

Data Structures

Lecture No. 15

Level-order Traversal of a Binary Tree

In the last lecture, we implemented the tree traversal in preorder, postorder and inorder using recursive and non-recursive methods. We left one thing to explore further that how can we print a binary tree level by level by employing recursive or non-recursive method.

See the figure below:

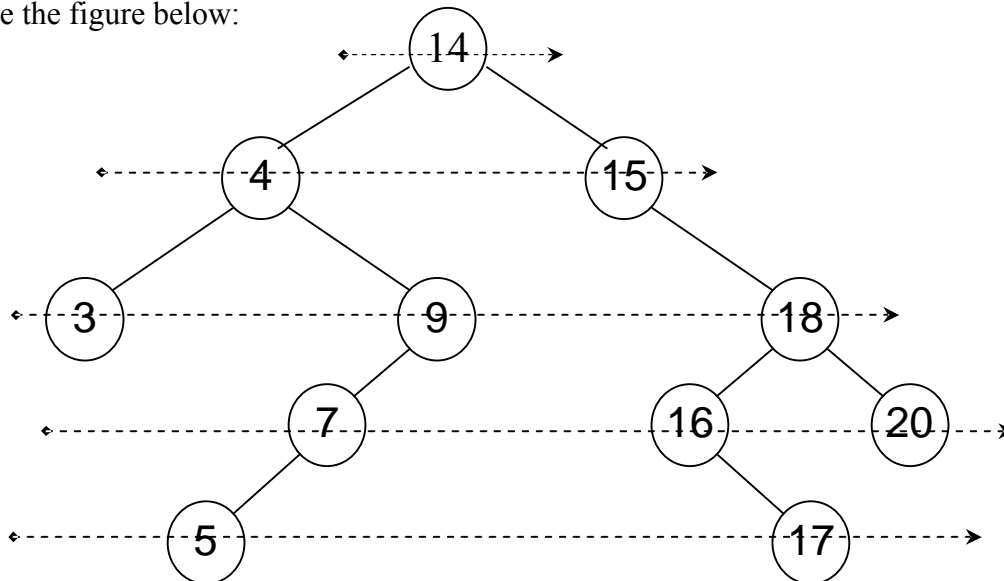


Fig 15.1: Level-order Traversal

In the above figure, levels have been shown using arrows:

At the first level, there is only one number 14.

At second level, the numbers are 4 and 15.

At third level, 3, 9 and 18 numbers are present.

At fourth level the numbers are 7, 16 and 20.

While on fifth and final level of this tree, we have numbers 5 and 17.

This will also be the order of the elements in the output if the level-order traversal is done for this tree. Surprisingly, its implementation is simple using non-recursive method and by employing queue instead of stack. A queue is a FIFO structure, which can make the level-order traversal easier.

The code for the level-order traversal is given below:

```

1. void levelorder( TreeNode <int> * treeNode )
2. {
3.     Queue <TreeNode<int> *> q;
4.
5.     if( treeNode == NULL ) return;
6.     q.enqueue(treeNode);
7.     while( !q.empty() )
8.     {
9.         treeNode = q.dequeue();
10.        cout << *(treeNode->getInfo()) << " ";
11.        if(treeNode->getLeft() != NULL )
12.            q.enqueue( treeNode->getLeft());
13.        if(treeNode->getRight() != NULL )
14.            q.enqueue( treeNode->getRight());
15.    }
16.    cout << endl;
17. }

```

The name of the method is *levelorder* and it is accepting a pointer of type *TreeNode <int>*. This method will start traversing tree from the node being pointed to by this pointer. The first line (line 3) in this method is creating a queue *q* by using the *Queue* class factory interface. This queue will be containing the *TreeNode<int> ** type of objects. Which means the queue will be containing nodes of the tree and within each node the element is of type *int*.

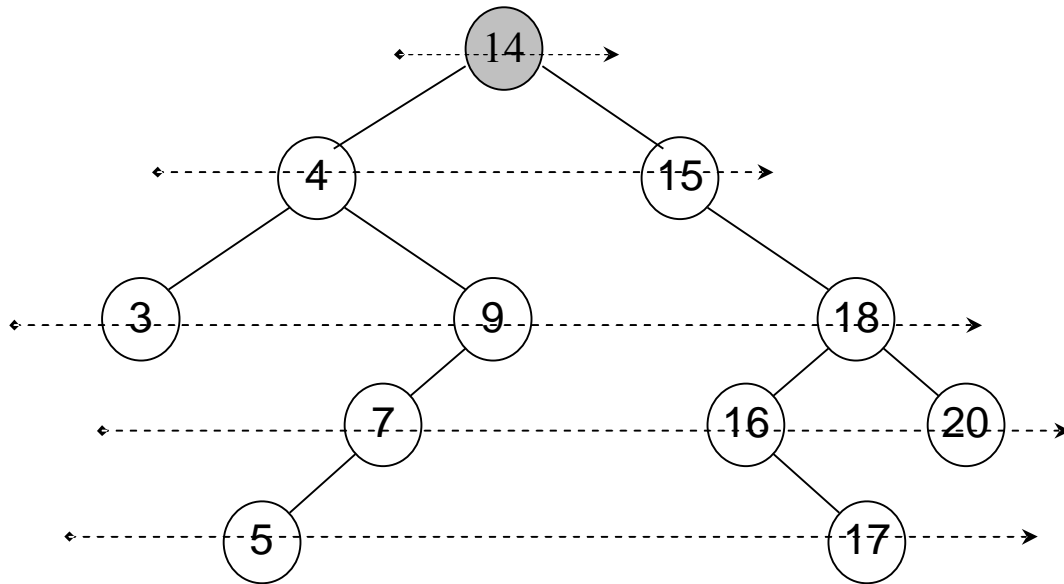
The line 5 is checking to see, if the *treeNode* pointer passed to the function is pointing to NULL. In case it is NULL, there is no node to traverse and the method will return immediately.

Otherwise at line 6, the very first node (the node pointed to by the *treeNode* pointer) is added in the queue *q*.

Next is the *while* loop (at line 7), which runs until the queue *q* does not become empty. As we have recently added one element (at line 6), so this loop is entered.

At line 9, *dequeue()* method is called to remove one node from the queue, the element at *front* is taken out. The return value will be a pointer to *TreeNode*. In the next line (line 10), the *int* value inside this node is taken out and printed. At line 11, we check to see if the left subtree of the tree node (we've taken out in at line 9) is present. In case the left subtree exists, it is inserted into the queue in the next statement at line 12. Next, we see if the right subtree of the node is there, it is inserted into the queue also. Next statement at line 15 closes the *while* loop. The control goes back to the line 7, where it checks to see if there is some element left in the queue. If it is not empty, the loop is entered again until it becomes empty.

Let's execute this method *levelorder(TreeNode <int> *)* by hand on a tree shown in the *fig 15.1* to understand the level order traversal better. We can see that the *root* node of the tree is containing element 14. This node is inserted in the queue first, it is shown in the below figure in gray shade.

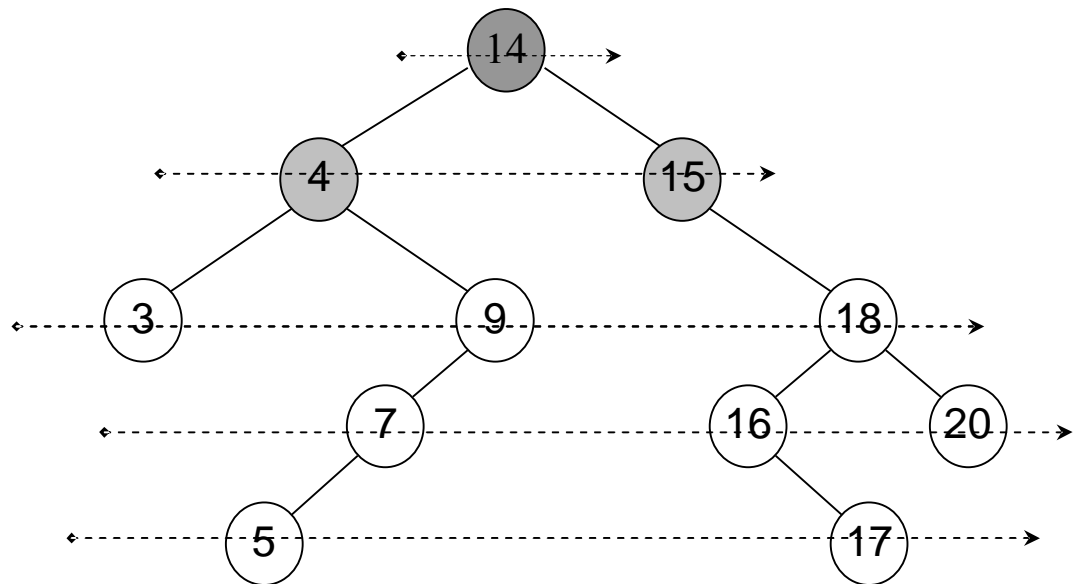


Queue: 14

Output:

Fig 15.2: First node is inserted in the queue

After insertion of node containing number 14, we reach the *while* loop statement, where this is taken out from the queue and element 14 is printed. Next the left and right subtrees are inserted in the queue. The following figure represents the current stage of the tree.

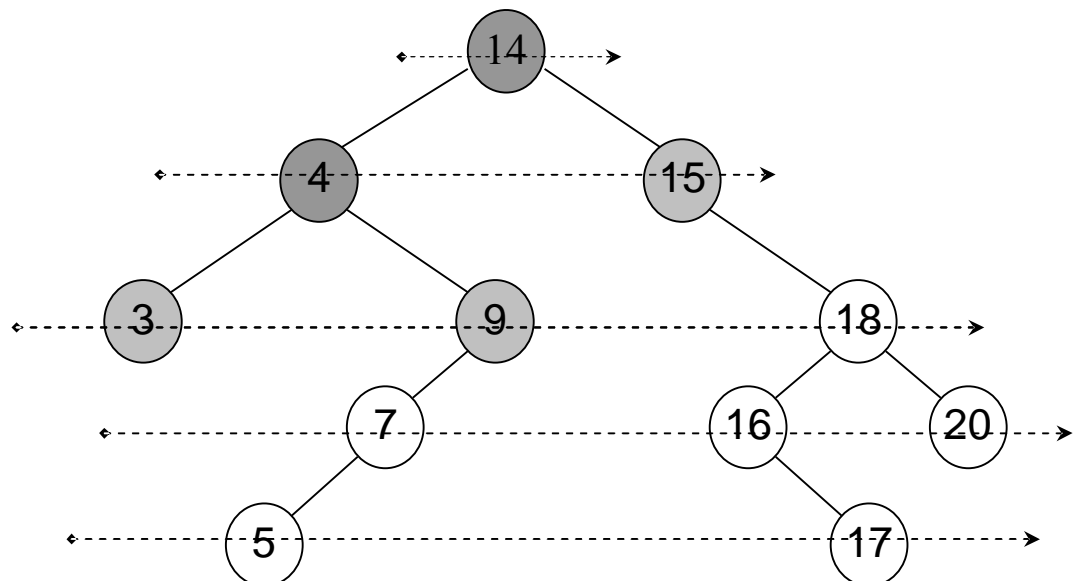


Queue: 4 15

Output: 14

Fig 15.3: 14 is printed and two more elements are added in the queue

The element that has been printed on the output is shown with dark gray shade while the elements that have been inserted are shown in the gray shade. After this the control is transferred again to the start of the *while* loop. This time, the node containing number 14 is taken out (because it was inserted first and then the right node) and printed. Further, the left and right nodes (3 and 9) are inserted in the queue. The following figure depicts the latest picture of the queue.



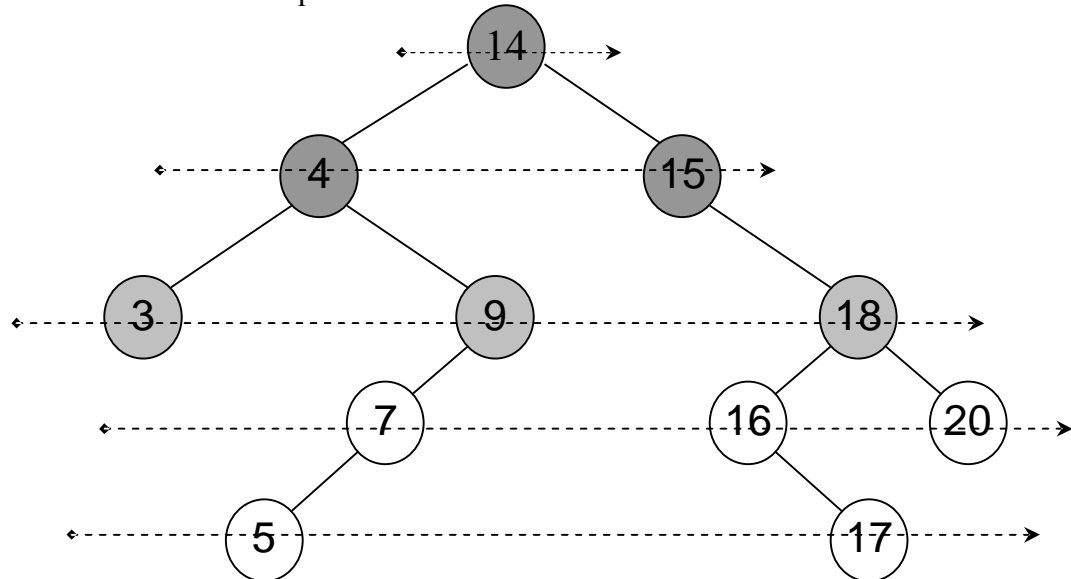
Queue: 15 3 9

Output: 14 4

Fig 15.4

In the above *Queue*, numbers 15, 3 and 9 have been inserted in the queue until now.

We came back to the *while* loop starting point again and dequeued a node from the queue. This time a node containing number 15 came out. Firstly, the number 15 is printed and then we checked for its left and right child. There is no left child of the node (with number 15) but the right child is there. Right child is a node containing number 18, so this number is inserted in the queue.

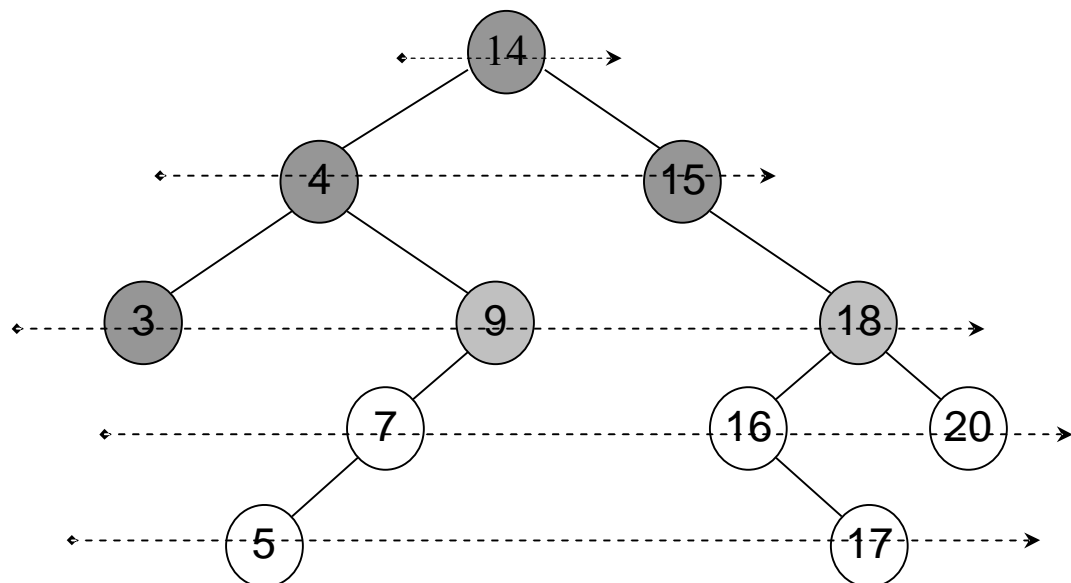


Queue: 3 9 18

Output: 14 4 15

Fig 15.5

This time the node that is taken out is containing number 3. It is printed on the output. Now, node containing number 3 does not have any *left* and *right* children, therefore, no new nodes is inserted in the queue in this iteration.

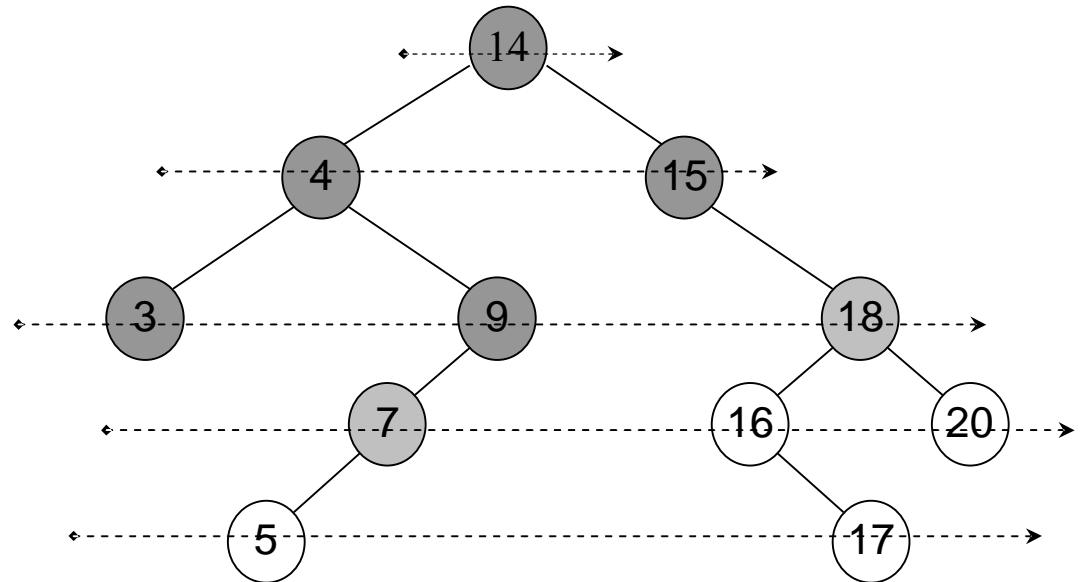


Queue: 9 18

Output: 14 4 15 3

Fig 15.6

In the next iteration, we take out the node containing number 9. Print it on the output. Look for its left subtree, it is there, hence inserted in the queue. There is no right subtree of 9.

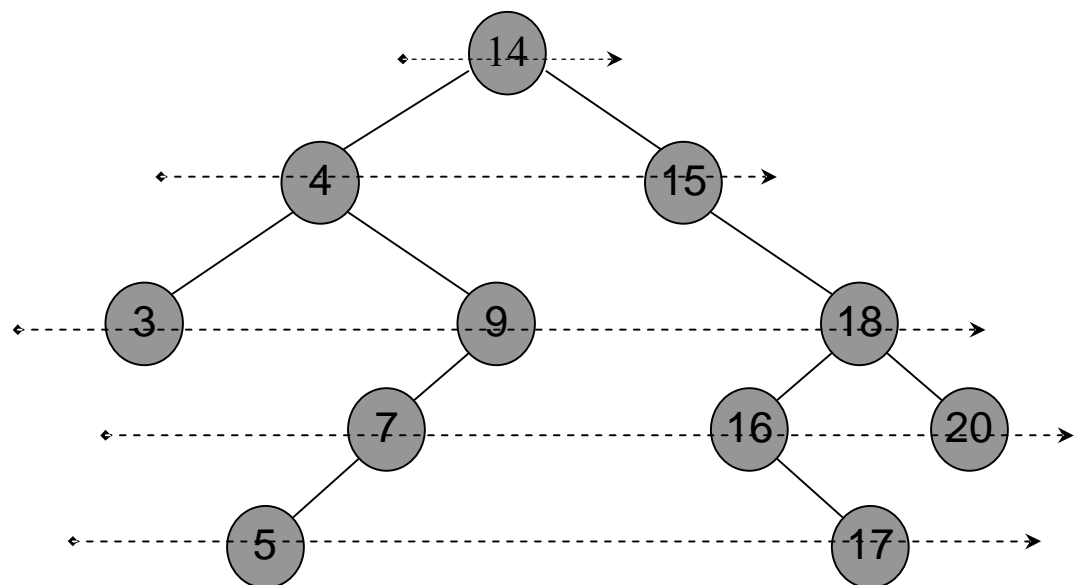


Queue: 18 7

Output: 14 4 15 3 9

Fig. 15.7

Similarly, if we keep on executing this function. The nodes are inserted in the queue and printed. At the end, we have the following picture of the tree:



Queue:

Output: 14 4 15 3 9 18 7 16 20 5 17

Fig 15.8

As shown in the figure above, the *Queue* has gone empty and in the output we can see that we have printed the tree node elements in the level order.

In the above algorithm, we have used *queue* data structure. We selected to use queue data structure after we analyzed the problem and sought most suitable data structure. The selection of appropriate data structure is done before we think about the programming constructs of *if* and *while* loops. The selection of appropriate data structure is very critical for a successful solution. For example, without using the queue data structure our problem of *levelorder* tree traversal would not have been so easier.

What is the reason of choosing queue data structure for this problem. We see that for this levelorder (level by level) tree traversal, the levels come turn by turn. So this turn by turn guides us of thinking of using queue data structure.

Always remember that we don't start writing code after collecting requirements, before that we select the appropriate data structures to be used as part of the design phase. It is very important that as we are getting along on our course of data structures, you should take care why, how and when these structures are employed.

Storing Other Types of Data in Binary Tree

Until now, we have used to place *int* numbers in the tree nodes. We were using *int* numbers because they were easier to understand for sorting or comparison problems. We can put any data type in tree nodes depending upon the problem where tree is employed. For example, if we want to enter the names of the people in the telephone directory in the binary tree, we build a binary tree of strings.

Binary Search Tree (BST) with Strings

Let's write C++ code to insert non-integer data in a binary search tree.

```
void wordTree()
{
    TreeNode<char> * root = new TreeNode<char>();
    static char * word[] = "babble", "fable", "jacket",
        "backup", "eagle", "daily", "gain", "bandit", "abandon",
        "abash", "accuse", "economy", "adhere", "advise", "cease",
        "debunk", "feeder", "genius", "fetch", "chain", NULL};
    root->setInfo( word[0] );

    for(i = 1; word[i]; i++)
        insert(root, word[i] );
    inorder( root );
    cout << endl;
}
```

The function name is *wordTree()*. In the first line, a *root* tree node is constructed, the

node will be containing *char* type of data. After that is a *static* character array *word*, which is containing words like *babble*, *fable* etc. The last word or string of the array is *NULL*. Next, we are putting the first word (*babble*) of the *word* array in the *root* node. Further is a *for* loop, which keeps on executing and inserting words in the tree using *insert (TreeNode<char> *, char *)* method until there is a word in the *word* array (*word* is not *NULL*). You might have noticed that we worked in the same manner when we were inserting *ints* in the tree. Although, in the *int* array, we used *-1* as the ending number of the array and here for words we are using *NULL*. After inserting the whole array, we use the *inorder()* method to print the tree node elements.

Now, we see the code for the *insert* method.

```
void insert(TreeNode<char> * root, char * info)
{
    TreeNode<char> * node = new TreeNode<char>(info);
    TreeNode<char> *p, *q;
    p = q = root;
    while( strcmp(info, p->getInfo()) != 0 && q != NULL )
    {
        p = q;
        if( strcmp(info, p->getInfo()) < 0 )
            q = p->getLeft();
        else
            q = p->getRight();
    }
    if( strcmp(info, p->getInfo()) == 0 )
    {
        cout << "attempt to insert duplicate: " << * info << endl;
        delete node;
    }
    else if( strcmp(info, p->getInfo()) < 0 )
        p->setLeft( node );
    else
        p->setRight( node );
}
```

The *insert(TreeNode<char> * root, char * info)* is accepting two parameters. First parameter *root* is a pointer to a *TreeNode*, where the node is containing element *char* type. The second parameter *info* is pointer to *char*.

In the first line of the *insert* method, we are creating a new *TreeNode* containing the *char* value passed in the *info* parameter. In the next statement, two pointers *p* and *q* of type *TreeNode<char>* are declared. Further, both pointers *p* and *q* start pointing to the tree node passed in the first parameter *root*.

You must remember while constructing binary tree of numbers, we were incrementing its count instead of inserting the new node again if the same number is present in the node. On the other hand, if the same number was not there but the new number was less than the number in the node, we used to traverse to the left subtree of the node. In case, the new number was greater than the number in the node, we used to seek its right subtree.

Similarly, in case of strings (words) here, we will increment the counter if it is already present, and will seek the left and right subtrees accordingly, if required.

In case of *int*'s we could easily compare them and see which one is greater but what will happen in case of strings. You must remember, every character of a string has an associated ASCII value. We can make a lexicographic order of characters based on their ASCII values. For example, the ASCII value of *B* (66) is greater than *A* (65), therefore, character *B* is greater than character *A*. Similarly if a word is starting with the letter *A*, it is smaller than the words starting from *B* or any other character up to *Z*. This is called lexicographic order.

C++ provides us overloading facility to define the same operators (<, >, <=, >=, == etc) that were used for *ints* to be used for other data types for example strings. We can also write functions instead of these operators if we desire. *strcmp* is similar kind of function, part of the standard C library, which compares two strings.

In the code above inside the *while* loop, the *strcmp* function is used. It is comparing the parameter *info* with the value inside the node pointed to by the pointer *p*. As *info* is the first parameter of *strcmp*, it will return a negative number if *info* is smaller, 0 if both are equal and a positive number if *info* is greater. The *while* loop will be terminated if the same numbers are found. There is another condition, which can cause the termination of loop that pointer *q* is pointing to NULL.

First statement inside the loop is the assignment of pointer *q* to *p*. In the second insider statement, the same comparison is done again by using *strcmp*. If the new word pointed to by *info* is smaller, we seek the left subtree otherwise we go to the right subtree of the node.

Next, we check inside the *if-statement*, if the reason of termination of loop is duplication. If it is, a message is displayed on the output and the newly constructed node (that was to be inserted) is deleted (deallocated).

If the reason of termination is not duplication, which means we have reached to the node where insertion of the new node is made. We check if the *info* is smaller than the word in the current tree node. If this is the case, the newly constructed node is inserted to the left of the current tree node. Otherwise, it is inserted to the right.

This *insert()* method was called from inside the *for* loop in the *wordTree()* method. That loop is terminated when the NULL is reached at the end of the array *word*. At the end, we printed the inserted elements of the tree using the *inorder()* method. Following is the output of *inorder()*:

Output:

```
abandon
abash
accuse
adhere
advise
babble
backup
bandit
cease
chain
```

daily
debunk
eagle
economy
fable
feeder
fetch
gain
genius
jacket

Notice that the words have been printed in the sorted order. Sorting is in increasing order when the tree is traversed in *inorder* manner. This should not come as a surprise if you consider how we built the binary search tree. For a given node, values less than the *info* in the node were all in the left subtree and values greater or equal were in the right. *Inorder* prints the left subtree first, then the node itself and at the end the right subtree. Building a binary search tree and doing an *inorder* traversal leads to a sorting algorithm.

We have found one way of sorting data. We build a binary tree of the data, traverse the tree in *inorder* fashion and have the output sorted in increasing order. Although, this is one way of sorting, it may not be the efficient one. When we will study sorting algorithms, will prove Mathematically that which method is the fastest.

Deleting a Node From BST

Until now, we have been discussing about adding data elements in a binary tree but we may also require to delete some data (nodes) from a binary tree. Consider the case where we used binary tree to implement the telephone directory, when a person leaves a city, its telephone number from the directory is deleted.

It is common with many data structures that the hardest operation is deletion. Once we have found the node to be deleted, we need to consider several possibilities.

For case 1, If the node is a *leaf*, it can be deleted quite easily.

See the tree figure below.

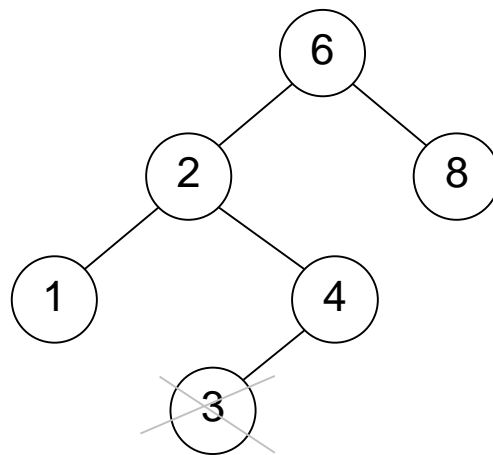


Fig 15.9: BST

Suppose we want to delete the node containing number 3, as it is a leaf node, it is pretty straight forward. We delete the leaf node containing value 3 and point the right subtree pointer to NULL.

For case 2, if the node has one child, the node can be deleted after its parent adjusts a pointer to bypass the node and connect to *inorder* successor.

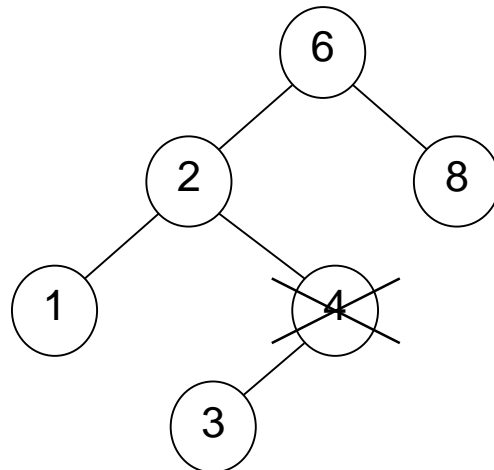


Fig 15.10: Deleting a Node From BST

If we want to delete the node containing number 4 then we have to adjust the right subtree pointer in the node containing value 2 to the *inorder* successor of 4. The important point is that the *inorder* traversal order has to be maintained after the delete.

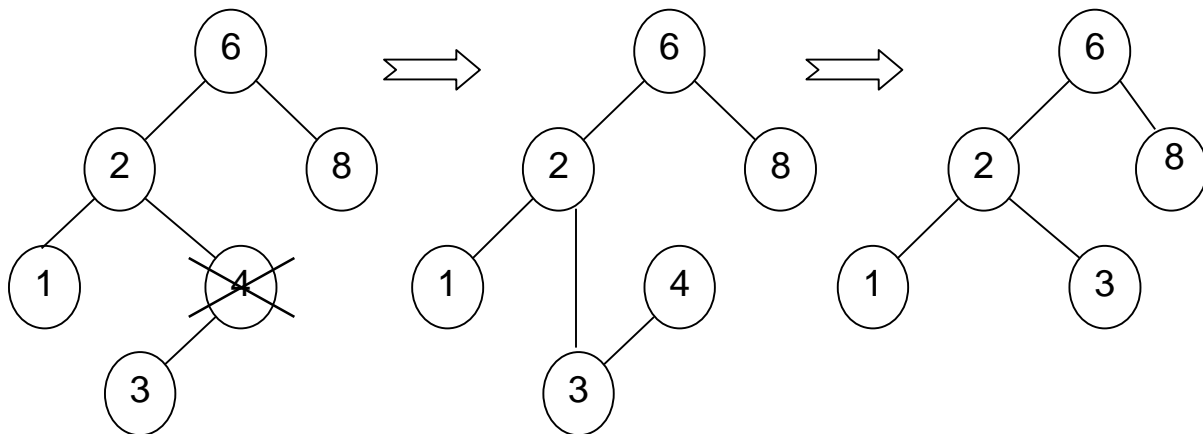


Fig 15.11: Deletion in steps

The case 3 is bit complicated, when the node to be deleted has both left and right subtrees.

The strategy is to replace the data of this node containing the smallest data of the right subtree and recursively delete that node.

Let's see this strategy in action. See the tree below:

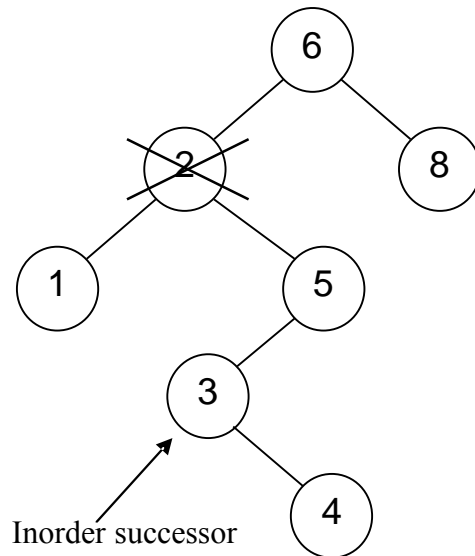


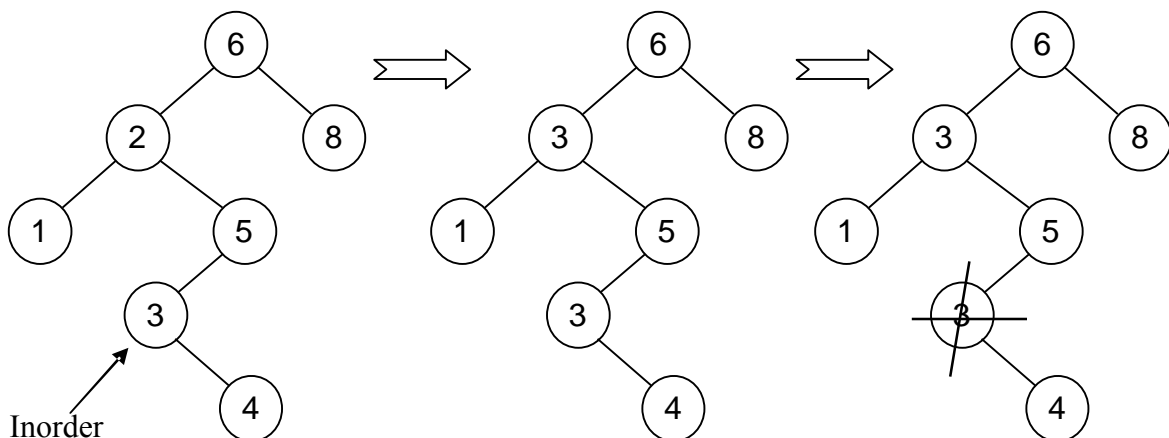
Fig 15.12: delete (2)

In this tree, we want to delete the node containing number 2. Let's do *inorder* traversal of this tree first. The *inorder* traversal give us the numbers: 1, 2, 3, 4, 5, 6 and 8.

In order to delete the node containing number 2, firstly we will see its right subtree and find the left most node of it.

The left most node in the right subtree is the node containing number 3. Pay attention to the nodes of the tree where these numbers are present. You will notice that node containing number 3 is not right child node of the node containing number 2 instead it is left child node of the right child node of number 2. Also the left child pointer of node containing number 3 is NULL.

After we have found the left most node in the right subtree of the node containing number 2, we copy the contents of this left most node i.e. 3 to the node to be deleted with number 2.



Inorder
successor

Fig 15.13: delete (2) - remove the inorder successor

Next step is to delete the left most node containing value 3. Now being the left most node, there will be no left subtree of it, it might have a right subtree. Therefore, the deletion of this node is the case 2 deletion and the delete operation can be called recursively to delete the node.

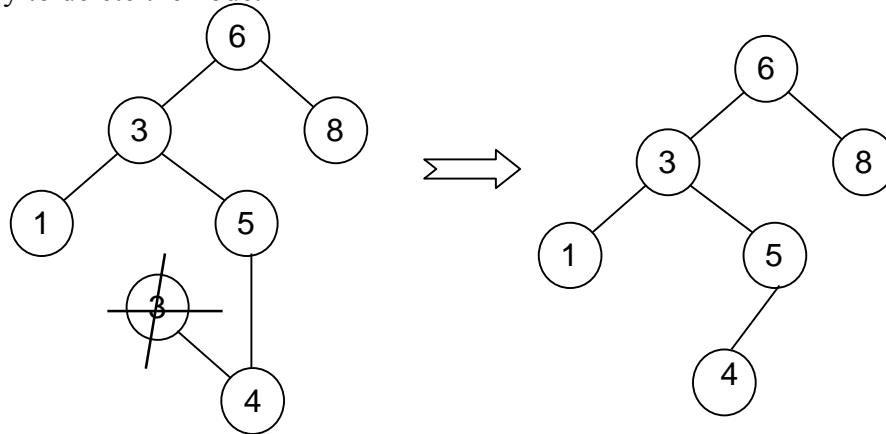


Fig 15.14: delete (2)

Now if we traverse the tree in *inorder*, we get the numbers as: 1, 3, 4, 5, 6 and 8. Notice that these numbers are still sorted. In the next lecture, we will also see the C++ implementation code for this deletion operation.