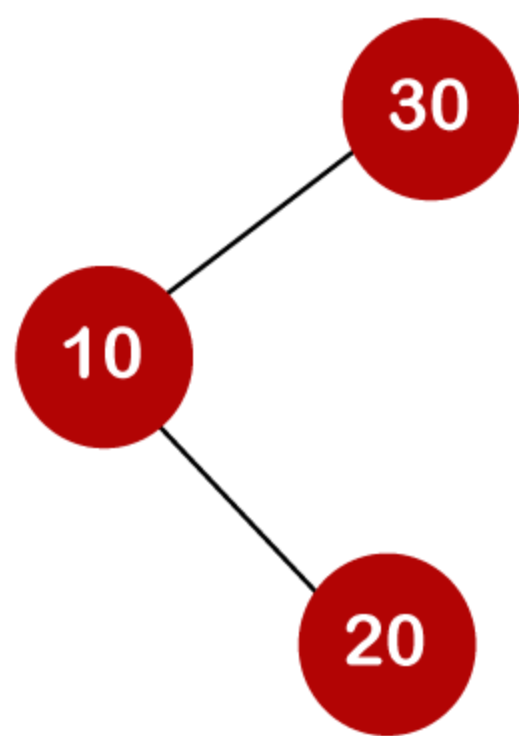
In the above tree, the average number of comparisons that can be made as:

average number of comparisons = $\frac{1+2+3}{3} = 2$



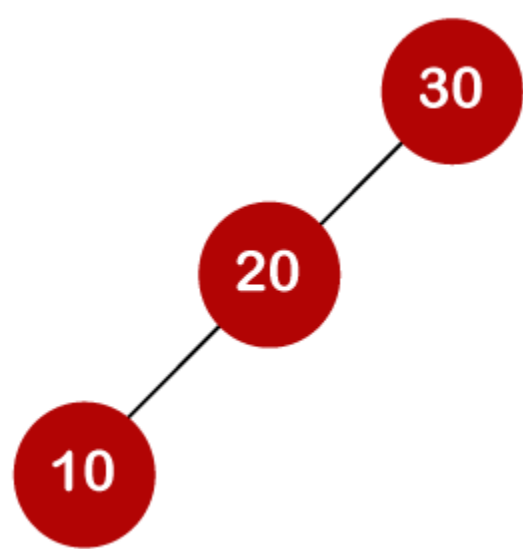
In the above tree, the average number of comparisons that can be made as:

$$\text{average number of comparisons} = \frac{1+2+2}{3} = 5/3$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$\text{average number of comparisons} = \frac{1+2+3}{3} = 2$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$\text{average number of comparisons} = \frac{1+2+3}{3} = 2$$

In the third case, the number of comparisons is less because the height of the tree is less, so it's a balanced binary search tree.

Till now, we read about the height-balanced binary search tree. To find the optimal binary search tree, we will determine the frequency of searching a key.

Let's assume that frequencies associated with the keys 10, 20, 30 are 3, 2, 5.

The above trees have different frequencies. The tree with the lowest frequency would be considered the optimal binary search tree. The tree with the frequency 17 is the lowest, so it would be considered as the optimal binary search tree.

Dynamic Approach

Consider the below table, which contains the keys and frequencies.

	1	2	3	4	
Keys →	10	20	30	40	
Frequency →	4	2	6	3	

i \ j	0	1	2	3	4
0					
1					
2					
3					
4					

First, we will calculate the values where j-i is equal to zero.

When i=0, j=0, then j-i = 0

When i = 1, j=1, then j-i = 0

When i = 2, j=2, then j-i = 0

When i = 3, j=3, then j-i = 0

When i = 4, j=4, then j-i = 0

Therefore, c[0, 0] = 0, c[1 , 1] = 0, c[2,2] = 0, c[3,3] = 0, c[4,4] = 0

Now we will calculate the values where j-i equal to 1.

When j=1, i=0 then j-i = 1

When j=2, i=1 then j-i = 1

When j=3, i=2 then j-i = 1

When j=4, i=3 then j-i = 1

Now to calculate the cost, we will consider only the jth value.

The cost of c[0,1] is 4 (The key is 10, and the cost corresponding to key 10 is 4).

The cost of c[1,2] is 2 (The key is 20, and the cost corresponding to key 20 is 2).

The cost of c[2,3] is 6 (The key is 30, and the cost corresponding to key 30 is 6)

The cost of c[3,4] is 3 (The key is 40, and the cost corresponding to key 40 is 3)

	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

Now we will calculate the values where $j-i = 2$

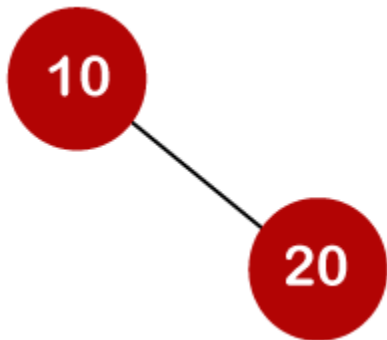
When $j=2, i=0$ then $j-i = 2$

When $j=3, i=1$ then $j-i = 2$

When $j=4, i=2$ then $j-i = 2$

In this case, we will consider two keys.

- When $i=0$ and $j=2$, then keys 10 and 20. There are two possible trees that can be made out from these two keys shown below:



In the first binary tree, cost would be: $4*1 + 2*2 = 8$

In the second binary tree, cost would be: $4*2 + 2*1 = 10$

The minimum cost is 8; therefore, $c[0,2] = 8$

	0	1	2	3	4
0	0	4	8		
1		0	2		
2			0	6	
3				0	3
4					0

- When $i=1$ and $j=3$, then keys 20 and 30. There are two possible trees that can be made out from these two keys shown below:

In the first binary tree, cost would be: $1*2 + 2*6 = 14$

In the second binary tree, cost would be: $1*6 + 2*2 = 10$

The minimum cost is 10; therefore, $c[1,3] = 10$

- When $i=2$ and $j=4$, we will consider the keys at 3 and 4, i.e., 30 and 40. There are two possible trees that can be made out from these two keys shown as below:

In the first binary tree, cost would be: $1*6 + 2*3 = 12$

In the second binary tree, cost would be: $1*3 + 2*6 = 15$

The minimum cost is 12, therefore, $c[2,4] = 12$

i \ j	1	0	1	2	3	4
0		0	4	8^1		
1			0	2	10^3	
2				0	6	12^3
3					0	3
4						0

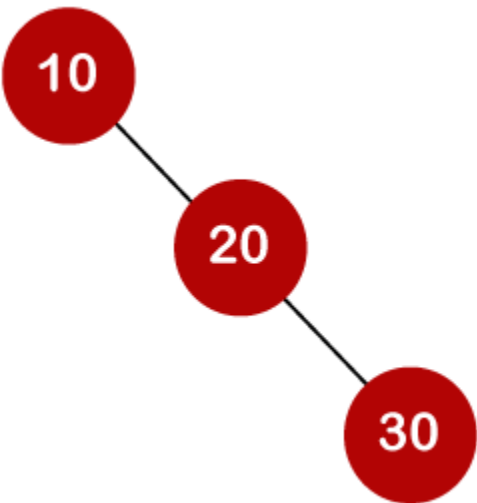
Now we will calculate the values when $j-i = 3$

When $j=3$, $i=0$ then $j-i = 3$

When $j=4$, $i=1$ then $j-i = 3$

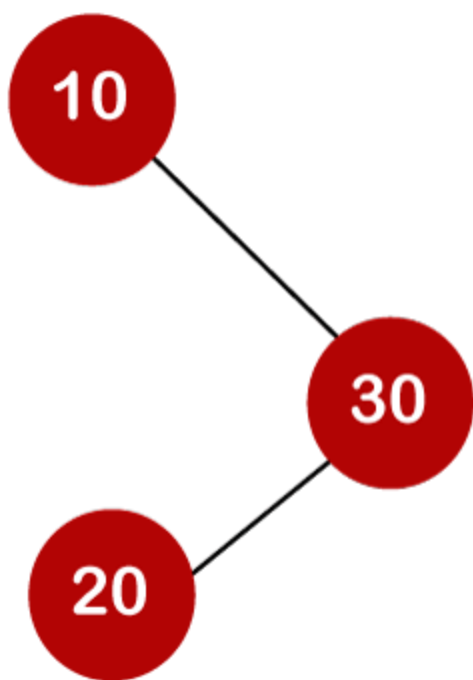
- When $i=0$, $j=3$ then we will consider three keys, i.e., 10, 20, and 30.

The following are the trees that can be made if 10 is considered as a root node.



In the above tree, 10 is the root node, 20 is the right child of node 10, and 30 is the right child of node 20.

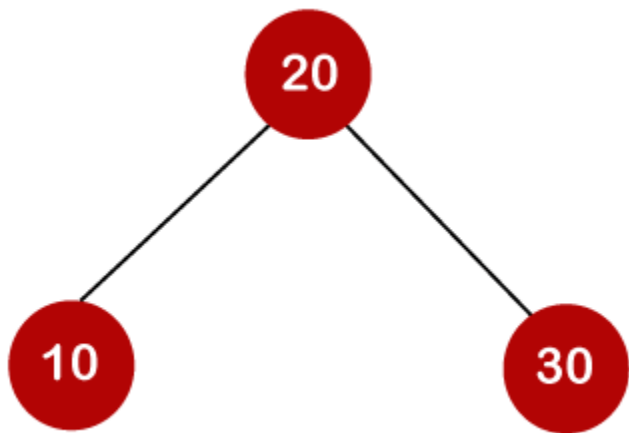
Cost would be: $1*4 + 2*2 + 3*6 = 26$



In the above tree, 10 is the root node, 30 is the right child of node 10, and 20 is the left child of node 20.

Cost would be: $1 \cdot 4 + 2 \cdot 6 + 3 \cdot 2 = 22$

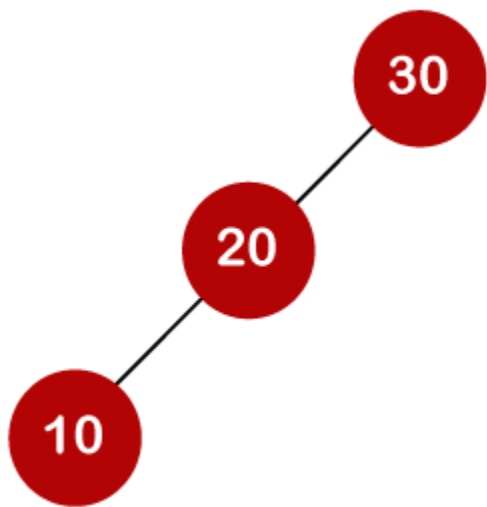
The following tree can be created if 20 is considered as the root node.



In the above tree, 20 is the root node, 30 is the right child of node 20, and 10 is the left child of node 20.

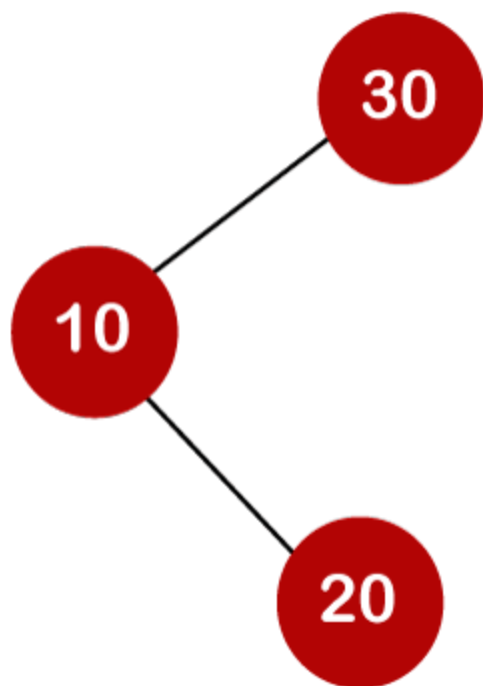
Cost would be: $1 \cdot 2 + 4 \cdot 2 + 6 \cdot 2 = 22$

The following are the trees that can be created if 30 is considered as the root node.



In the above tree, 30 is the root node, 20 is the left child of node 30, and 10 is the left child of node 20.

Cost would be: $1 \cdot 6 + 2 \cdot 2 + 3 \cdot 4 = 22$



In the above tree, 30 is the root node, 10 is the left child of node 30 and 20 is the right child of node 10.

Cost would be: $1 \cdot 6 + 2 \cdot 4 + 3 \cdot 2 = 20$

Therefore, the minimum cost is 20 which is the 3rd root. So, $c[0,3]$ is equal to 20.

- When $i=1$ and $j=4$ then we will consider the keys 20, 30, 40

$$c[1,4] = \min\{ c[1,1] + c[2,4], c[1,2] + c[3,4], c[1,3] + c[4,4] \} + 11$$

$$= \min\{0+12, 2+3, 10+0\} + 11$$

$$= \min\{12, 5, 10\} + 11$$

The minimum value is 5; therefore, $c[1,4] = 5+11 = 16$

i \ j	1	0	1	2	3	4
0	0	4	8^1	20^3		
1		0	2	10^3	16^3	
2			0	6	12^3	
3				0	3	
4						0

- **Now we will calculate the values when $j-i = 4$**

When $j=4$ and $i=0$ then $j-i = 4$

In this case, we will consider four keys, i.e., 10, 20, 30 and 40. The frequencies of 10, 20, 30 and 40 are 4, 2, 6 and 3 respectively.

$$w[0, 4] = 4 + 2 + 6 + 3 = 15$$

If we consider 10 as the root node then

$$C[0, 4] = \min\{c[0,0] + c[1,4]\} + w[0,4]$$

$$= \min\{0 + 16\} + 15 = 31$$

If we consider 20 as the root node then

$$C[0,4] = \min\{c[0,1] + c[2,4]\} + w[0,4]$$

= min{4 + 12} + 15

= 16 + 15 = 31

If we consider 30 as the root node then,

$C[0,4] = \min\{c[0,2] + c[3,4]\} + w[0,4]$

= min {8 + 3} + 15

= 26

If we consider 40 as the root node then,

$C[0,4] = \min\{c[0,3] + c[4,4]\} + w[0,4]$

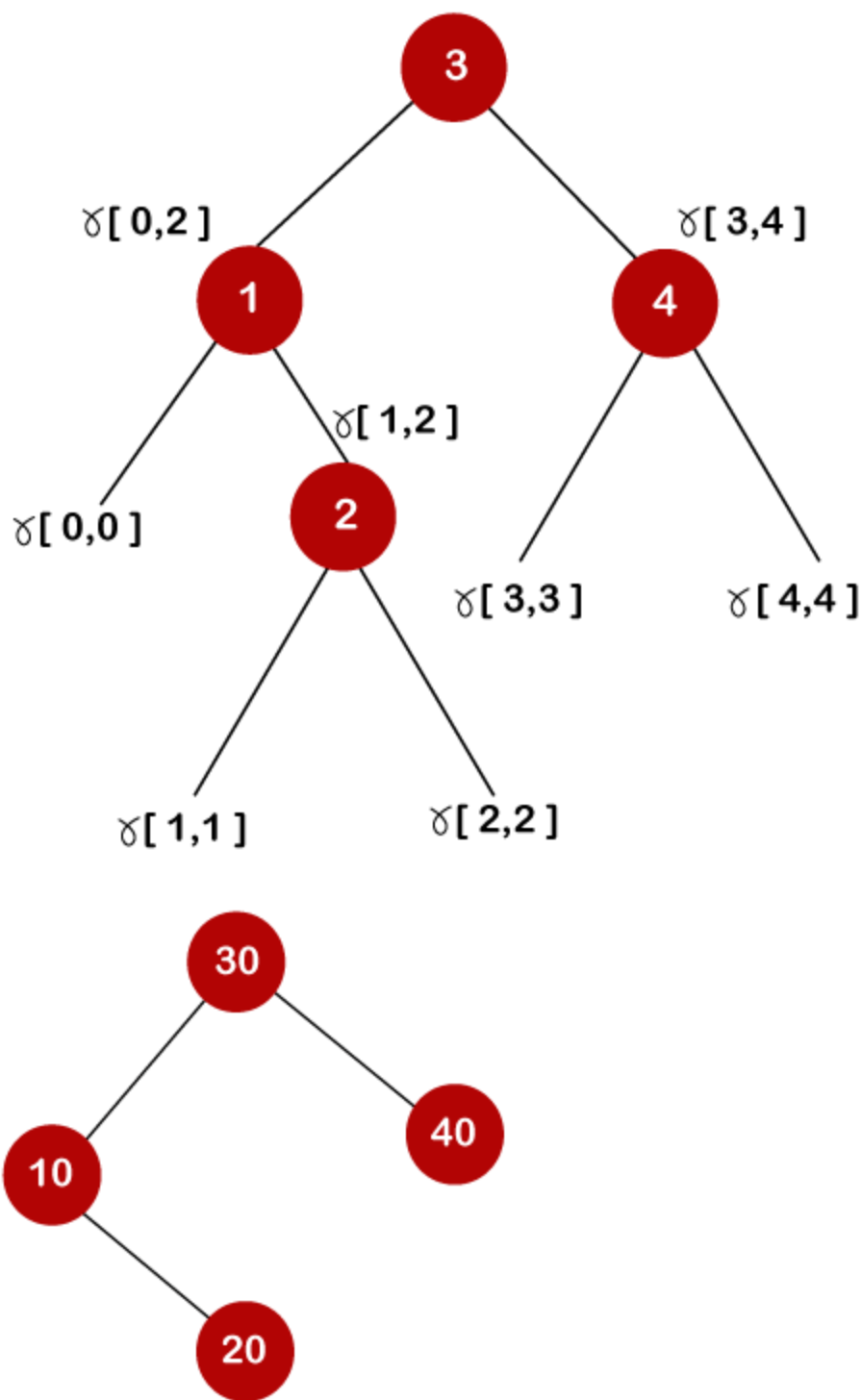
= min{20 + 0} + 15

= 35

In the above cases, we have observed that 26 is the minimum cost; therefore, c[0,4] is equal to 26.

i \ j	1	0	1	2	3	4
0		0	4	8 ¹	20 ³	26 ³
1			0	2	10 ³	16 ³
2				0	6	12 ³
3					0	3
4						0

The optimal binary tree can be created as:



General formula for calculating the minimum cost is:

$$C[i,j] = \min\{c[i, k-1] + c[k,j]\} + w(i,j)$$

Priority Queue using Linked list

A priority queue is a type of queue in which each element in a queue is associated with some priority, and they are served based on their priorities. If the elements have the same priority, they are served based on their order in a queue.

Mainly, the value of the element can be considered for assigning the priority. For example, the highest value element can be used as the highest priority element. We can also assume the lowest value element to be the highest priority element. In other cases, we can also set the priority based on our needs.

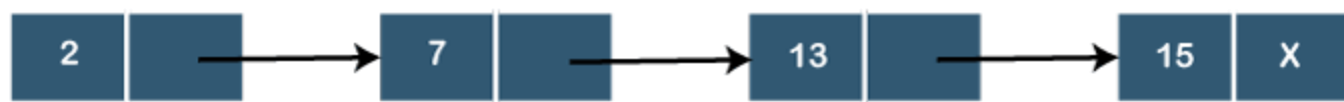
The following are the functions used to implement priority queue using linked list:

- **push():** It is used to insert a new element into the Queue.
- **pop():** It removes the highest priority element from the Queue.
- **peep():** This function is used to retrieve the highest priority element from the queue without removing it from the queue.

The [linked list](#) of [priority queue](#) is created in such a way that the highest priority element is always added at the head of the queue. The elements are arranged in a descending order based on their priority so that it takes **O(1)** time in deletion. In case of insertion, we need to traverse the whole list in order to find out the suitable position based on their priority; so, this process takes O(N) time.

Let's understand through an example.

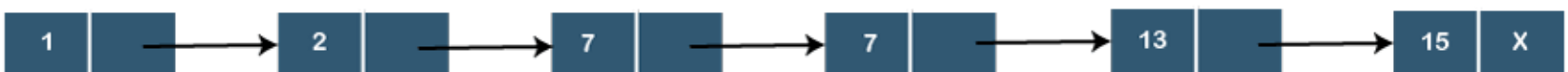
Consider the below-linked list that consists of elements 2, 7, 13, 15.



Suppose we want to add the node that contains the value 1. Since the value 1 has more priority than the other nodes so we will insert the node at the beginning of the list shown as below:



Now we have to add 7 element to the linked list. We will traverse the list to insert element 7. First, we will compare element 7 with 1; since 7 has lower priority than 1, so it will not be inserted before 7. Element 7 will be compared with the next node, i.e., 2; since element 7 has a lower priority than 2, it will not be inserted before 2.. Now, the element 7 is compared with a next element, i.e., since both the elements have the same priority so they will be served based on the first come first serve. The new element 7 will be added after the element 7 shown as below:



Implementation in C

```

1. #include<stdio.h>
2. #include<malloc.h>
3.
4. typedef struct node
5. {
6.     int priority;
7.     int info;
8.     struct node *link;
9. }NODE;
10. NODE *front = NULL;
11.
12. // insert method
13. void insert(int data,int priority)
14. {
15.     NODE *temp,*q;
16.
17.     temp = (NODE *)malloc(sizeof(NODE));
18.     temp->info = data;
19.     temp->priority = priority;
20.     // condition to check whether the first element is empty or the element to be inserted has more priority than the
    first element
21.     if( front == NULL || priority < front->priority )
22.     {
23.         temp->link = front;
24.         front = temp;
25.     }
26.     else
27.     {
28.         q = front;
29.         while( q->link != NULL && q->link->priority <= priority )
30.             q=q->link;
31.         temp->link = q->link;
32.         q->link = temp;
33.     }
34. }
35.
  
```

```


36. // delete method
37.
38. void del()
39. {
40.     NODE *temp;
41.     // condition to check whether the Queue is empty or not
42.     if(front == NULL)
43.         printf("Queue Underflow\n");
44.     else
45.     {
46.         temp = front;
47.         printf("Deleted item is %d\n", temp->info);
48.         front = front->link;
49.         free(temp);
50.     }
51.
52. // display method
53. void display()
54. {
55.     NODE *ptr;
56.     ptr = front;
57.     if(front == NULL)
58.         printf("Queue is empty\n");
59.     else
60.     {
61.         printf("Queue is :\n");
62.         printf("Priority    Item\n");
63.         while(ptr != NULL)
64.         {
65.             printf("%5d    %5d\n",ptr->priority,ptr->info);
66.             ptr = ptr->link;
67.         }
68.     }
69. }
70. /*End of display*/
71.
72. // main method
73. int main()
74. {
75.     int choice, data, priority;
76.     do
77.     {
78.         printf("1.Insert\n");
79.         printf("2.Delete\n");
80.         printf("3.Display\n");
81.         printf("4.Quit\n");
82.         printf("Enter your choice : ");
83.         scanf("%d", &choice);
84.         switch(choice)
85.         {
86.             case 1:
87.                 printf("Enter the data which is to be added in the queue : ");
88.                 scanf("%d",&data);
89.                 printf("Enter its priority : ");
90.                 scanf("%d",&priority);
91.                 insert(data,priority);
92.                 break;

```



```
93.         case 2:
94.             del();
95.             break;
96.         case 3:
97.             display();
98.             break;
99.         case 4:
100.             break;
101.         default :
102.             printf("Wrong choice\n");
103.     }
104. }while(choice!=4);
105.
106.     return 0;
107. }
```

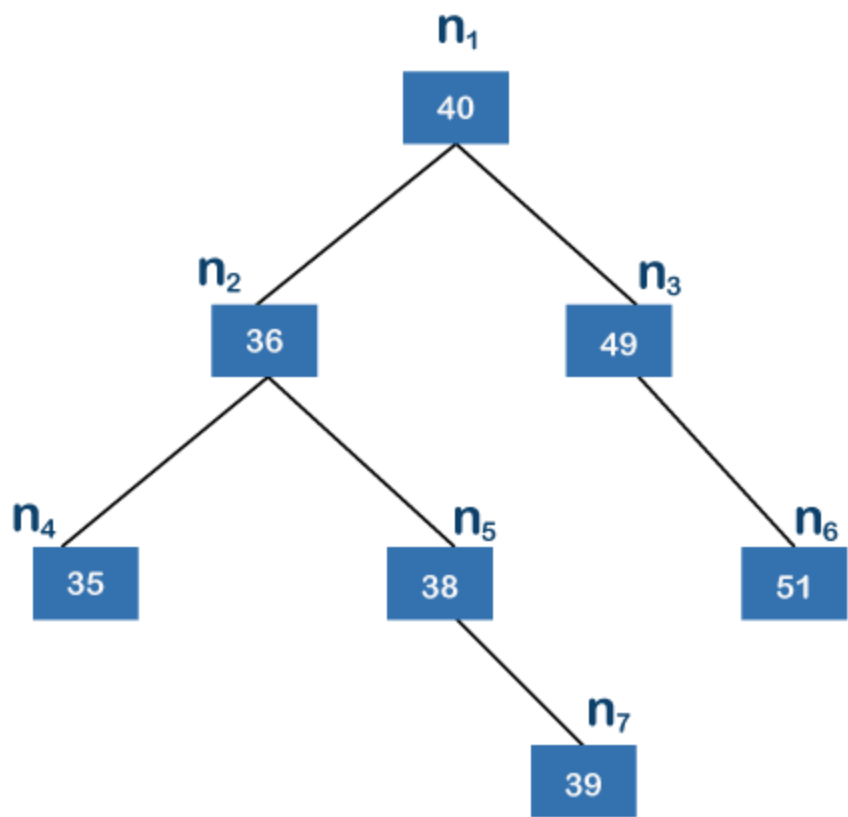
Output



```
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 1
Enter the data which is to be added in the queue : 23
Enter its priority : 1
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 1
Enter the data which is to be added in the queue : 45
Enter its priority : 2
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice : 3
Queue is :
Priority      Item
    1         23
    2         45
1.Insert
2.Delete
3.Display
4.Quit
Enter your choice :
```

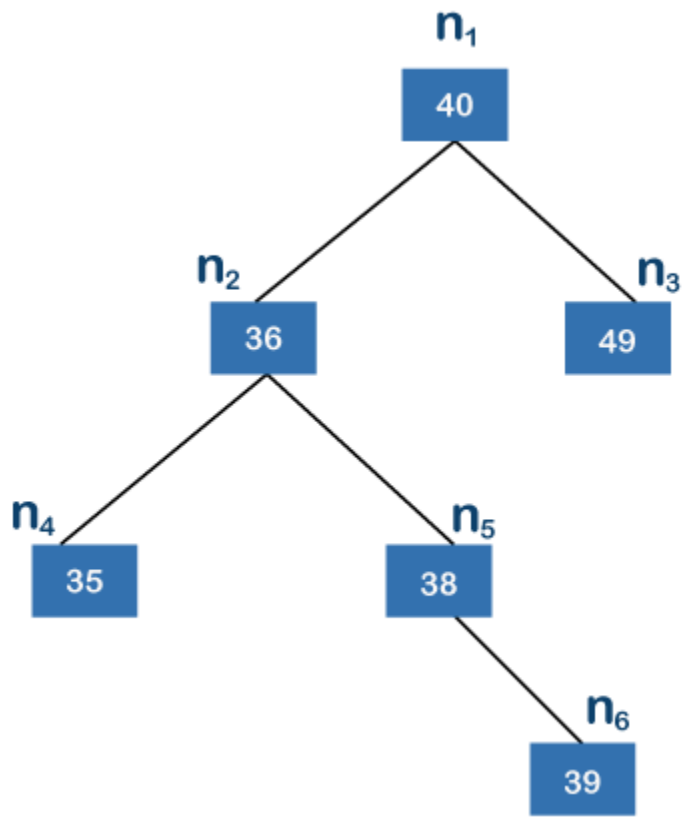
Balanced Binary Search Tree

A balanced binary tree is also known as height balanced tree. It is defined as binary tree in when the difference between the height of the left subtree and right subtree is not more than m, where m is usually equal to 1. The height of a tree is the number of edges on the longest path between the root node and the leaf node.



The above tree is a [binary search tree](#). A binary search tree is a tree in which each node on the left side has a lower value than its parent node, and the node on the right side has a higher value than its parent node. In the above tree, n1 is a root node, and n4, n6, n7 are the leaf nodes. The n7 node is the farthest node from the root node. The n4 and n6 contain 2 edges and there exist three edges between the root node and n7 node. Since n7 is the farthest from the root node; therefore, the height of the above tree is 3.

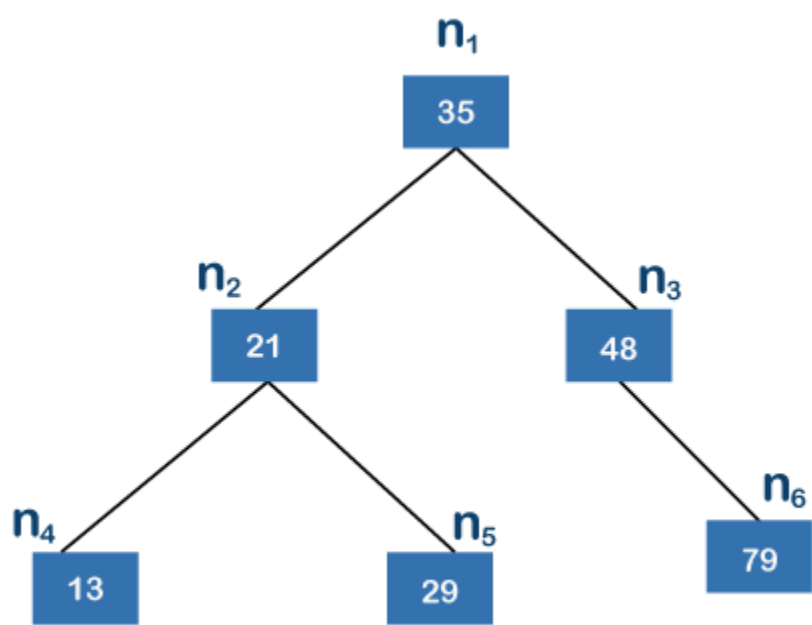
Now we will see whether the above tree is balanced or not. The left subtree contains the nodes n2, n4, n5, and n7, while the right subtree contains the nodes n3 and n6. The left subtree has two leaf nodes, i.e., n4 and n7. There is only one edge between the node n2 and n4 and two edges between the nodes n7 and n2; therefore, node n7 is the farthest from the root node. The height of the left subtree is 2. The right subtree contains only one leaf node, i.e., n6, and has only one edge; therefore, the height of the right subtree is 1. The difference between the heights of the left subtree and right subtree is 1. Since we got the value 1 so we can say that the above tree is a height-balanced tree. This process of calculating the difference between the heights should be performed for each node like n2, n3, n4, n5, n6 and n7. When we process each node, then we will find that the value of k is not more than 1, so we can say that the above tree is a balanced [binary tree](#).



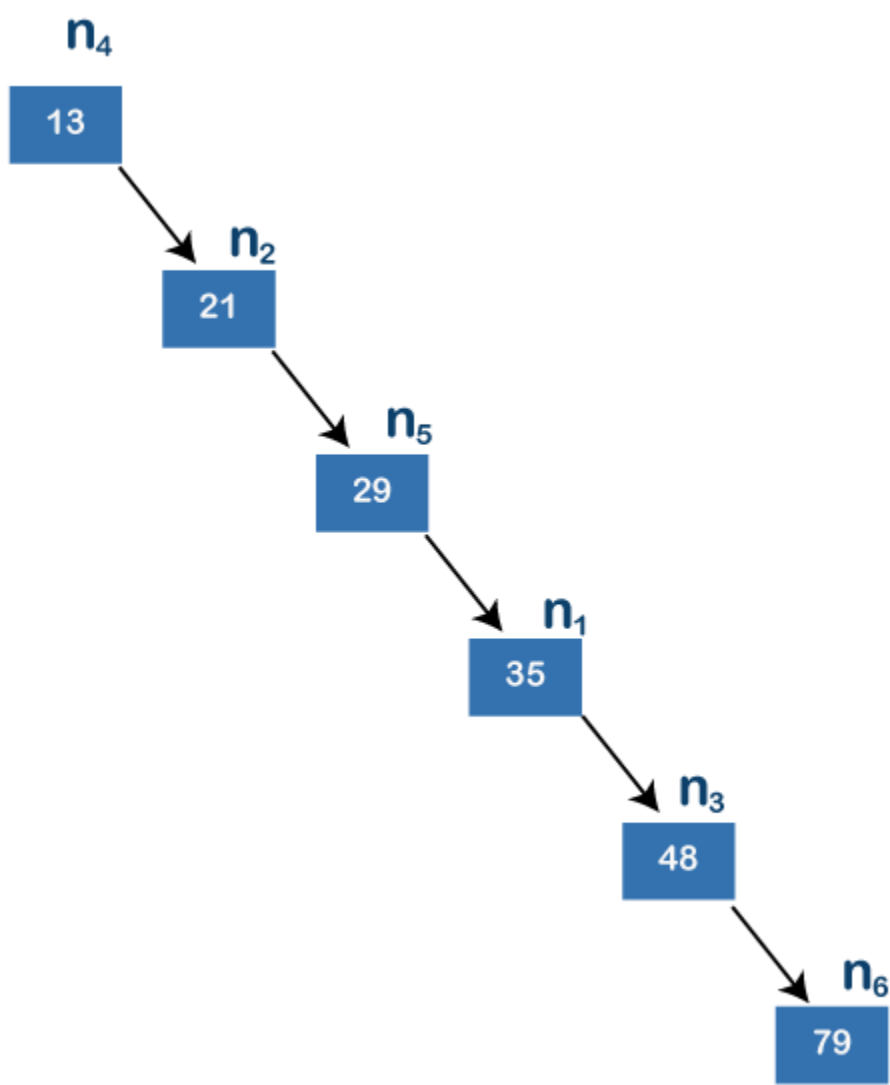
In the above tree, n6, n4, and n3 are the leaf nodes, where n6 is the farthest node from the root node. Three edges exist between the root node and the leaf node; therefore, the height of the above tree is 3. When we consider n1 as the root node, then the left subtree contains the nodes n2, n4, n5, and n6, while subtree contains the node n3. In the left subtree, n2 is a root node, and n4 and n6 are leaf nodes. Among n4 and n6 nodes, n6 is the farthest node from its root node, and n6 has two edges; therefore, the height of the left subtree is 2. The right subtree does have any child on its left and right; therefore, the height of the right subtree is 0. Since the height of the left subtree is 2 and the right subtree is 0, so the difference between the height of the left subtree and right subtree is 2. According to the definition, the difference between the height of left sub tree and the right subtree must not be greater than 1. In this case, the difference comes to be 2, which is greater than 1; therefore, the above binary tree is an unbalanced binary search tree.

Why do we need a balanced binary tree?

Let's understand the need for a balanced binary tree through an example.



The above tree is a binary search tree because all the left subtree nodes are smaller than its parent node and all the right subtree nodes are greater than its parent node. Suppose we want to find the value 79 in the above tree. First, we compare the value of node n1 with 79; since the value of 79 is not equal to 35 and it is greater than 35 so we move to the node n3, i.e., 48. Since the value 79 is not equal to 48 and 79 is greater than 48, so we move to the right child of 48. The value of the right child of node 48 is 79 which is equal to the value to be searched. The number of hops required to search an element 79 is 2 and the maximum number of hops required to search any element is 2. The average case to search an element is $O(\log n)$.



The above tree is also a binary search tree because all the left subtree nodes are smaller than its parent node and all the right subtree nodes are greater than its parent node. Suppose we want to find the value 79 in the above tree. First, we compare the value 79 with a node n4, i.e., 13. Since the value 79 is greater than 13 so we move to the right child of node 13, i.e., n2 (21). The value of the node n2 is 21 which is smaller than 79, so we again move to the right of node 21. The value of right child of node 21 is 29. Since the value 79 is greater than 29 so we move to the right child of node 29. The value of right child of node 29 is 35 which is smaller than 79 so we move to the right child of node 35, i.e., 48. The value 79 is greater than 48, so we move to the right child of node 48. The value of right child node of 48 is 79 which is equal to the value to be searched. In this case, the number of hops required to search an element is 5. In this case, the worst case is $O(n)$.

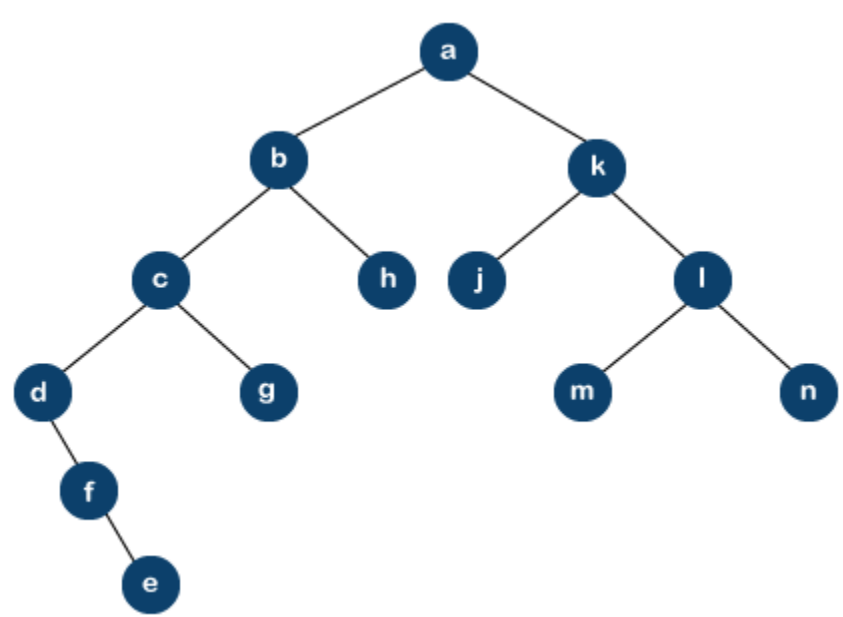
If the number of nodes increases, the formula used in the tree diagram1 is more efficient than the formula used in the tree diagram2. Suppose the number of nodes available in both above trees is 100,000. To search any element in a tree diagram2, the time taken is 100,000μs whereas the time taken to search an element in tree diagram is log(100,000) which is equal 16.6 μs. We can observe the enormous difference in time between above two trees. Therefore, we conclude that the balance binary tree provides searching more faster than linear tree data structure.

Boundary Traversal of Binary tree

The boundary traversal of the binary tree consists of the left boundary, leaves, and right boundary without duplicate nodes as the nodes may contain duplicate values. There are two types of boundary, i.e., left boundary and right boundary. The left boundary can be defined as the path from the root to the left-most node, whereas the right boundary can be defined as the path from the root to the right-most node. If the root node does not contain any left and right subtree, then the root node itself would be considered as the left boundary and right boundary.

Let's understand the boundary traversal of binary tree through an example.

Consider the below tree:



We have to perform a boundary traversal in the above [binary tree](#). First, we traverse all the nodes that appear on the left in the above binary tree that comes under the boundary traversal. The nodes that appear on the left are **a b c d f e**. The nodes that appear on the right are a k l n. We have traversed the left and right nodes, and now we will traverse the leaf nodes. In the above tree, the leaf nodes are e g h j m n. Some of the nodes repeated in all the boundaries; for example, node 'a' appears in both left and right boundary, so we will remove 'a' from the right boundary, and now it appears only once. Some of the nodes are also repeated in leaf nodes. Since node 'e' appears in both left boundary and leaf node so we will remove 'e' from the leaf node. The node 'n' also appears in both right boundary and leaf node, so we will remove the node 'n' from the leaf node. Therefore, the final boundary traversal of tree would be:

a b c d f e k l n g h j

Suppose we want to perform boundary binary tree traversal in an anti-clockwise direction, then the problem is broken down into four parts:

1. First root would be printed.
2. The leftmost nodes except the leaf nodes would be traversed.
3. Leaf nodes
4. The rightmost nodes except the leaf nodes would be traversed.

Source code to find leftmost nodes in a binary tree given below:

```
1. void left_boundary_tree(node *p)
2. {
3.     if(p)
4.     {
5.         if(p->left)
6.         {
7.             print(p->data);
8.             left_boundary_tree(p->left);
9.         }
10. }
```

```

11. else if(p->right)
12. {
13.     print(p->data);
14.     left_boundary_tree(p->right);
15. }
16. }
17. In the above, the pointer p which is passed to the function is the pointer that points to the root node of the binary tree. If the current node has a left node then the condition p->left would be true. It will print the current node data and the pointer p moves to the left of the current node data. The function left_boundary_tree(p->left); calls itself till we find the left node.

```

Source code to find rightmost nodes in a binary tree given below:

```

1. void right_boundary_tree(node *p)
2. {
3.     if(p)
4.     {
5.         if(p->right)
6.         {
7.             print(p->data);
8.             right_boundary_tree(p->right);
9.         }
10.    }
11. else if(p->left)
12. {
13.     print(p->data);
14.     right_boundary_tree(p->left);
15. }
16. }

```

Source code to print the leaf nodes.

```

1. void print_leaf_node(node *p)
2. {
3.     if(p)
4.     {
5.         print_leaf(p->left);
6.         if((!p->left) && !(p->right))
7.         {
8.             print(p->data);
9.         }
10.    print_leaf(p->right);
11. }
12. }

```

Below is the C implementation for the boundary traversal of a binary tree

```

1. /* C program for boundary traversal
2. of a binary tree */
3.
4. #include <stdio.h>
5. #include <stdlib.h>
6.
7. /* Creation of user-defined structure named as node that has data, pointer to left child
8. and a pointer to right child */
9.
10. struct node {
11.     int data;
12.     struct node *left, *right;

```

```

13. };
14.
15. // A function that prints all the leaf nodes of a binary tree
16. void print_Leaf(struct node* root)
17. {
18.     if (root == NULL)
19.         return;
20.
21.     print_Leaf(root->left);
22.
23.     // Condition would be true if the node does not have any left and right child.
24.     if (!(root->left) && !(root->right))
25.         printf("%d ", root->data);
26.
27.     print_Leaf(root->right);
28. }
29.
30. // A function that prints all the left boundary nodes, except a leaf node.
31. // Print the nodes in from the top to the bottom manner
32. void print_left_boundary(struct node* root)
33. {
34.     if (root == NULL)
35.         return;
36.
37.     if (root->left) {
38.
39.         // print the root data before calling the function itself in a top-down manner.
40.
41.         printf("%d ", root->data);
42.         print_left_boundary(root->left);
43.     }
44.     else if (root->right) {
45.         printf("%d ", root->data);
46.         print_left_boundary(root->right);
47.     }
48.
49. // if the node is a leaf node then we do not need to do anything otherwise it will create duplicacy..
50. }
51.
52. // A function that prints all the left boundary nodes, except a leaf node.
53. // Print the nodes in from the bottom to the top manner
54.
55. void print_right_boundary(struct node* root)
56. {
57.     if (root == NULL)
58.         return;
59.
60.     if (root->right) {
61.         // First, we will call the right subtree in order to maintain the bottom to top approach.
62.         print_right_boundary(root->right);
63.         printf("%d ", root->data);
64.     }
65.     else if (root->left) {
66.         print_right_boundary(root->left);
67.         printf("%d ", root->data);
68.     }
69. // if the node is a leaf node then we do not need to do anything otherwise it will create duplicacy..

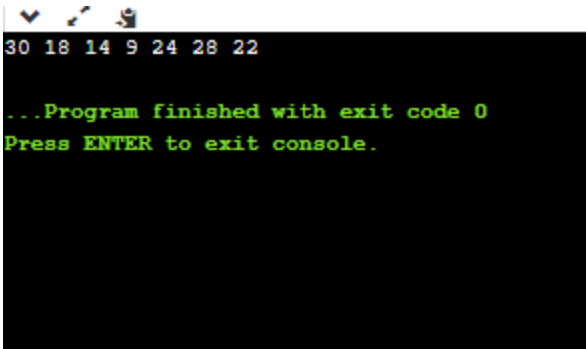
```

```

70.
71.}
72.
73.// A function to do boundary traversal of a given binary tree
74.void print_boundary_nodes(struct node* root)
75.{
76.    if (root == NULL)
77.        return;
78.
79.    printf("%d ", root->data);
80.
81.    // Print the left boundary in top-down manner.
82.    print_left_boundary(root->left);
83.
84.    // Print all leaf nodes
85.    print_Leaf(root->left);
86.    print_Leaf(root->right);
87.
88.    // Print the right boundary in bottom-up manner
89.    print_right_boundary(root->right);
90.}
91.
92.// A function used to create a node
93.struct node* new_Node(int data)
94.{
95.    struct node* temp = (struct node*)malloc(sizeof(struct node));
96.
97.    temp->data = data;
98.    temp->left = temp->right = NULL;
99.
100.        return temp;
101.    }
102.
103.    // main() function from where the execution gets started.
104.    int main()
105.    {
106.        // first we construct a tree by calling the function new_Node()
107.        struct node* root = new_Node(30);
108.        root->left = new_Node(18);
109.        root->left->left = new_Node(14);
110.        root->left->right = new_Node(21);
111.        root->left->right->left = new_Node(9);
112.        root->left->right->right = new_Node(24);
113.        root->right = new_Node(22);
114.        root->right->right = new_Node(28);
115.
116.        print_boundary_nodes(root);
117.
118.        return 0;
119.    }

```

Output



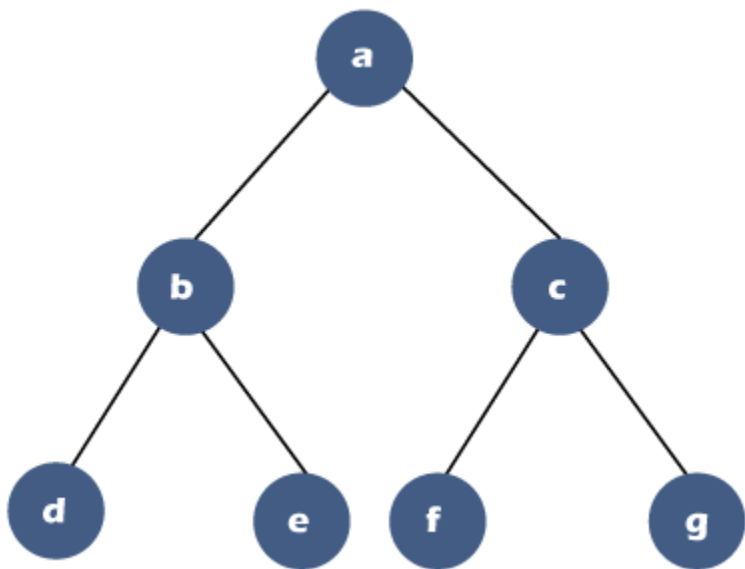
Diagonal Traversal of Binary Tree

Here we will see how we can traverse diagonally in a binary tree. To print the diagonal nodes in a binary tree, we need to calculate the diagonal distance. Let's understand this through an example.

Consider the below tree:

In the above tree, the diagonal distance is represented by using the notation 'd'. There are two rules for marking the diagonal distance:

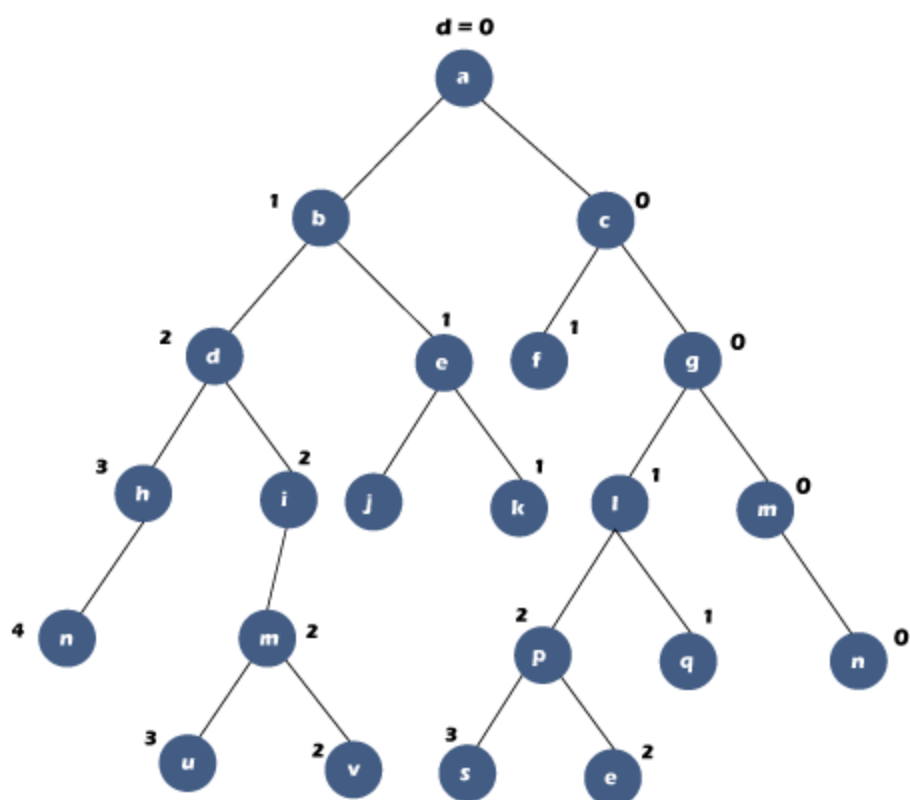
- The 'd' variable increments by 1 only when the node has a left child.
- For every right child, 'd' remains same as of parent ('d' remains the same for right child).



In the above tree, the diagonal distance of node 'a' is 0. Since the node 'a' has two children, i.e., node 'b' (left child) and node 'c' (right child), so the diagonal distance for the node 'b' gets incremented and becomes 1, whereas the diagonal distance for the node 'c' would remain same and becomes 0. The node 'b' has a left child, i.e., d, and its diagonal distance would become 2, while the node 'e' is the right child, so its diagonal distance value would remain the same, i.e., 1. The node 'c' has a left child, i.e., f, so its diagonal distance value gets incremented, and 'd' value of 'f' would become 1. The node 'c' has also the right child, i.e., 'g' and its diagonal distance value would remain the same as its parent, i.e., 'c'.

Once the diagonal distance of all the nodes is calculated, we will find out the diagonals. The nodes that are having the same value of diagonal distance will be considered as a diagonal. In the above tree, we can observe that the nodes 'a', 'c', and 'g' have the same diagonal distance value so "acg" would be considered as a diagonal. The nodes 'b', 'e' and 'f' have the same diagonal distance value, i.e., 1, so "aef" would be considered as a diagonal. Only one node with a diagonal distance value is 2. Therefore, there are three diagonals in the above binary tree: "acg", "bef", and "d".

Consider the below tree to understand the marking of nodes with the diagonal distance more clearly.



In the above **binary tree**, the nodes with a diagonal distance value 0 are 'a', 'c', 'g', 'm' and 'r'. The nodes with a diagonal distance value 1 are 'b', 'e', 'f', 'k', 'l' and 'q'. The nodes with a diagonal distance value 2 are 'd', 'i', 'j', 'm', 'p', 'v' and 't'. The nodes with a diagonal distance value 3 are 'h', 'u', and 's'. There is only one node with a value 4 is 'n'. Therefore, there exist 5 diagonals that are "a c g m r", "b e f k l q", "d i j m p v t", "h u s" and "n".

Short trick to mark the node with a diagonal distance value:

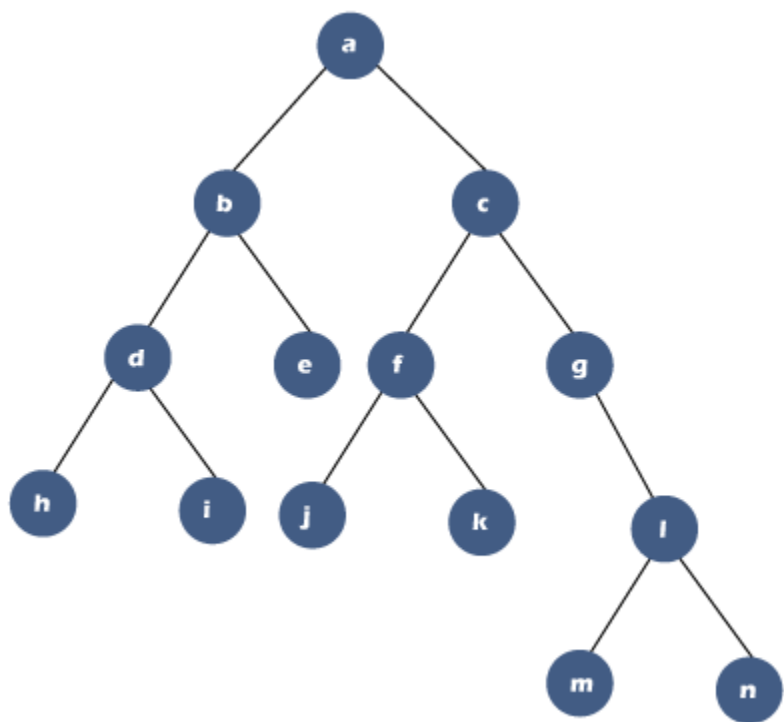
- Firstly, mark the root node as 0. Mark the right-side series of a root node as 0 shown as below:
In simple words, we can say that 0th diagonal is "a c g m r".
- Secondly, the left children of the elements in a 0th diagonal should be marked as 1, i.e., $0+1 = 1$.
- Mark the right-side series as same as we did in the first step. The right-side series of node 'b', i.e., 'e' and 'k' will be marked as 1. The node 'f' does not have any child. The node 'l' has a right-child, i.e., 'q' will be marked with 1.:
Therefore, 1th diagonal would be "b e k l q".
- Now, all the left children of the elements in the 1th diagonal should be marked as 2, i.e., $1+1 = 2$.
- Mark the right-side series as its parent. The right-side series of node 'd' is "i m v" so nodes 'i', 'm' and 'v' are marked as 2. The right-side series of node 'p' is 't'.
Therefore, the 2nd diagonal would be "d i m v j p t".
- Now, we will mark the children of nodes having diagonal distance value as 2. First, we will mark the left-child of nodes. The left-child of node 'd' is 'h', so node 'h' will be marked as 3, i.e., $2+1 = 3$. The left-child of node 'm' is 'u', so node 'u' will be marked as 3, i.e., $2+1 = 3$. The left-child of node 'p' is 's', so node 's' will be marked as 3, i.e., $2+1 = 3$. Once the left-side series is marked, we will mark the right-side series. Since there is no right-child series of nodes so there will be no marking. The final tree would like as:
Therefore, the 3rd diagonal would be "h u s".
- Now we will mark the children of nodes having diagonal distance value as 3. There is only one node 'h' that have the left-child, i.e., 'n'. So, 'n' would be marked as 4. Therefore, the 4th diagonal would be "n".

Algorithm

1. Enqueue(root)
2. Enqueue(NULL)
- While(queue is not empty)
- {
- p = dequeue();
- if(p == NULL)
- {
- Enqueue(NULL);
- p = dequeue();
- if(p == NULL)
- break;
- }

```
13. While (p != NULL)
14. {
15.   Print(p);
16.   if(p->left)
17.     enqueue(p->left);
18.   p = p->right;
19. }
20. }
```

Consider the below tree to understand the above algorithm more clearly.



Here we take Queue data structure for printing the diagonal elements.

According to the above algorithm, the element 'a' is inserted in a Queue and then NULL value shown as below:

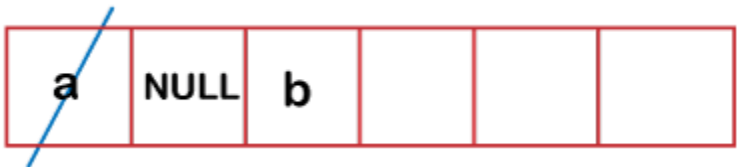


Then, the while loop will execute till the queue is not empty.

In the 0th iteration, 'a' would be dequeued from the Queue. Now, 'p' points to the node 'a' and not equal to NULL, so it will print 'a' shown as below:

output : a

a->left is not NULL so enqueue(a->left) //b; queue would look like:



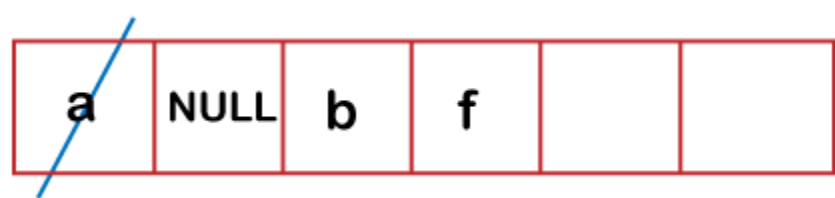
p = a ->right // c

Since 'p' points to the node 'c', so the condition **p != NULL becomes true and will print c.**

output : a, c

In the 1st iteration, 'p' is not equal to NULL; it is equal to 'c'.

c->left is not equal to NULL // 'f'; so 'f' would be enqueued in a queue shown as below:



p = c-> right // 'g'

Since 'p' points to the node 'g', so the condition **p!= NULL** becomes true and will print 'g'.

output : a, c, g

In the **2nd iteration**, 'p' is not NULL; it is equal to 'g'.

g->left is equal to NULL; so, condition p->left is false.

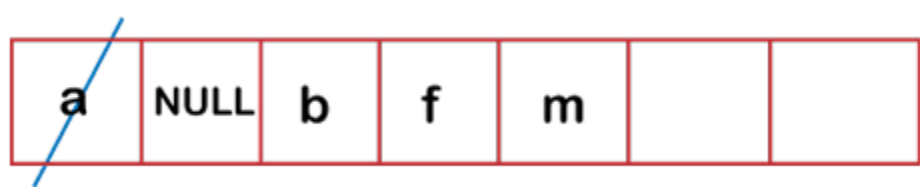
p = g->right // 'l'

Since 'p' points to the node 'l', so the condition **p!= NULL** becomes true and will print 'l'.

output : a, c, g, l

In the **3rd iteration**, 'p' is not NULL; it is equal to 'l'.

p-> left is not equal to NULL // 'm'; so 'm' would be enqueued in a queue shown as below:



p = l->right // 'n'

Since 'p' points to the node 'n', so the condition p != NULL becomes true and will print 'n'

output : a, c, g, l, n

In the **4th iteration**, 'p' is not equal to NULL; it is equal to 'n'.

p->left is equal to NULL; so, the condition p->left is false.

p =p->right; Since node 'n' does not have a right child so 'p' is equal to NULL.

Now the condition p!= NULL is false, so the control comes out of the inner loop. The control moves to the outer loop, where we will check whether the queue is empty or not. Since the queue is not empty so control moves inside the loop, the element would be dequeued from the queue and now 'p' points to the node 'd'.

In the **0th iteration**,

Since the p!= NULL so the condition becomes true of the inner loop. It will print 'b'.

output : a, c, g, l, n, b

p->left is not equal to Null // 'd', so the condition p->left becomes true and it will enqueue 'd' in the queue shown as below:

p = b->right // 'e'

In the **1st iteration**, the condition **p!=NULL** is true as 'p' points to the node 'e'. It will print 'e'.

output : a, c, g, l, n, b, e

p-> left is equal to Null so the condition p->left becomes false.

p = e->right //Since the node 'e' does not have a right child so 'p' contains NULL value.

Now the condition **p!= NULL** is false so the control comes out of the inner loop. The control moves to the outer loop where we will check whether the queue is empty or not. Since the queue is not empty so control inside the loop. Firstly, the element would be dequeued from the queue and now 'p' points to the node 'f'.

In the **0th iteration**,

Since the **p!= NULL** so the condition becomes true of the inner loop. It will print 'f'.

output : a, c, g, l, n, b, e, f

p-> left is not equal to NULL so the condition p-> left becomes true.

p = f->right // 'k'

In the **1st iteration**, the condition **p!= NULL** is true as 'p' points to the node 'k'. It will print 'k'.

output : a, c, g, l, n, b, e, f, k

p-> left is equal to NULL so the condition p-> left becomes false.

p = k-> right // Since the node 'k' does not have a right child so 'p' contains NULL value.

Now the condition **p!= NULL** is false so the control comes out of the inner loop. The control moves to the outer loop where we will check whether the queue is empty or not. Since the queue is not empty so control inside the loop. Firstly, the element would be dequeued from the queue and now 'p' points to the node 'm'.

In the **0th iteration**,

Since the condition 'p!= NULL' is true so the control goes inside the inner loop. It will print 'm'.

output : a, c, g, l, n, b, e, f, k, m

p-> left is equal to NULL so the condition p-> left becomes false.

p = m -> right // Since the node 'm' does not have a right child, so 'p' contains NULL value.

Now the condition **p!= NULL** is false so the control comes out of the inner loop. The control moves to the outer loop where we will check whether the queue is empty or not. Since the queue is not empty so control inside the loop. Firstly, the element would be dequeued from the queue and now 'p' points to the node 'd'.

In the **0th iteration**,

Since the condition p!= NULL is true, so control goes inside the loop. It will print 'd'.

output : a, c, g, l, n, b, e, f, k, m, d

p-> left is not equal to NULL // 'h'; so, condition p->left is true and it will enqueue 'h' in a queue shown as below:

p = d->right; // i;

In the **1st iteration**,

Since the condition **p!= NULL** is true as 'p' points to the node 'i', so the control goes inside the loop. It will print 'i'.

output : a, c, g, l, n, b, e, f, k, m, d, i

p-> left is equal to NULL as node 'i' does not have a left child, so condition p-> left becomes false.

p = i->right; // Since node 'i' does not have a right child; therefore, 'p' points to NULL value.

Now the condition **p!= NULL** is false so the control comes out of the inner loop. The control moves to the outer loop where we will check whether the queue is empty or not. Since the queue is not empty so control inside the loop. Firstly, the element would be dequeued from the queue and now 'p' points to the node 'j'.

In the 0th iteration,

Since the condition **p!= NULL** is true as 'p' points to the node 'j', so control goes inside the loop. It will print 'j'.

output : a, c, g, l, n, b, e, f, k, m, d, i, j

p->left is equal to NULL as node 'j' does not have a left child.

p = j->right; Since the node 'j' does not have a right child so 'p' contains NULL value.

Now the condition **p!= NULL** is false so the control comes out of the inner loop. The control moves to the outer loop where we will check whether the queue is empty or not. Since the queue is not empty so control goes inside the loop. Firstly, the element would be dequeued from the queue and now 'p' points to the node 'h'.

In the 0st iteration,

Since the condition **p!= NULL** is true as 'p' points to the node 'h', so control goes inside the loop. It will print 'h'.

output : a, c, g, l, n, b, e, f, k, m, d, i, j, h

p->left is equal to NULL as node 'h' does not have a left child.

p= h->right; Since the node 'h' does not have a right child, so 'p' contains the NULL value.

Now the condition **p!= NULL** is false so the control comes out of the inner loop. The control moves to the outer loop where we will check whether the queue is empty or not. Since the queue is empty so control comes out of the outer the loop.

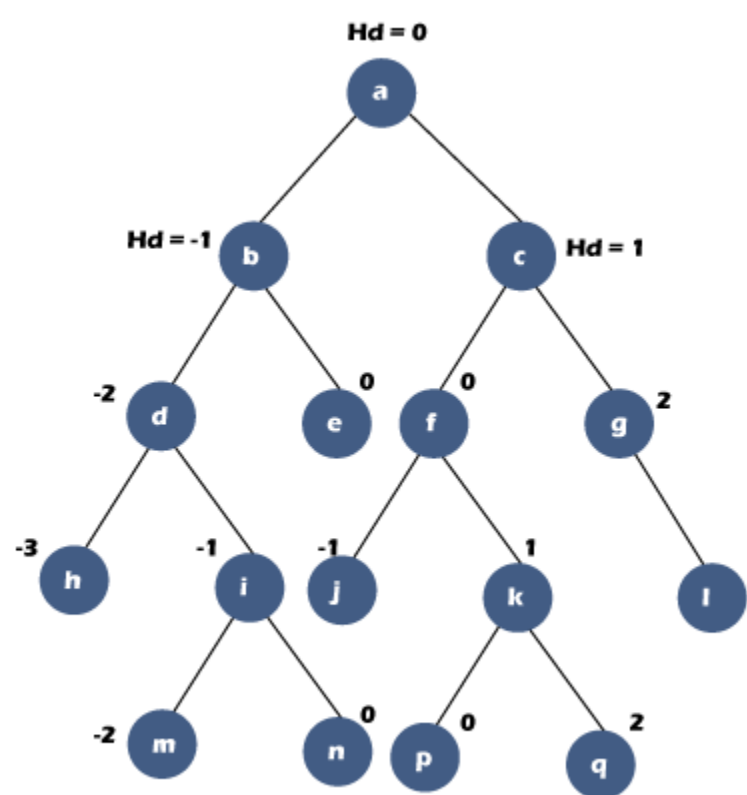
Vertical Traversal of a Binary tree

In this topic, we will see the vertical traversal of a binary tree. For the vertical traversal, we will calculate the horizontal distance. We will assign the horizontal distance to every node, and the horizontal distance could be from any side of the tree. In this case, we will take the left side of the tree from where we will calculate the horizontal distance.

For assigning the horizontal distance to all the node, we will be using the following rules:

- For the root node, the value of H_d is equal to 0, i.e., **$H_d = 0$** .
- For a left child, the value of H_d is equal to the H_d of the parent node minus one, i.e., **$H_d = H_d - 1$** .
- For the right child, the value of H_d is equal to the H_d of the parent node plus one, i.e., **$H_d = H_d + 1$** .

Consider the below tree:



In the above tree, 'a' is the root node, so we will assign the value **H_d** as 0 to the node. Since 'b' is the left child of node 'a', so rule number 2 will be applied to node 'b'. The **H_d** value of node 'b' would be equal to 0 minus 1(0-1), i.e., the **H_d** of node 'b' is equal to -1. The node 'c' is the right child of node 'a', so rule number 3 will be applied to the node 'c'. The **H_d** value of node 'c' would be equal to 0 plus 1 (0+1), i.e., **H_d** of node 'c' is equal to 1. The node 'd' is the left child of node 'b', so the **H_d** value of node 'd' is equal to the **H_d** value of its parent minus one; therefore, **H_d** = -1-1 = -2. The node 'e' is the right child of node 'b', so rule number 3 will be applied to this node. The **H_d** value of 'e' is equal to the H_d value of its parent plus one; therefore, H_d = -1 + 1 = 0. The node 'f' is the left child of node 'c'; therefore, the H_d value of 'f' is equal to (1-1) zero. The node 'g' is the right child of node 'c'; therefore, the H_d value of node 'g' is equal to (1+1) = 2. The node 'h' is a left child of node 'd'; therefore, the H_d value of node 'h' is equal to (-2-1) = -3. The node 'i' is the right child of node 'd'; therefore, the H_d value of node 'i' is equal to -1. The node 'j' is the left child of node 'e'; therefore, the H_d value of node 'j' is equal to -1. The node 'k' is the right child of node 'e'; therefore, the H_d value of node 'k' is equal to 1. The node 'l' is the left child of node 'g'; therefore, the H_d value of node 'l' is equal to (2-1) = 1. The node 'm' is the right child of node 'g'; therefore, the H_d value of node 'm' is equal to (2+1) = 3.

The node with the minimum horizontal distance value is 'h', i.e., -3, and the node with a maximum horizontal distance value is 'm', i.e., 3. The nodes that have the same horizontal distance value would exist in the same vertical line.

Now we will create the vertical lines.

- The node with a value -3 is 'h', so there exists a vertical line that passes through the node 'h' shown as below:
- The node with a value -2 is 'd', so there exists a vertical line that passes through the node 'd' shown as below:
- For **H_d = -1**, there exist three nodes that are 'b', 'i', and 'j', so a vertical line would be created that passes through these three nodes shown as below:
- For **H_d = 0**, there exist two nodes that are 'a', 'e', and 'f', so a vertical line would be created that passes through these three nodes shown as below:
- For **H_d = 1**, there exist three nodes that are 'c', 'k' and 'l', so a vertical line would be created that passes through these three nodes shown as below:
- For **H_d = 2**, there exist a single node, i.e., 'g', so a vertical line passes through the node 'g' shown as below:
- For **H_d = 3**, there exist a single node, i.e., 'm', so a vertical line passes through the node 'm' shown as below:

Algorithm to implement the vertical traversal of a binary tree

This algorithm is a combination of level order traversal and hash table. The following are the steps required for the vertical traversal of a **binary tree**:

Step 1: Enqueue root.

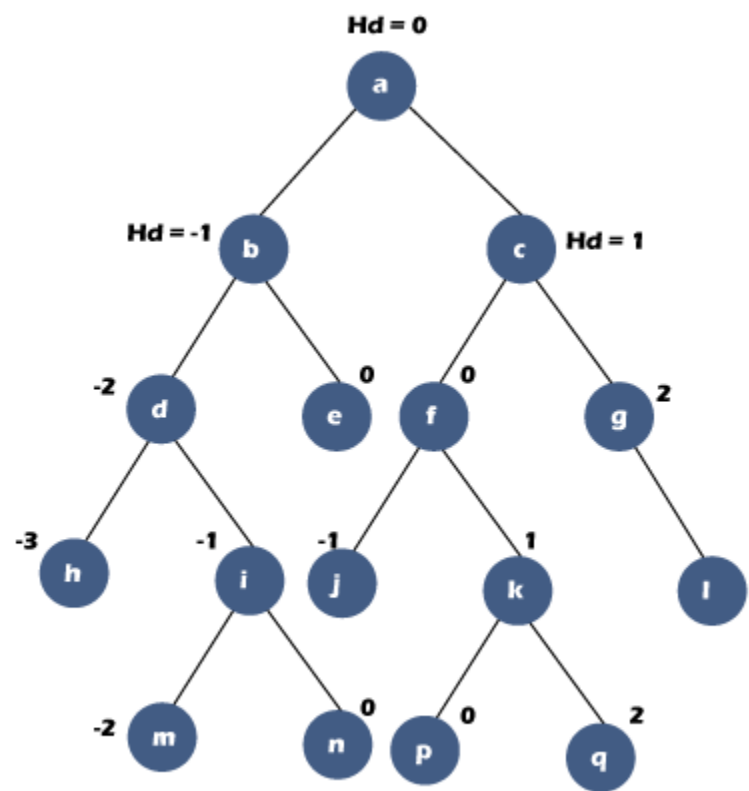
Step 2: Update H_d distance for root as 0.

Step 3: Add H_d as 0 in a hash table and root as the value.

Step 4: First perform Dequeue operation and then perform the following steps:

- Check for the left and right child, and then update **H_d** in a hash table.
- Enqueue the left and right child

Consider the below tree:



We will use two data structures such as Queue, and hash table to implement the vertical traversal,

- We first insert the node 'a' in a queue and update the horizontal distance value of node 'a' as 0. We will also add the H_d of node 'a' and the value in a hash table shown as below:

Queue

a							
----------	--	--	--	--	--	--	--

Hd	Nodes
0	a

According to Step 4 specified in the algorithm, element 'a' is dequeued from the queue and update the hash table with Hd value of left and right child of node 'a' shown as below:

Hd	Nodes
0	a
-1	b
1	c

Once the hash table is updated, we will enqueue the left and right child of node 'a' in a queue shown as below:

a	b	c					
--------------	---	---	--	--	--	--	--

Step 4 is in loop, and it will iterate till the queue does not become empty.

- Check whether the queue is empty or not. The queue is not empty, so dequeue the element 'b' from the queue, and check the left and the right child of 'b' node. Since the node 'b' has both left and right child so we will update the hash table with the Hd value of node 'd' and 'e' shown as below:

a	b	c	d	e			
--------------	--------------	---	---	---	--	--	--

Hd	Nodes
0	a,e
-1	b
1	c
-2	d

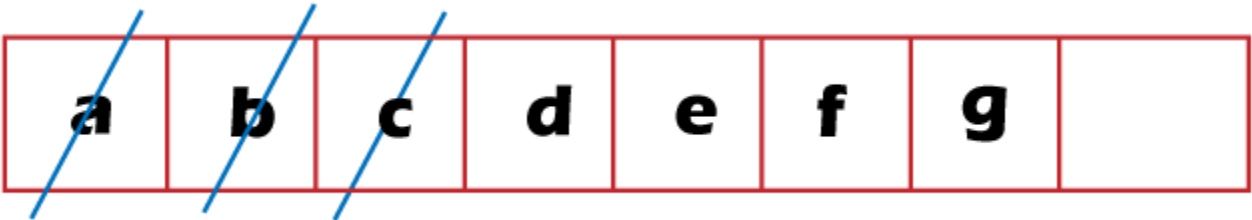
Once the hash table gets updated, enqueue the nodes 'd' and 'e' in a queue shown as below:

- Check whether the queue is empty or not. The queue is not empty so dequeue the element 'c' from the queue, and check the left and right child of 'c' node. Since the node 'c' has both left and right child so we will update the hash table with Hd values of node 'f' and 'g' shown as below:

--	--

Hd	Nodes
0	a,e
-1	b
1	c
-2	d
2	g

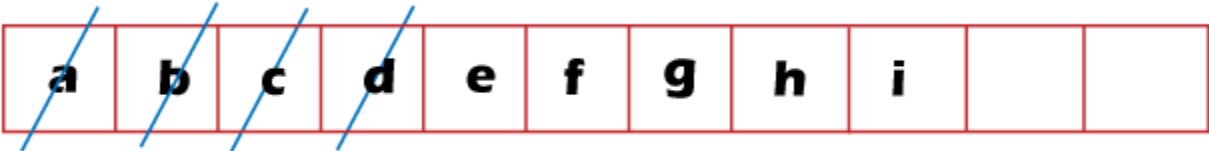
Once the hash table gets updated, enqueue the nodes 'f' and 'g' in a queue shown as below:



- Check again whether the queue is empty or not. The queue is not empty so dequeue the element 'd' from the queue, and check the left and right child of node 'd'. Since the node 'd' has both left and right child so we will update the hash table with Hd values of 'h' and 'i' shown as below:

Hd	Nodes
0	a,e,f
-1	b,i
1	c
-2	d
2	g
-3	h

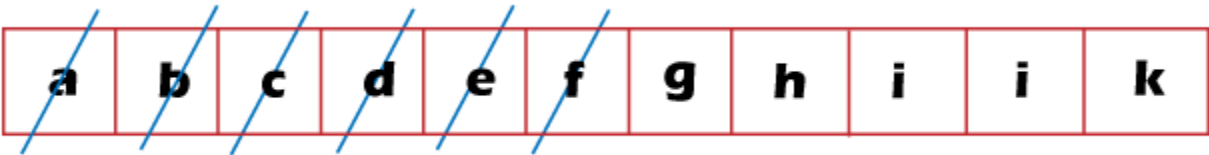
Once the hash table gets updated, enqueue the nodes 'h' and 'i' in a queue shown as below:



- Check again whether the queue is empty or not. The queue is not empty so dequeue the element 'e' from the queue, and check the left and right child of node 'e'. Since node 'e' does not have any left and right child so there will be no updation in the hash table:
- We will check again whether the queue is empty or not. The queue is not empty so dequeue the element 'f' from the queue, and check the left and right of node 'f'. Since the node 'f' has both left and right child, we will update the hash table with Hd values of 'j' and 'k'.

Hd	Nodes
0	a,e,f
-1	b,i, j
1	c,k
-2	d
2	g
-3	h

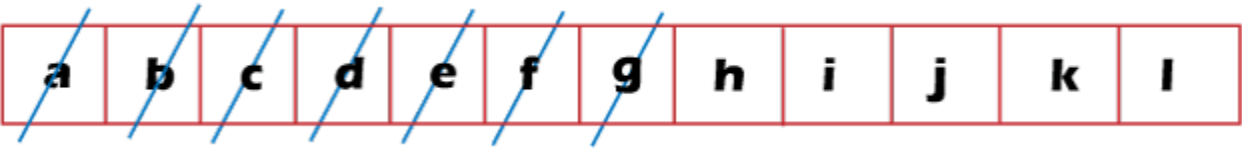
Once the hash table gets updated, enqueue the nodes 'j' and 'k' in a queue shown as below:



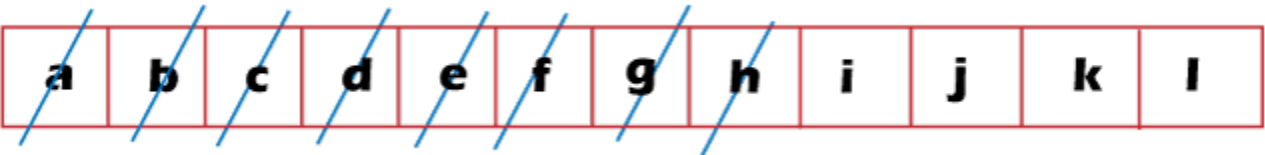
- We will check again whether the queue is empty or not. The queue is not empty so dequeue the element 'g' from the queue, and check the left and right of node 'i'. Since the node 'g' has the only right child, we will update the hash table with Hd values of 'i'.

Hd	Nodes
0	a,e,f
-1	b,i, j
1	c,k
-2	d
2	g
-3	h
3	i

Once the hash table gets updated, enqueue the node 'g' in a queue shown as below:



- Check again whether the queue is empty or not. The queue is not empty so dequeue the element 'h' from the queue, and check the left and right child of node 'h'. Since node 'h' does not have any left and right child so there will be no updation in the hash table:

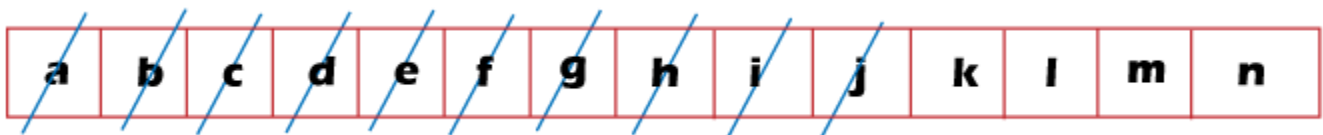


- We will check again whether the queue is empty or not. The queue is not empty so dequeue the element 'i' from the queue, and check the left and right of node 'i'. Since the node 'i' has both left and right child so we will update the hash table with Hd values of 'm' and 'n'.

Once the hash table gets updated, enqueue the nodes 'm' and 'n' in a queue shown as below:



- Check again whether the queue is empty or not. The queue is not empty so dequeue the element 'j' from the queue, and check the left and right child of node 'j'. Since node 'j' does not have any left and right child so there will be no updation in the hash table:



- Check again whether the queue is empty or not. The queue is not empty so dequeue the element 'k' from the queue, and check the left and right child of node 'k'. Since node 'k' has both left and right child so we will update the hash table with Hd values of 'p' and 'q'.

Once the hash table gets updated, enqueue the nodes 'p' and 'q' in a queue shown as below:



- Check again whether the queue is empty or not. The queue is not empty so dequeue the element 'l' from the queue, and check the left and right child of node 'l'. Since node 'l' does not have any left and right child so there will be no updation in the hash table:



- Check again whether the queue is empty or not. The queue is not empty so dequeue the element 'm' from the queue, and check the left and right child of node 'm'. Since node 'm' does not have any left and right child so there will be no updation in the hash table:



- Check again whether the queue is empty or not. The queue is not empty so dequeue the element 'n' from the queue, and check the left and right child of node 'n'. Since node 'n' does not have any left and right child so there will be no updation in the hash table:



- Check again whether the queue is empty or not. The queue is not empty so dequeue the element 'p' from the queue, and check the left and right child of node 'p'. Since node 'p' does not have any left and right child so there will be no updation in the hash table:



- Check again whether the queue is empty or not. The queue is not empty so dequeue the element 'q' from the queue, and check the left and right child of node 'q'. Since node 'q' does not have any left and right child so there will be no updation in the hash table:

Time Complexity of Sorting Algorithms

We might have come across various instances where we need to process the data in a specific format without taking any further delay and the same in case of unsorted data processed with higher speed so that results could be put to some use. In such instances, we use sorting algorithms so that the desired efficiency is achieved. In this article, we will discuss various types of sorting algorithms with higher emphasis on time complexities. But, before moving any further, let's understand what complexity is and what's so important to talk about it.

Complexity

Complexity has no formal definition at all. It just defines the rate of efficiency at which a task is executed. In data structures and algorithms, there are two types of complexities that determine the efficiency of an algorithm. They are:

Space Complexity: Space complexity is the total memory consumed by the program for its execution.

Time Complexity: It is defined as the times in number instruction, in particular, is expected to execute rather than the total time is taken. Since time is a dependent phenomenon, time complexity may vary on some external factors like processor speed, the compiler used, etc.

In computer science, the time complexity of an algorithm is expressed in big O notation. Let's discuss some time complexities.

O(1): This denotes the constant time. $O(1)$ usually means that an algorithm will have constant time regardless of the input size. **Hash Maps** are perfect examples of constant time.

O(log n): This denotes logarithmic time. $O(\log n)$ means to decrease with each instance for the operations. **Binary search trees** are the best examples of logarithmic time.

O(n): This denotes linear time. $O(n)$ means that the performance is directly proportional to the input size. In simple terms, the number of inputs and the time taken to execute those inputs will be proportional or the same. Linear search in **arrays** is the best example of linear time complexity.

O(n²): This denotes quadratic time. $O(n^2)$ means that the performance is directly proportional to the square of the input taken. In simple, the time taken for execution will take square times the input size. **Nested loops** are perfect examples of quadratic time complexity.

Let's move on to the main plan and discuss the time complexities of different sorting algorithms.

Time Complexity of Bubble Sort

Bubble sort is a simple sorting algorithm where the elements are sorted by comparing each pair of elements and switching them if an element doesn't follow the desired order of sorting. This process keeps repeating until the required order of an element is reached.

Average case time complexity: **O(n²)**

Worst-case time complexity: **O(n²)**

Best case time complexity: **O(n)**

The best case is when the given list of elements is already found sorted. This is why bubble sort is not considered good enough when the input size is quite large.

Time Complexity of Selection Sort

Selection sort works on the fundamental of in-place comparison. In this algorithm, we mainly pick up an element and move on to its correct position. This process is carried out as long as all of them are sorted in the desired order.

Average case time complexity: **O(n²)**

Worst-case time complexity: **O(n²)**

Best case time complexity: **O(n²)**

Selection sort also suffers the same disadvantage as we saw in the bubble sort. It is inefficient to sort large data sets. It is usually preferred because of its simplicity and performance-enhancing in situations where auxiliary memory is limited.

Time Complexity of Insertion Sort

Insertion sort works on the phenomenon by taking inputs and placing them in the correct order or location. Thus, it is based on iterating over the existing elements while taking input and placing them where they are ought to be.

Best case time complexity: **$O(n)$**

Average and worst-case time complexity: **$O(n^2)$**

Time Complexity of QuickSort

Quicksort works under the hood of the famous divide and conquer algorithm. In this technique, large input arrays are divided into smaller sub-arrays, and these sub-arrays are recursively sorted and merged into an enormous array after sorting.

Best and Average time complexity: **$O(n \log n)$**

Worst-case time complexity: **(n^2)**

Time Complexity Of Merge Sort

Merge Sort also works under the influence of the divide and conquer algorithm. In this sorting technique, the input array is divided into half, and then these halves are sorted. After sorting, these two halved sub-arrays are merged into one to form a complete sorted array.

Best and Average time complexity: **$O(n \log n)$**

Worst-case time complexity: **$O(n \log n)$**

Conclusion

Time complexity plays a crucial role in determining the overall performance of a program. It is solely intended to improve the performance of a program and impact the overall performance of the system. However, with great speed comes greater responsibility. Hence, to achieve the best time complexity, a developer needs to have a keen eye on using a particular algorithm or technique that delivers the best case complexity. Furthermore, to be at such a pace, a developer needs to carry prior knowledge about the sorting algorithm. Therefore, it is highly recommended to understand each of the techniques discussed in this article in detail and figure out the best one that suits the situation.

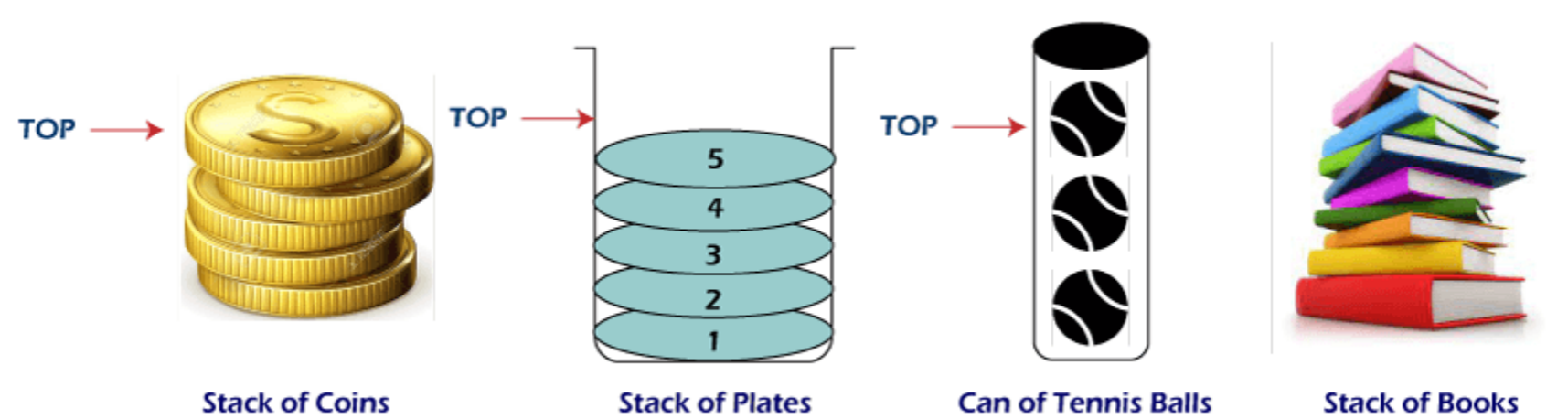
Applications of Stack in Data Structure:

In this article, we will understand the Applications of Stack in the data structure.

What do you mean by Stack?

A Stack is a widely used linear data structure in modern computers in which insertions and deletions of an element can occur only at one end, i.e., top of the Stack. It is used in all those applications in which data must be stored and retrieved in the last.

An everyday analogy of a stack data structure is a stack of books on a desk, Stack of plates, table tennis, Stack of bootless, Undo or Redo mechanism in the Text Editors, etc.



Following is the various Applications of Stack in Data Structure:

- Evaluation of Arithmetic Expressions
- Backtracking
- Delimiter Checking
- Reverse a Data

- Processing Function Calls

1. Evaluation of Arithmetic Expressions

A stack is a very effective **data structure** for evaluating arithmetic expressions in programming languages. An arithmetic expression consists of operands and operators.

In addition to operands and operators, the arithmetic expression may also include parenthesis like "left parenthesis" and "right parenthesis".

Example: $A + (B - C)$

To evaluate the expressions, one needs to be aware of the standard precedence rules for arithmetic expression. The precedence rules for the five basic arithmetic operators are:

Operators	Associativity	Precedence
^ exponentiation	Right to left	Highest followed by *Multiplication and /division
*Multiplication, /division	Left to right	Highest followed by + addition and - subtraction
+ addition, - subtraction	Left to right	lowest

Evaluation of Arithmetic Expression requires two steps:

- First, convert the given expression into special notation.
- Evaluate the expression in this new notation.

Notations for Arithmetic Expression

There are three notations to represent an arithmetic expression:

- Infix Notation
- Prefix Notation
- Postfix Notation

Infix Notation

The infix notation is a convenient way of writing an expression in which each operator is placed between the operands. Infix expressions can be parenthesized or unparenthesized depending upon the problem requirement.

Example: $A + B, (C - D)$ etc.

All these expressions are in infix notation because the operator comes between the operands.

Prefix Notation

The prefix notation places the operator before the operands. This notation was introduced by the Polish mathematician and hence often referred to as polish notation.

Example: $+ A B, -CD$ etc.

All these expressions are in prefix notation because the operator comes before the operands.

Postfix Notation

The postfix notation places the operator after the operands. This notation is just the reverse of Polish notation and also known as Reverse Polish notation.

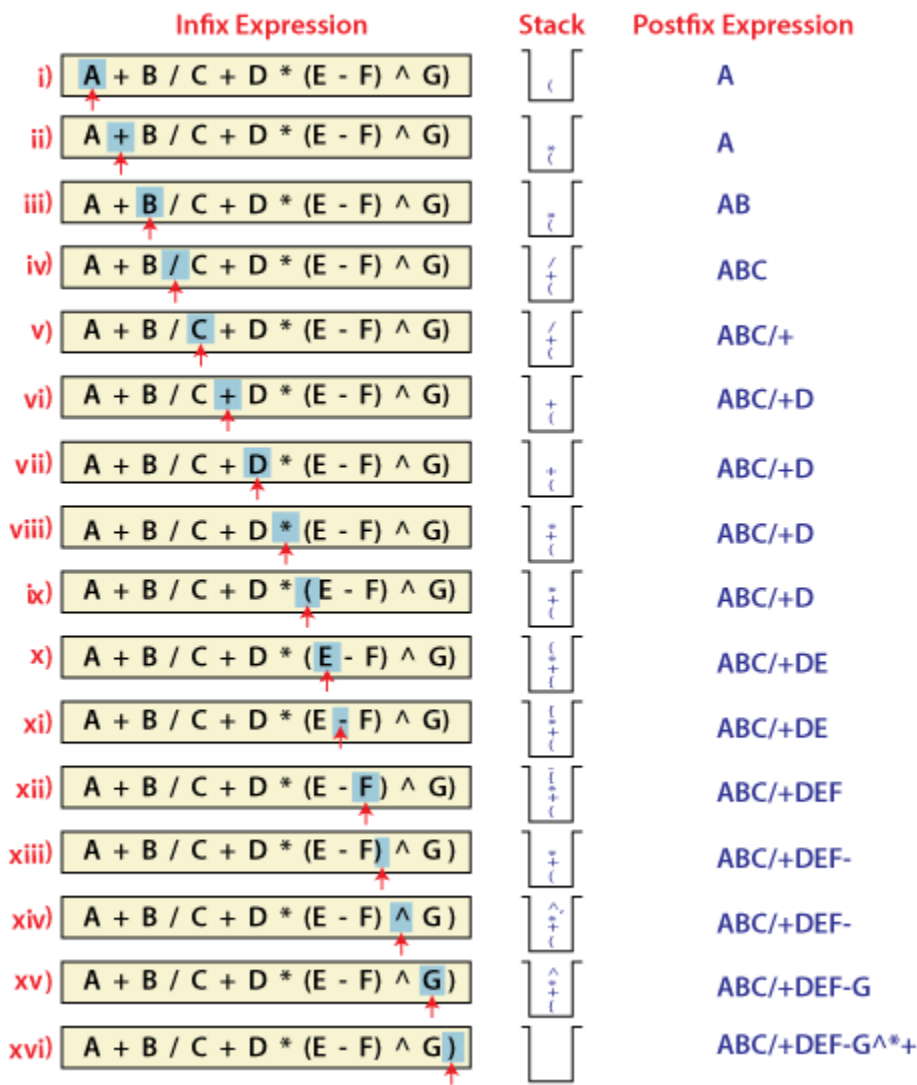
Example: $AB +, CD+,$ etc.

All these expressions are in postfix notation because the operator comes after the operands.

Conversion of Arithmetic Expression into various Notations:

Infix Notation	Prefix Notation	Postfix Notation
A * B	* A B	AB*
(A+B)/C	/+ ABC	AB+C/
(A*B) + (D-C)	+*AB - DC	AB*DC-+

Let's take the example of Converting an infix expression into a postfix expression.



In the above example, the only change from the postfix expression is that the operator is placed before the operands rather than between the operands.

Evaluating Postfix expression:

Stack is the ideal data structure to evaluate the postfix expression because the top element is always the most recent operand. The next element on the Stack is the second most recent operand to be operated on.

Before evaluating the postfix expression, the following conditions must be checked. If any one of the conditions fails, the postfix expression is invalid.

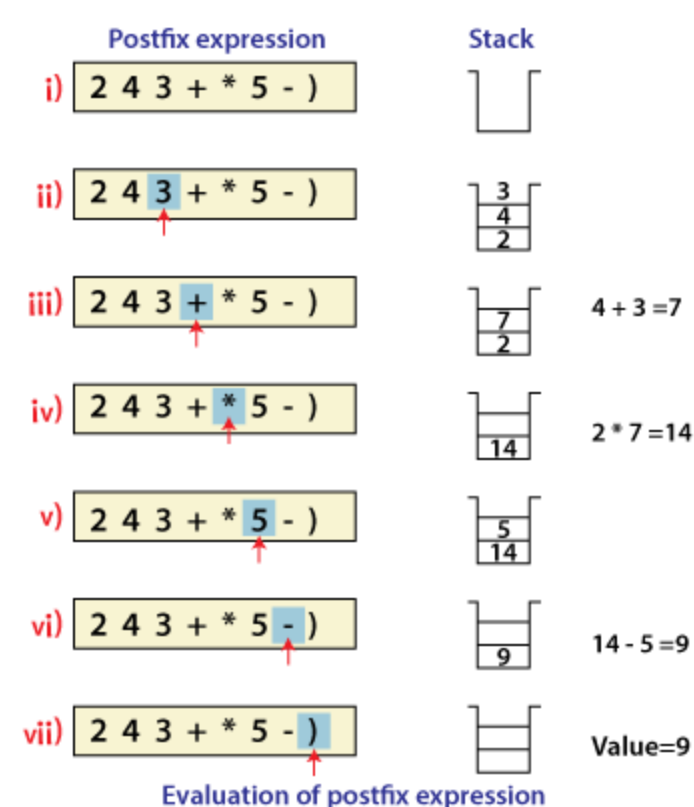
- When an operator encounters the scanning process, the Stack must contain a pair of operands or intermediate results previously calculated.
- When an expression has been completely evaluated, the Stack must contain exactly one value.

Example:

Now let us consider the following infix expression $2 * (4+3) - 5$.

Its equivalent postfix expression is $2 4 3 + * 5$.

The following step illustrates how this postfix expression is evaluated.



2. Backtracking

Backtracking is another application of Stack. It is a recursive algorithm that is used for solving the optimization problem.

3. Delimiter Checking

The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically. It is also called parenthesis checking. When the compiler translates a source program written in some programming language such as C, C++ to a machine language, it parses the program into multiple individual parts such as variable names, keywords, etc. By scanning from left to right. The main problem encountered while translating is the unmatched delimiters. We make use of different types of delimiters include the parenthesis checking (,), curly braces {}, and square brackets [], and common delimiters /* and */. Every opening delimiter must match a closing delimiter, i.e., every opening parenthesis should be followed by a matching closing parenthesis. Also, the delimiter can be nested. The opening delimiter that occurs later in the source program should be closed before those occurring earlier.

Valid Delimiter	Invalid Delimiter
While (i > 0)	While (i >
/* Data Structure */	/* Data Structure
{ (a + b) - c }	{ (a + b) - c

To perform a delimiter checking, the compiler makes use of a stack. When a compiler translates a source program, it reads the characters one at a time, and if it finds an opening delimiter it places it on a stack. When a closing delimiter is found, it pops up the opening delimiter from the top of the Stack and matches it with the closing delimiter.

On matching, the following cases may arise.

- If the delimiters are of the same type, then the match is considered successful, and the process continues.
- If the delimiters are not of the same type, then the syntax error is reported.

When the end of the program is reached, and the Stack is empty, then the processing of the source program stops.

Example: To explain this concept, let's consider the following expression.

[{a -b) * (c -d)}/f]

Input left	Characters Read	Stack Contents
(((a-b) * (c-d))/f]	[[
((a-b) * (c-d))/f]	{	[[
(a-b) * (c-d))/f]	([[(
a-b) * (c-d))/f]	a	[[(
-b) * (c-d))/f]	-	[[(
b) * (c-d))/f]	b	[[(
) * (c-d))/f])	[[
* (c-d))/f]	*	[[
(c-d))/f]	([[(
c-d))/f]	g	[[(
-d))/f]	-	[[(
d))/f]	d	[[(
))/f])	[[
]/f]	}	[
/f]	/	[
f]	f	[
]		

4. Reverse a Data:

To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.

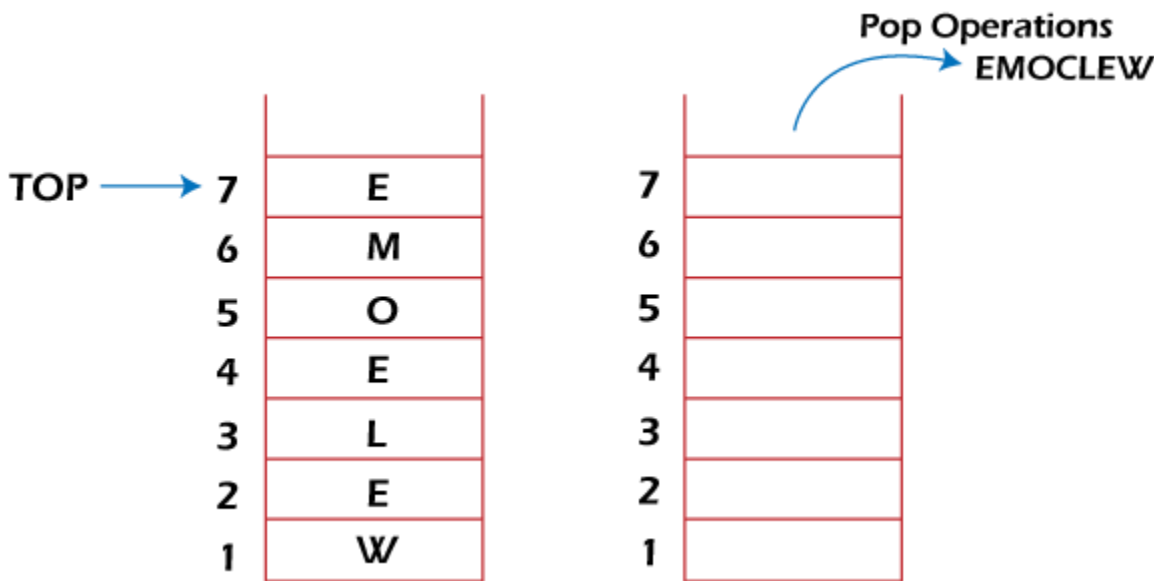
Example: Suppose we have a string Welcome, then on reversing it would be Emoclew.

There are different reversing applications:

- Reversing a string
- Converting Decimal to Binary

Reverse a String

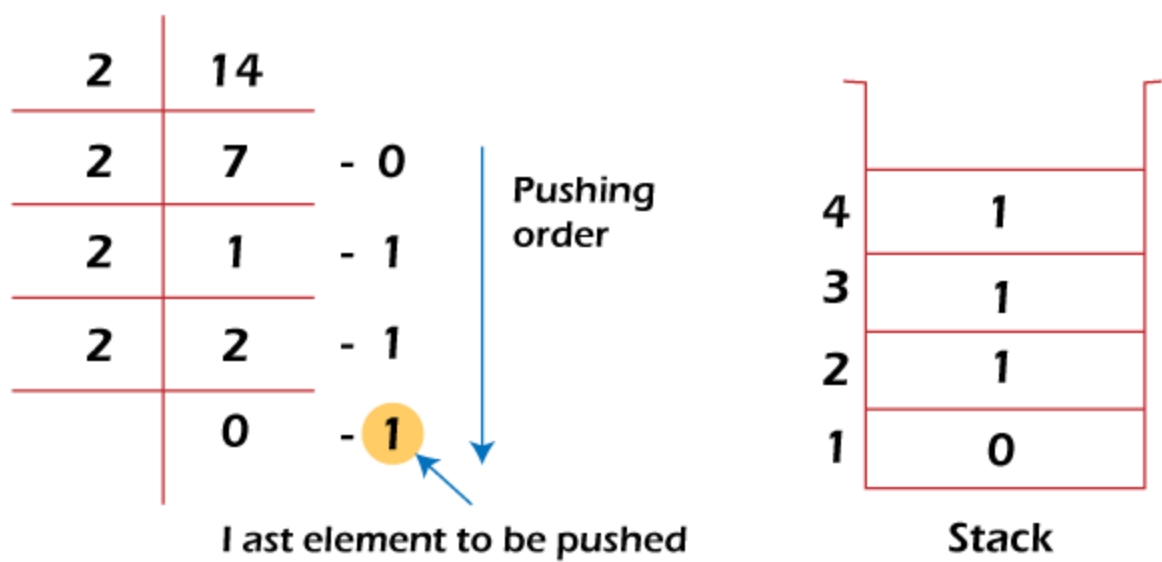
A Stack can be used to reverse the characters of a string. This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one. Because of the **last in first out** property of the Stack, the first character of the Stack is on the bottom of the Stack and the last character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.



Converting Decimal to Binary:

Although decimal numbers are used in most business applications, some scientific and technical applications require numbers in either binary, octal, or hexadecimal. A stack can be used to convert a number from decimal to binary/octal/hexadecimal form. For converting any decimal number to a binary number, we repeatedly divide the decimal number by two and push the remainder of each division onto the Stack until the number is reduced to 0. Then we pop the whole Stack and the result obtained is the binary equivalent of the given decimal number.

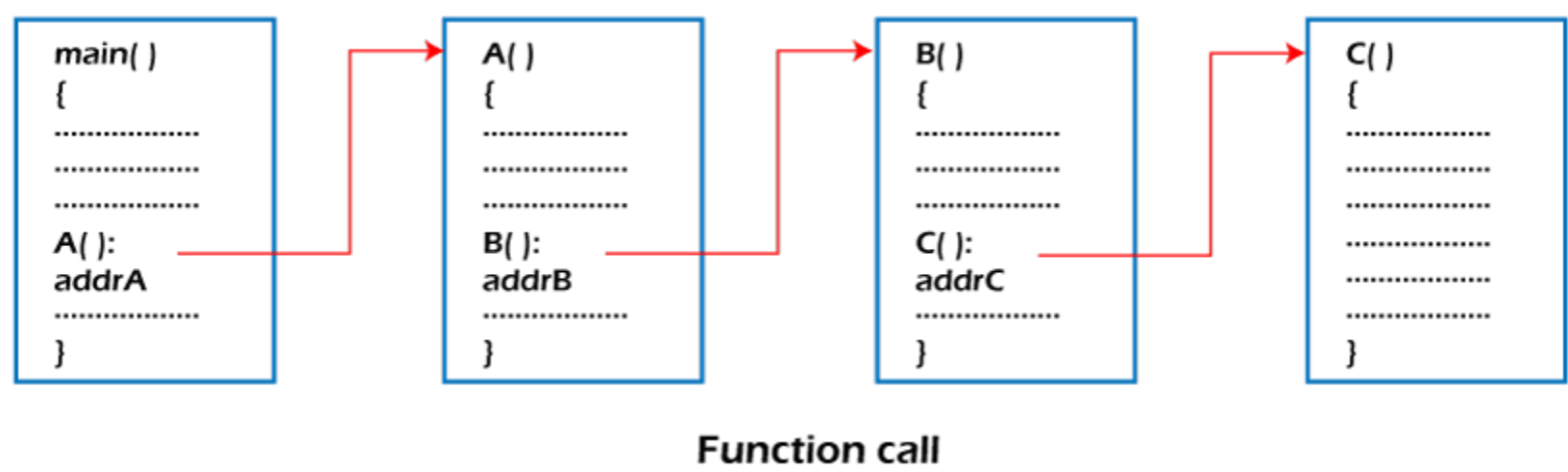
Example: Converting 14 number Decimal to Binary:



In the above example, on dividing 14 by 2, we get seven as a quotient and one as the remainder, which is pushed on the Stack. On again dividing seven by 2, we get three as quotient and 1 as the reminder, which is again pushed onto the Stack. This process continues until the given number is not reduced to 0. When we pop off the Stack completely, we get the equivalent binary number **1110**.

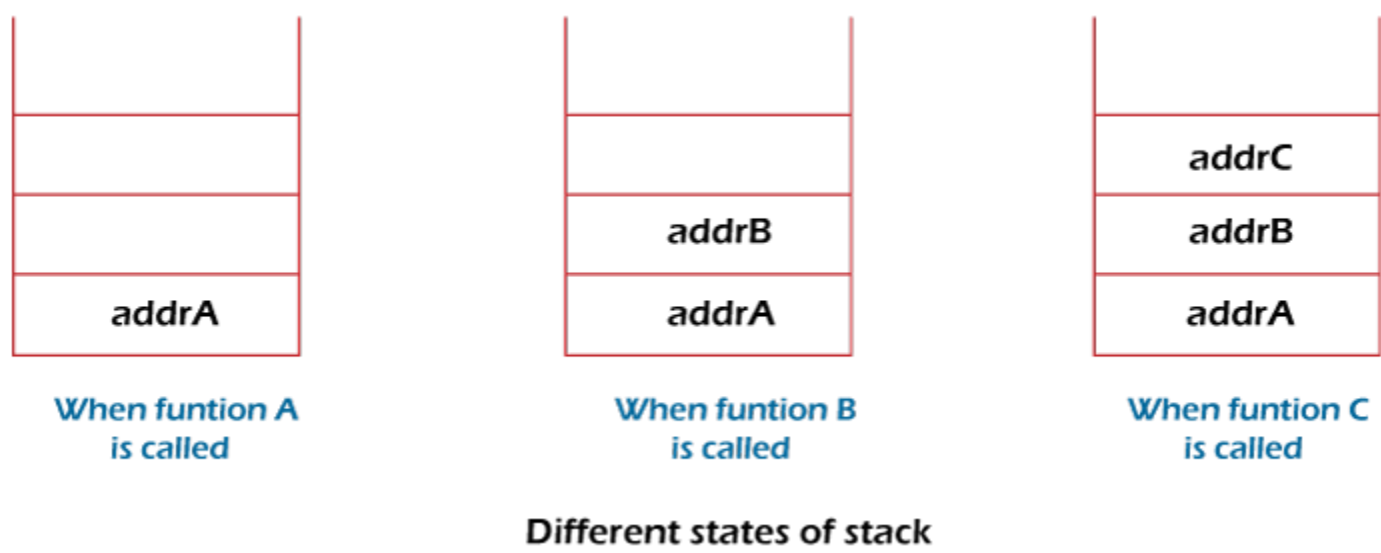
5. Processing Function Calls:

Stack plays an important role in programs that call several functions in succession. Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.



When we invoke function A, which contains a call to function B, then its processing will not be completed until function B has completed its execution and returned. Similarly for function B and C. So we observe that function A will only be completed after function B is completed and function B will only be completed after function C is completed. Therefore, function A is first to be started and last to be completed. To conclude, the above function activity matches the last in first out behavior and can easily be handled using Stack.

Consider addrA, addrB, addrC be the addresses of the statements to which control is returned after completing the function A, B, and C, respectively.



The above figure shows that return addresses appear in the Stack in the reverse order in which the functions were called. After each function is completed, the pop operation is performed, and execution continues at the address removed from the Stack. Thus the program that calls several functions in succession can be handled optimally by the stack data structure. Control returns to each function at a correct place, which is the reverse order of the calling sequence.

Dictionary Data Structure

Dictionary is one of the important Data Structures that is usually used to store data in the key-value format. Each element presents in a dictionary data structure compulsorily have a key and some value is associated with that particular key. In other words, we can also say that Dictionary data structure is used to store the data in key-value pairs. Other names for the Dictionary data structure are associative array, map, symbol table but broadly it is referred to as Dictionary.

A dictionary or associative array is a general-purpose **data structure** that is used for the storage of a group of objects.

Many popular languages add Dictionary or associative array as a primitive data type in their languages while other languages which don't consider Dictionary or associative array as a primitive data type have included Dictionary or associative array in their software libraries. A direct form of hardware-level support for the Dictionary or associative array is **Content-addressable memory**.

In Dictionary or associative array, the relation or association between the key and the value is known as the mapping. We can say that each value in the dictionary is mapped to a particular key present in the dictionary or vice-versa.

The various operations that are performed on a Dictionary or associative array are:

- **Add or Insert:** In the Add or Insert operation, a new pair of keys and values is added in the Dictionary or associative array object.
- **Replace or reassign:** In the Replace or reassign operation, the already existing value that is associated with a key is changed or modified. In other words, a new value is mapped to an already existing key.
- **Delete or remove:** In the Delete or remove operation, the already present element is unmapped from the Dictionary or associative array object.
- **Find or Lookup:** In the Find or Lookup operation, the value associated with a key is searched by passing the key as a search argument.

Now, let us see the usage of the dictionary data structure in different programming languages.

Python:

A sample python code to perform all the basic four operations like create, update, delete and update.

Code:

```
1. # creating a dictionary object named players having key as jersey number and value as the name of the player
2. players=dict()
3.
4.
5. # function to add or insert data into the dictionary object named players
6. def take_input():
7.
8.     print('Enter the jersey number to be entered')
9.     key = int(input())
10.    print('Enter the player name to be entered')
11.    value = input()
12.
13.    players[key]=value
14.
15.
16. # function to update the values of the elements of the
17. object named players
18. def update_values():
19.
20.    print('Enter the key whose value you want to update')
21.    key=int(input())
22.    print('Enter the new value that you want to assign to the key entered')
23.    new_value=input()
24.
25.    players[key]=new_value
```

```
26.
27.
28. # function to delete the elements of the dictionary object named players
29. def delete_elements():
30.
31.     print('Enter the key that you want to delete or remove from the dictionary object')
32.     key=int(input())
33.
34.     players.pop(key)
35.
36.
37.
38.
39. # function to print all the elements of the dictionary object named players
40. def print_dictionary():
41.     print(players)
42.
43.
44.
45.
46.
47. # calling the input fucntion
48. take_input()
49. # calling the printing fucntion
50. print_dictionary()
51. take_input()
52. take_input()
53. take_input()
54. print_dictionary()
55. # calling the updation fucntion
56. print('The dictionary after updating the values is:')
57. update_values()
58. print_dictionary()
59. # calling the deletion fucntion
60. print('The dictionary after deleting the values is:')
61. delete_elements()
62. print_dictionary()
```

Output:

```
Enter the jersey number to be entered
45
Enter the player name to be entered
Rohit Sharma
{45: 'Rohit Sharma'}
Enter the jersey number to be entered
18
Enter the player name to be entered
Virat Kholi
Enter the jersey number to be entered
7
Enter the player name to be entered
Mahendra Singh Dhoni
Enter the jersey number to be entered
42
Enter the player name to be entered
Shikar Dhawan
{45: 'Rohit Sharma', 18: 'Virat Kholi', 7: 'Mahendra Singh Dhoni', 42: 'Shikar Dhawan'}

Enter the key whose value you want to update
42
Enter the new value that you want to assign to the key entered
Shikhar Dhawan
The dictionary after updating the values is:
{45: 'Rohit Sharma', 18: 'Virat Kholi', 7: 'Mahendra Singh Dhoni', 42: 'Shikhar Dhawan'}

Enter the key that you want to delete or remove from the dictionary object
18
The dictionary after deleting the values is:
```

```
{45: 'Rohit Sharma', 7: 'Mahendra Singh Dhoni', 42: 'Shikhar Dhawan'}
```

In this code, we have created a dictionary object named players having key as integer and value as

Strings. We have also created functions to implement all the basic four functionalities of the dictionary that are create, delete, update, etc.

As we can see in the output of the above code, we have 4 elements as input to our dictionary object created and then update one value in the dictionary and then we deleted one value in the dictionary and in the last we have printed the final dictionary object.

Java

A sample java code to perform all the basic operations like create, delete and update.

Code:

```
1. // java code to implement all the basic fucntionalites[add, remove. print, search] of the Dictionary Data Structure
2.
3.
4. // importing all the required classes for creating a dictionary in Java.
5. import java.util.Scanner;
6. import java.util.Hashtable;
7. import java.util.Dictionary;
8. import java.util.Enumeration;
9.
10.
11. // A class named DictionaryJava is created that will consist of four functions to implement all the utilities of the Dictionary
12. class DictionaryJava{
13.
14.
15. // a Dictionary object by the name dictObject is created with the help of Hashtable class
16. static Dictionary dictObject = new Hashtable();
17.
18. // a Scanner object is created to take input from the user
19. static Scanner sc = new Scanner(System.in);
20.
21.
22.
23. // a function named add_element is created to add the elements in the Dictionary object named dictObject
24. public static void add_element(){
25.
26.     System.out.println("Enter the key for the Dictionary Object");
27.     String key = sc.nextLine();
28.     System.out.println("Enter the value for the Dictionary Object");
29.     String value = sc.nextLine();
30.
31.     dictObject.put(key,value);
32.
33. }
34.
35. // a function named print_dictionary is created to print the elements in the Dictionary object named dictObject
36. public static void print_dictionary(){
37.
38.     System.out.println( dictObject.toString() );
39.
40. }
41.
42.
```

```
43.
44. // a function named remove_element is created to delete or remove the elements from the Dictionary object named dictObject
45. public static void remove_element(){
46.
47.     System.out.println("Enter the key of the element that you want to remove or delete ");
48.     String key = sc.nextLine();
49.
50.     dictObject.remove(key);
51. }
52.
53.
54.
55. // a function named search_element is created to find or search the elements in the Dictionary object named dictObject
56. public static void search_element(){
57.
58.     System.out.println("Enter the key of the element that you want to search or find ");
59.     String key = sc.nextLine();
60.
61.
62.     System.out.println("Value of the key " + key + " is " + dictObject.get(key));
63.
64. }
65.
66.
67. }
68.
69.
70.
71. // a Main class is created to call the utility functionalities from the DictionaryJava class
72. public class Main{
73.
74.
75.     public static void main(String ... nk){
76.
77.
78.         // an object of DictionaryJava is created by the name object
79.         DictionaryJava object = new DictionaryJava();
80.
81.
82.         // add_element function of the DictionaryJava is called to add data in the Dictionary object
83.         object.add_element();
84.         object.add_element();
85.
86.         // print_dictionary function of the DictionaryJava is called to print the data of the Dictionary object
87.         object.print_dictionary();
88.
89.         // remove_element function of the DictionaryJava is called to remove or delete the data of the Dictionary object
90.         object.remove_element();
91.         System.out.println("Dictionary after removing the element(s)");
92.         object.print_dictionary();
93.
94.
95.         // search_element function of the DictionaryJava is called to search or find the data of the Dictionary object
96.         object.search_element();
```

```
97.  
98.  }  
99.}
```

Output:

```
Enter the key for the Dictionary Object  
name  
Enter the value for the Dictionary Object  
Nirnay Khajuria  
Enter the key for the Dictionary Object  
age  
Enter the value for the Dictionary Object  
23  
{age=23, name=Nirnay Khajuria}  
  
Enter the key of the element that you want to remove or delete  
age  
Dictionary after removing the element(s)  
{name=Nirnay Khajuria}  
  
Enter the key of the element that you want to search or find  
name  
The value of the key name is Nirnay Khajuria
```

In this code, we have created a dictionary object named **dictObject** having key as string and value as

Strings. We have also created functions to implement all the basic four functionalities of the dictionary that are the create, delete, update, search, etc.

As we can see in the output of the above code, we have two elements as input to our dictionary object created and then update one value in the dictionary and then we searched one value in the dictionary and in the last we have printed the final dictionary object that we have created.

CPP:

Now let us write a C++ code that will give us an idea about how to use Dictionary or associative array and their basic functionalities in C++.

Code:

```
1. // c++ code to implement all the basic fucntionalites[add, remove. print, search] of the Dictionary Data Structure  
2.  
3.  
4. // iostream library is included for basic input output operations  
5. #include <iostream>  
6. // map library is included to use map in our c++ code  
7. #include <map>  
8. // string header is also included in the code to make use of the string objects in the c++ code  
9. #include <string>  
10.  
11. using namespace std;  
12.  
13.  
14. // a map object is created that will stores strings indexed by strings ( that means both the key and the value will be  
    of the string type.)  
15. // NOTE: In C++ we need to explicitly specify the data type of the key and values to be stored in the map or dictio  
    nary object  
16. std::map<std::string, std::string> capitals;  
17.  
18.  
19.  
20. // a fuction named insert_elements is created to add elements into the map or dictionary object named capitals  
21. void insert_elements(){  
22.  
23.     std::string key;  
24.     std::string value;
```

```

25.
26. std::cout<<"\nEnter the name of the country : ";
27. std::cin>>key;
28. std::cout<<"Enter the capital of "<<key<<" : ";
29. std::cin>>value;
30.
31.
32. // value is mapped to the key and inserted successfully to the map or dictionary object named capitals
33. capitals[key]=value;
34.
35.}
36.
37.
38.
39.// a fucntion named print_elements is created to all the elements present in the capitals dictionary object
40.void print_elements(){
41.
42.
43. // each element of the capitals object of the dictionary is iterated and printed
44. for ( auto item : capitals ) {
45.
46.     std::cout <<"Name of the country " << item.first << ": Name of the capital ";
47.     std::cout << item.second << std::endl;
48. }
49.
50.
51.}
52.
53.// a fucntion named delete_elements is created to delete element or elements from the dictionary object named ca
    pitals
54.void delete_elements(){
55.
56.     std :: string key_to_be_deleted;
57.
58.     std :: cout << "\nEnter the name of the country that you want to delete : ";
59.     std :: cin >> key_to_be_deleted;
60.
61.     capitals.erase(key_to_be_deleted);
62.
63.}
64.
65.// a function named search_elements is created to perform search or find operation on the capitals dictionary obje
    ct
66.void search_elements(){
67.
68.     std :: string key_to_be_searched;
69.
70.     std :: cout << "\nEnter the name of the country that you want to search : ";
71.     std :: cin >> key_to_be_searched;
72.
73.     std :: cout << "Capital of " << key_to_be_searched << " is " << capitals[key_to_be_searched]<< "\n";
74.
75.}
76.
77.
78.// a fucntion named update_elements is created to update_elements is created to update or modify the already pr
    esent elements in the dictionary object

```



```

79. void update_elements(){
80.
81.     std :: string key_to_be_updated;
82.     std :: string new_key;
83.
84.     std :: cout << "\nEnter the name of the country whose capital you want to update : ";
85.     std :: cin >> key_to_be_updated;
86.     std :: cout << "Enter the name of new capital : ";
87.     std :: cin >> new_key;
88.
89.     capitals[key_to_be_updated]=new_key;
90.
91. }
92.
93. // main function is written to handle the execution of the code.
94. int main()
95. {
96.
97.     int choice;
98.
99.
100.     // a menu-
    driven program is written to call the various functions that does various operations on the dictionary object named
    capitals
101.
102.     while(1){
103.
104.
105.         std::cout<<"\n1. To insert data into the Dictionary."<<std::endl;
106.         std::cout<<"2. To print data from the Dictionary."<<std::endl;
107.         std::cout<<"3. To delete data from the Dictionary."<<std::endl;
108.         std::cout<<"4. To search data from the Dictionary."<<std::endl;
109.         std::cout<<"5. To update data from the Dictionary."<<std::endl;
110.         std::cout<<"0. To exit the code."<<std::endl;
111.         std::cout<<"Enter your choice:";
112.
113.         std::cin>>choice;
114.
115.
116.         switch(choice){
117.
118.             case 0 :
119.
120.                 // to exit the code.
121.                 exit(0);
122.
123.             case 1 :
124.
125.                 // to insert elements in the dictionary object
126.                 insert_elements();
127.                 break;
128.
129.             case 2 :
130.
131.
132.                 // to print elements in the dictionary object
133.                 std::cout<<std::endl;

```

```

134.         std::cout<<"Contents of the Dictionary are : \n";
135.         print_elements();
136.         break;
137.
138.     case 3 :
139.
140.
141.         // to delete elements in the dictionary object
142.         std::cout<<std::endl;
143.         delete_elements();
144.         std::cout<<"Element deleted sucessfully.\n";
145.         break;
146.
147.     case 4 :
148.
149.
150.         // to search elements in the dictionary object
151.         std::cout<<std::endl;
152.         std::cout<<"Result of the search in the dictionary is : ";
153.         search_elements();
154.         break;
155.
156.     case 5 :
157.
158.
159.         // to update elements in the dictionary object
160.         std::cout<<std::endl;
161.         update_elements();
162.         std::cout<<"Contents of the Dictionary updated sucessfully.\n";
163.         break;
164.
165.     default :
166.
167.         std::cout<<"Please Enter valid input.";
168.
169.     }
170.
171. }
172.
173. return 0;
174. }
175. //end of the main function

```

Output:

```

1. To insert data into the Dictionary.
2. To print data from the Dictionary.
3. To delete data from the Dictionary.
4. To search data from the Dictionary.
5. To update data from the Dictionary.
0. To exit the code.
Enter your choice:1

Enter the name of the country : India
Enter the capital of India : Delhi

1. To insert data into the Dictionary.
2. To print data from the Dictionary.
3. To delete data from the Dictionary.
4. To search data from the Dictionary.
5. To update data from the Dictionary.
0. To exit the code.
Enter your choice:1

Enter the name of the country : Dominica
Enter the capital of Dominica : Roseau

```

```
1. To insert data into the Dictionary.
2. To print data from the Dictionary.
3. To delete data from the Dictionary.
4. To search data from the Dictionary.
5. To update data from the Dictionary.
0. To exit the code.
Enter your choice:1

Enter the name of the country : Haiti
Enter the capital of Haiti : Port-au-prince

1. To insert data into the Dictionary.
2. To print data from the Dictionary.
3. To delete data from the Dictionary.
4. To search data from the Dictionary.
5. To update data from the Dictionary.
0. To exit the code.
Enter your choice:1

Enter the name of the country : USA
Enter the capital of USA : Washington

1. To insert data into the Dictionary.
2. To print data from the Dictionary.
3. To delete data from the Dictionary.
4. To search data from the Dictionary.
5. To update data from the Dictionary.
0. To exit the code.
Enter your choice:2

Contents of the Dictionary are :
Name of the country Dominica: Name of the capital Roseau
Name of the country Haiti: Name of the capital Port-au-prince
Name of the country India: Name of the capital Delhi
Name of the country USA: Name of the capital Washington

1. To insert data into the Dictionary.
2. To print data from the Dictionary.
3. To delete data from the Dictionary.
4. To search data from the Dictionary.
5. To update data from the Dictionary.
0. To exit the code.
Enter your choice:3

Enter the name of the country that you want to delete : Haiti
Element deleted successfully.

1. To insert data into the Dictionary.
2. To print data from the Dictionary.
3. To delete data from the Dictionary.
4. To search data from the Dictionary.
5. To update data from the Dictionary.
0. To exit the code.
Enter your choice:2

Contents of the Dictionary are :
Name of the country Dominica: Name of the capital Roseau
Name of the country India: Name of the capital Delhi
Name of the country USA: Name of the capital Washington

1. To insert data into the Dictionary.
2. To print data from the Dictionary.
3. To delete data from the Dictionary.
4. To search data from the Dictionary.
5. To update data from the Dictionary.
0. To exit the code.
Enter your choice:4

Result of the search in the dictionary is :
Enter the name of the country that you want to search : USA
Capital of USA is Washington

1. To insert data into the Dictionary.
2. To print data from the Dictionary.
3. To delete data from the Dictionary.
4. To search data from the Dictionary.
5. To update data from the Dictionary.
0. To exit the code.
Enter your choice:5

Enter the name of the country whose capital you want to update : India
Enter the name of new capital : New-Delhi
Contents of the Dictionary updated sucessfully.

1. To insert data into the Dictionary.
2. To print data from the Dictionary.
3. To delete data from the Dictionary.
4. To search data from the Dictionary.
5. To update data from the Dictionary.
0. To exit the code.
Enter your choice:2
```

```
Contents of the Dictionary are :
Name of the country Dominica: Name of the capital Roseau
Name of the country India: Name of the capital New-Delhi
Name of the country USA: Name of the capital Washington

1. To insert data into the Dictionary.
2. To print data from the Dictionary.
3. To delete data from the Dictionary.
4. To search data from the Dictionary.
5. To update data from the Dictionary.
0. To exit the code.
Enter your choice:0
```

As we can see in the above code, we have successfully implemented all the basic operations that are search, update, insert, delete and print an object of dictionary named capitals that we have created to store the name of the countries and their respective capitals.

Different functions are created for all of the operations that are mentioned above. First, we created a dictionary and added elements to this created object. Once the insertion of the data is completed in the object, we displayed the added data. After displaying, we deleted already existing data from the dictionary object. After deletion, a searching and updation operations are performed on the dictionary object storing the name of the countries and their respective capitals that are searched and updated various elements from the dictionary object respectively.

So, this article explains the Dictionary Data Structure and what are the basic functions or operations that we can perform on a Dictionary Data Structure object. We also understood the usage of the Dictionary Data Structure in various programming languages like Java, Python, and C++ along with their functionalities that are required to perform the basic operations on this Data Structure. Other than these examples, there are various scenarios where we can use the Dictionary Data Structure. The most ideal scenario for using the Dictionary Data Structure where we need to store our data in key-value pairs.

Structured Data and Unstructured Data

Before understanding the Structured Data and the Unstructured Data, let us know a little bit about the data.

Data can be defined as the information converted into a very economical form for translation or processing. Data, including video, images, sounds, and text, are represented as binary values that mean either 0 or 1. Using these two numbers, patterns are generated to store different types of data. The smallest unit of data in a computer system is a bit, and a single value is represented using a bit. A byte is eight binary digits long.

In the context of today's computers and transmission media, data can be defined as information that is converted into binary digital form. With the increase in the number of computer users, the amount of data generated also get increased drastically within the last decade. So, a new term is coined for such a huge volume of data that is generating at a rapid speed. It is called big data. It is not only the volume of the data that has increased over time. Along with the volume, the variety of the data getting generated is increasing rapidly. So, it becomes very important to classify the types of data that is getting generated. In this era of the internet, a vast amount of data is generated. This data can be either text, images, videos, documents, pdf files, videos, log files, and many more.

Now, let us classify this vast amount of data in broadly two categories. These two categories are:

- Structured Data
- Unstructured Data

Structured Data

We can define Structured Data as the data which has some fixed pattern in them or it is systematic in nature. Structured data is data in which the elements are addressable for efficacious analysis. Structured data is the sort of data that is easily trackable.

The structured data is usually stored in a formatted repository that is typically a database. Most of the time relational databases (RDBMS) are used to store Structured data. All the data that can be stored in a SQL database in a table having some rows and columns depict the structured data. The structured data can always be stored in pre-designed fields, and it also has relational keys. Various data types like ZIP codes, Social Security numbers, or phone numbers are stored in those fields. The records in the table even store the text strings of variable length like names so that they can become easy to search.

The data generated can be either generated by humans or machines. As most of the structured data is stored in Relational databases, it becomes very easy to search the desired data from the stored structured data. In other words, we can say that structured data increases the findability of the data.

Structured data is the information that can be measured easily and can be added into the easy-to-read reports without any further exploitation.

Unstructured Data

Unstructured data can be defined as the data which doesn't exhibit any particular pattern. Unstructured Data is not organized in a predefined manner as Unstructured Data doesn't have any predefined data model and fixed structure, so it is not suitable to store in the mainstream relational database. But there are various alternative options for storing various types of unstructured data. Unstructured Data can be either textual or non-textual data.

Even though unstructured data is not structured in a predefined way, it has a native, internal structure.

Almost 80 to 85 percent of the data that is collected by all the major companies is unstructured data. Unstructured data is very flexible in nature as it doesn't have any schema. The data in the Unstructured data is not bounded or constrained by any kind of fixed schemas. Unstructured data is very much portable and scalable in nature.

Some of the examples of unstructured data are Word, PDF, Text, Media logs, Satellite imagery, Scientific data, Sensor data, Surveillance photos and video, chat, IM, phone recordings, collaboration software, Data from Facebook, Twitter, LinkedIn.

Other than the structured and unstructured data, there is also semi-structured data which is a combination of both structured and unstructured data as it exhibits properties of both the structured and unstructured data.

So, this article helps us to have a better understanding and perspective of structured data and unstructured data.

