

The general binary tree shown is not terribly useful in practice. The chief use of binary trees is for providing rapid access to data (indexing, if you will) and the general binary tree does not have good performance.

Suppose that we wish to store data elements that contain a number of fields, and that one of those fields is distinguished as the key upon which searches will be performed.

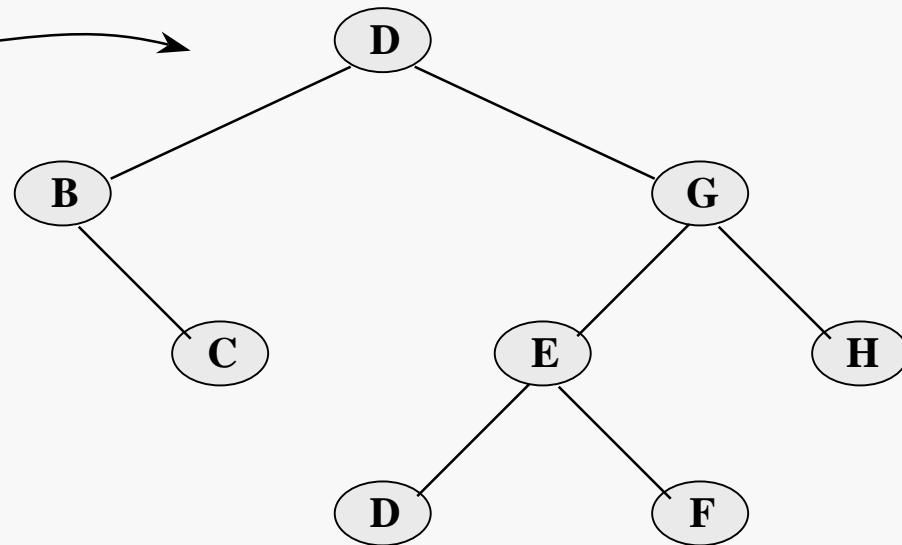
A binary search tree or BST is a binary tree that is either empty or in which the data element of each node has a key and satisfies the conditions:

1. The key of the left child (if there is one) is less than the key of its parent node.
2. The key of the right child (if there is one) is greater than or equal to the key of its parent node.
3. The left and right subtrees of the root are binary search trees.

Here, the key values are characters (and only key values are shown).

Inserting the following key values in the given order yields the given BST:

D G H E B D F C



In a BST, insertion is always at the leaf level. Traverse the BST, comparing the new value to existing ones, until you find the right spot, then add a new leaf node holding that value.

What is the resulting tree if the (same) key values are inserted in the order:

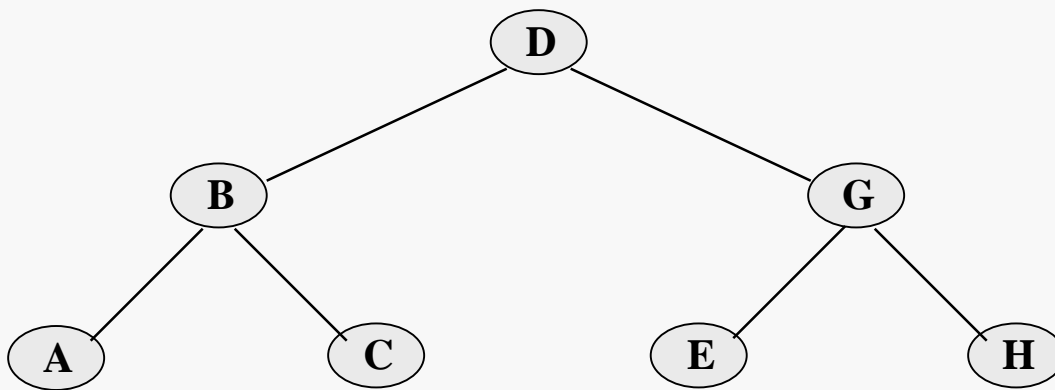
B C D D E F G H

or

E B C D D F G H

Because of the key ordering imposed by a BST, searching resembles the binary search algorithm on a sorted array, which is $O(\log N)$ for an array of N elements.

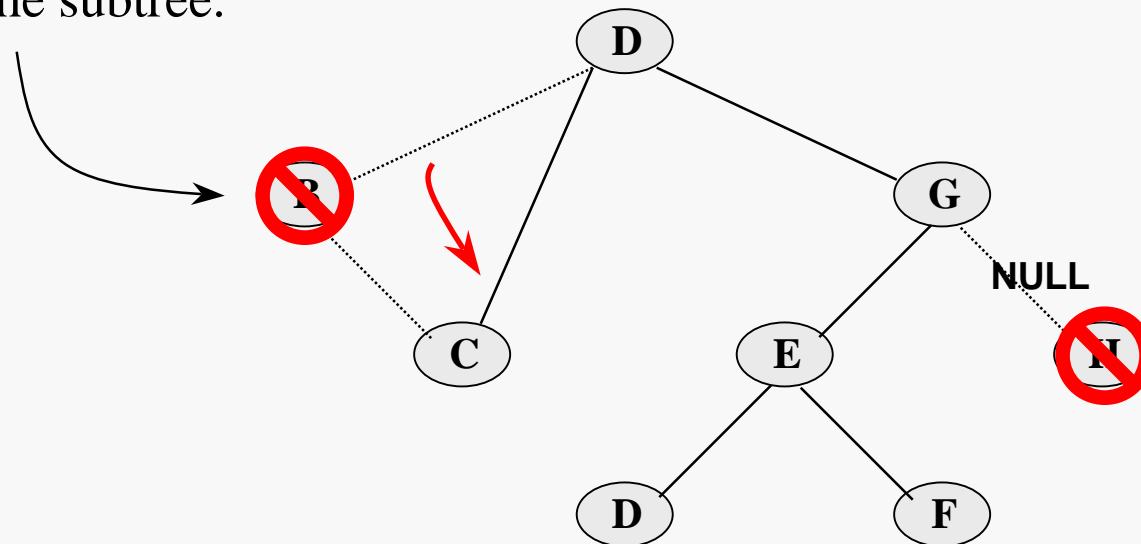
A BST offers the advantage of purely dynamic storage, no wasted array cells and no shifting of the array tail on insertion and deletion.



Trace searching for the key value E.

Deletion is perhaps the most complex operation on a BST, because the algorithm must result in a BST. The question is: what value should replace the one deleted? As with the general tree, we have cases:

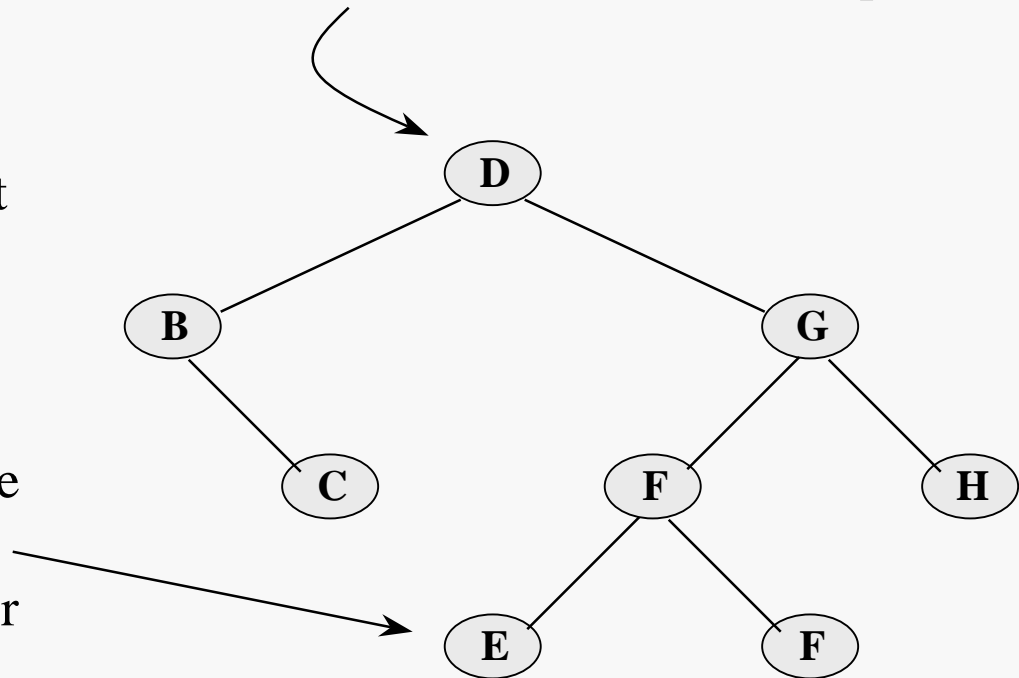
- Removing a leaf node is trivial, just set the relevant child pointer in the parent node to NULL.
- Removing an internal node which has only one subtree is also trivial, just set the relevant child pointer in the parent node to target the root of the subtree.



- Removing an internal node which has two subtrees is more complex...

Simply removing the node would disconnect the tree. But what value should replace the one in the targeted node?

To preserve the BST property, we must take the smallest value from the right subtree, which would be the closest successor of the value being deleted.

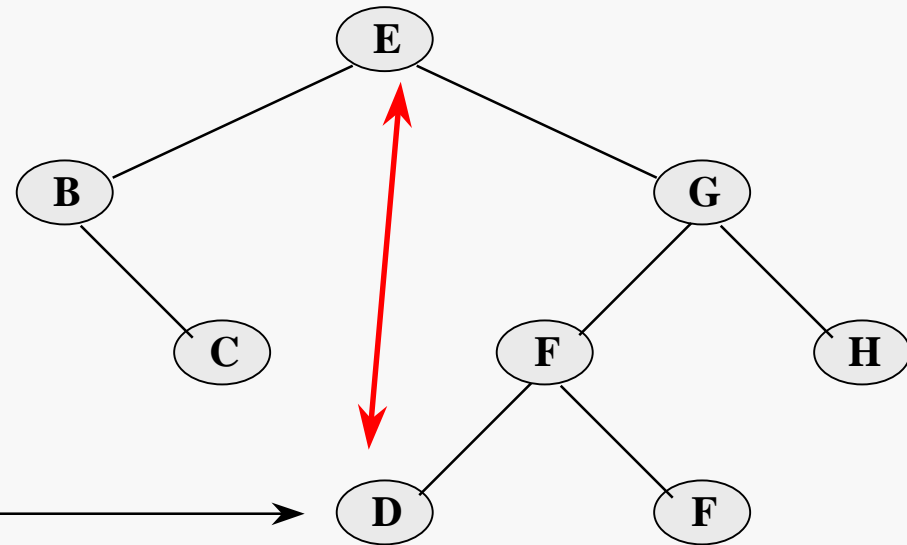


Fortunately, the smallest value will always lie in the left-most node of the subtree.

So, we first find the left-most node of the right subtree, and then swap data values between it and the targeted node.

Note that at this point we don't necessarily have a BST.

Now we must delete the swapped value from the right subtree.



That looks straightforward here since the node in question is a leaf. However...

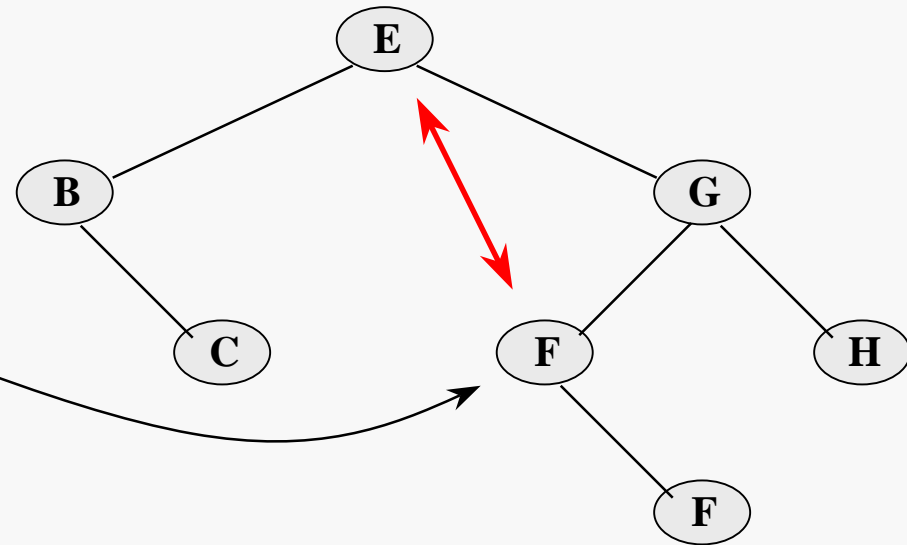
- the node will NOT be a leaf in all cases
- the occurrence of duplicate values is a complicating factor (so we might want to have a `DeleteRightMinimum()` function to clean up at this point).

Deleting the Minimum Value

Suppose we want to delete the value 'E' from the BST:

After swapping the 'F' with the 'E', we must delete

We must be careful to not confuse this with the other node containing an 'F'.



Also, consider deleting the value 'G'. In this case, the right subtree is just a leaf node, whose parent is the node originally targeted for deletion.

Moral: be careful to consider ALL cases when designing.

A BST class may be derived from a general binary tree class, saving some work of implementation.

```
template <class Data> class BSTreeT : public BinaryTreeT<Data> {
protected:

    bool InsertHelper(Data D, BinNodeT<Data>* sRoot);
    bool DeleteHelper(Data D, BinNodeT<Data>* sRoot);
    bool moveToDataHelper(Data D, BinNodeT<Data>* sRoot);

public:
    BSTreeT();
    BSTreeT(Data newData);

    bool Insert(Data D);
    bool Delete(Data D);
    bool moveToData(Data D);
    bool setCurrentData(Data D);

    ~BSTreeT();
};

// . . . continues . . .
```

The insert/delete rules are different for a BST, so we must override the base class versions of those.

Searching is different in a BST, so we must override that as well.

Finally, we should override the set data function in order to prevent a client from breaking the data organization required in a BST.

The BST constructors can just use the base class constructors since the data members are identical.

```
template <class Data>
BSTreeT<Data>::BSTreeT() : BinaryTreeT<Data>() {

}

template <class Data>
BSTreeT<Data>::BSTreeT(Data newData) : BinaryTreeT<Data>(newData) {

}
```

An empty BST destructor is provided. Recall that the base class destructor will be invoked automatically for a derived class object.

```
template <class Data>
BSTreeT<Data>::~~BSTreeT() {

}
```

The BST function takes advantage of the data organization:

```
template <class Data>
bool BSTreeT<Data>::moveToData(Data toFind) {

    if (Root == NULL) return false;
    return (moveToDataHelper(toFind, Root));
}
```

```
template <class Data>
bool BSTreeT<Data>::moveToDataHelper(Data toFind, BinNodeT<Data>* sRoot) {

    if (sRoot == NULL) return false;

    if (sRoot->getData() == toFind) {
        Current = sRoot;
        return true;
    }

    if (toFind < sRoot->getData())
        return moveToDataHelper(toFind, sRoot->getLeft());
    else
        return moveToDataHelper(toFind, sRoot->getRight());
}
```

Uses operator== for the class Data, which is customized by the client for the particular application at hand.

Search direction is determined by relationship of target data to data in current node.

The public Insert function is logically identical to that in the base class:

```
template <class Data>
bool BSTreeT<Data>::Insert(Data D) {

    if (Root == NULL) {
        BinNodeT<Data>* Temp = new BinNodeT<Data>(D);
        if (Temp == NULL) return false;
        Root = Current = Temp;
        return true;
    }
    return InsertHelper(D, Root);
}
```

Warning:

because the BST definition in these notes allows for duplicate data values to occur, the logic of insertion is slightly different from that presented in Kruse/Ryba.

However, the implementation of the helper function is substantially different.

In the BST, we must follow the route determined by comparing the new data value to the existing data values in order to reach the appropriate point for an insertion.

The modifications are left to the reader... there are substantial differences from the implementation given in Kruse/Ryba due to the encapsulation of the data in the BinNodeT class.

The public Delete function is very different from that in the base class. The public function still needs to handle deletion of the Root node; however, the logic is entirely different in the case where the Root node has two children:

```
template <class Data>
bool BSTreeT<Data>::Delete(Data D) {

    if (Root == NULL) return false;

    if (Root->getData() == D) {
        BinNodeT<Data>* toDelete = Root;

        if (Root->getLeft() == NULL) {
            Root = Root->getRight();
            delete toDelete;
            return true;
        }
        else if (Root->getRight() == NULL) {
            Root = Root->getLeft();
            delete toDelete;
            return true;
        }
    }

    // . . . continues . . .
}
```

Easy cases, where...

... Root node has no left subtree...

... Root node has a left subtree, but no right subtree...

The public Delete function is very different from that in the base class. The public function still needs to handle deletion of the Root node; however, the logic is entirely different in the case where the Root node has two children:

```
// . . . continued . . .  
  
    else {  
  
        BinNodeT<Data>* Look = Root->getRight();  
        while (Look->getLeft() != NULL)  
            Look = Look->getLeft();  
  
        Root->setData(Look->getData());  
        Look->setData(D);  
  
        return DeleteRightMinimum(Root);  
    }  
}  
  
return DeleteHelper(D, Root);  
}
```

Hard case where Root node has both a left subtree and a right subtree...

... must find the closest successor of the data value in the Root node ... that will be in the left-most node of the right subtree...

... then swap that data value into the Root node...

... and then chase that data value down the right subtree and delete it there
...

The protected Delete helper function shares similarities with, but is also very different from, that in the base class.

The changes are logically similar to those shown in the public Delete function, and are left to the reader...

Warnings:

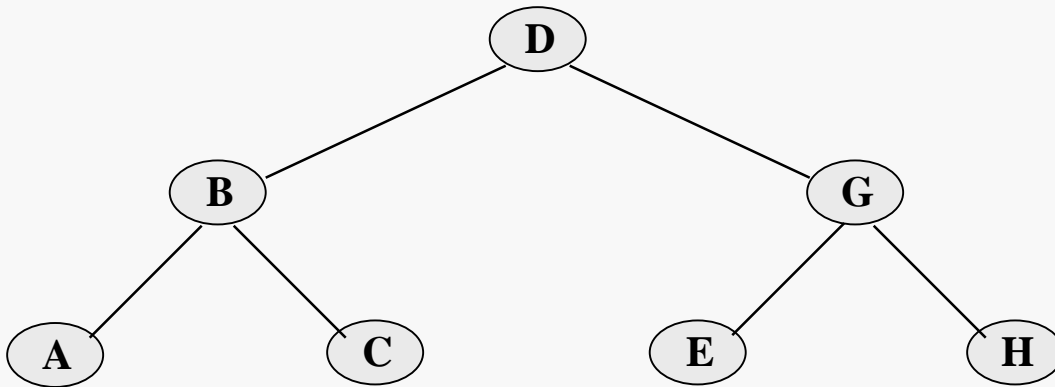
The logic of deletion in a BST differs from that shown earlier for a general binary tree, since in a BST the search is driven by the organization of the data elements in the BST.

The logic of deletion is different than that shown in Kruse/Ryba because we allow duplicate key values to occur.

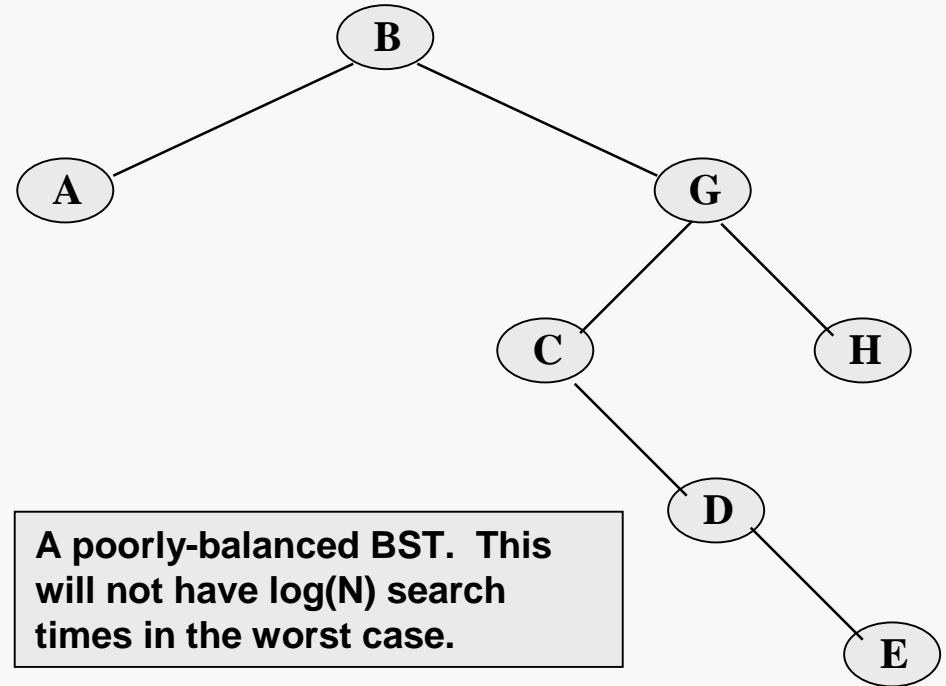
The implementation is (again) also very different because of the encapsulation of the private data in the `BinNodeT` class, in contrast to the public struct elements used in Kruse/Ryba.

QTP: Could we use the maximum value from the left subtree (as in Kruse/Ryba) when deleting a node with two subtrees?

However, a BST with N nodes does not always provide $O(\log N)$ search times.



A well-balanced BST. This will have $\log(N)$ search times in the worst case.



A poorly-balanced BST. This will not have $\log(N)$ search times in the worst case.

What if we inserted the values in the order:

A B C D E G H