Binary Search Trees

The general binary tree shown in the previous chapter 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:

1. All keys in the left subtree (if there is one) are less than the key in the root node.
2. All keys in the right subtree (if there is one) are greater than (or equal to)* the key in the root node.
3. The left and right subtrees of the root are binary search trees.

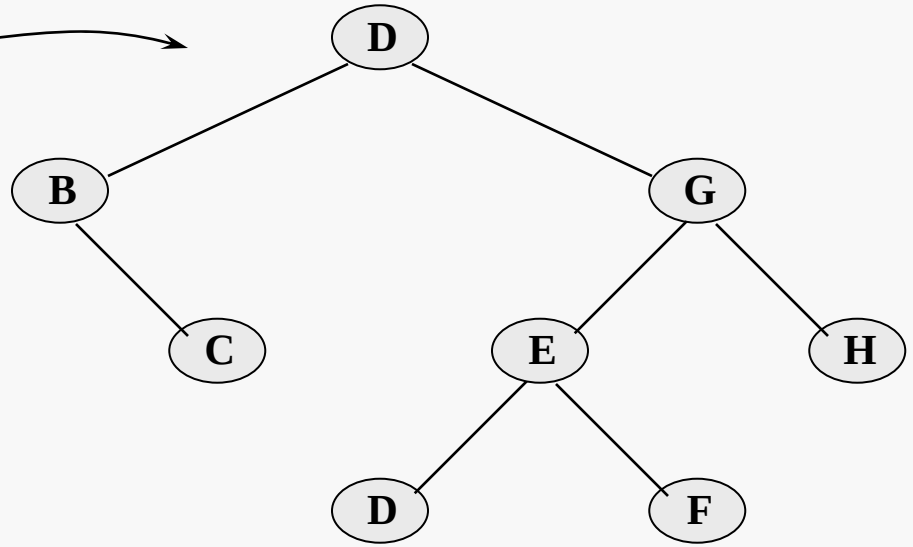
* In many uses, duplicate values are not allowed.

BST Insertion

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

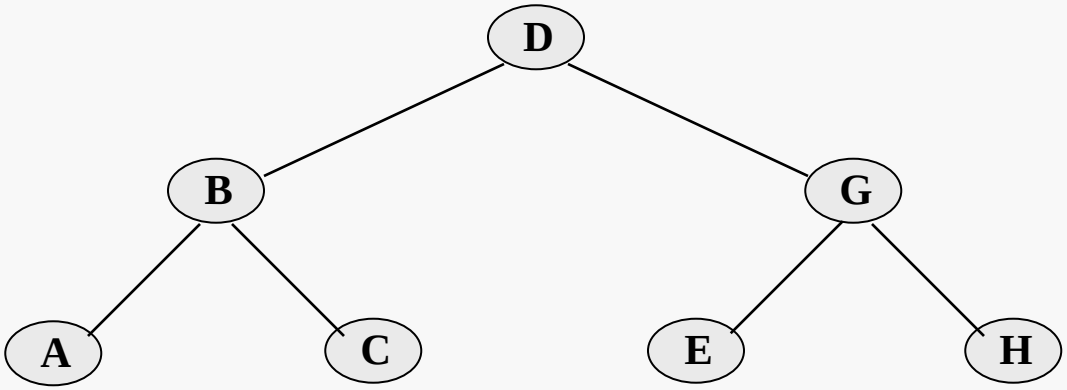
or

E B C D D F G H

Searching in a BST

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.

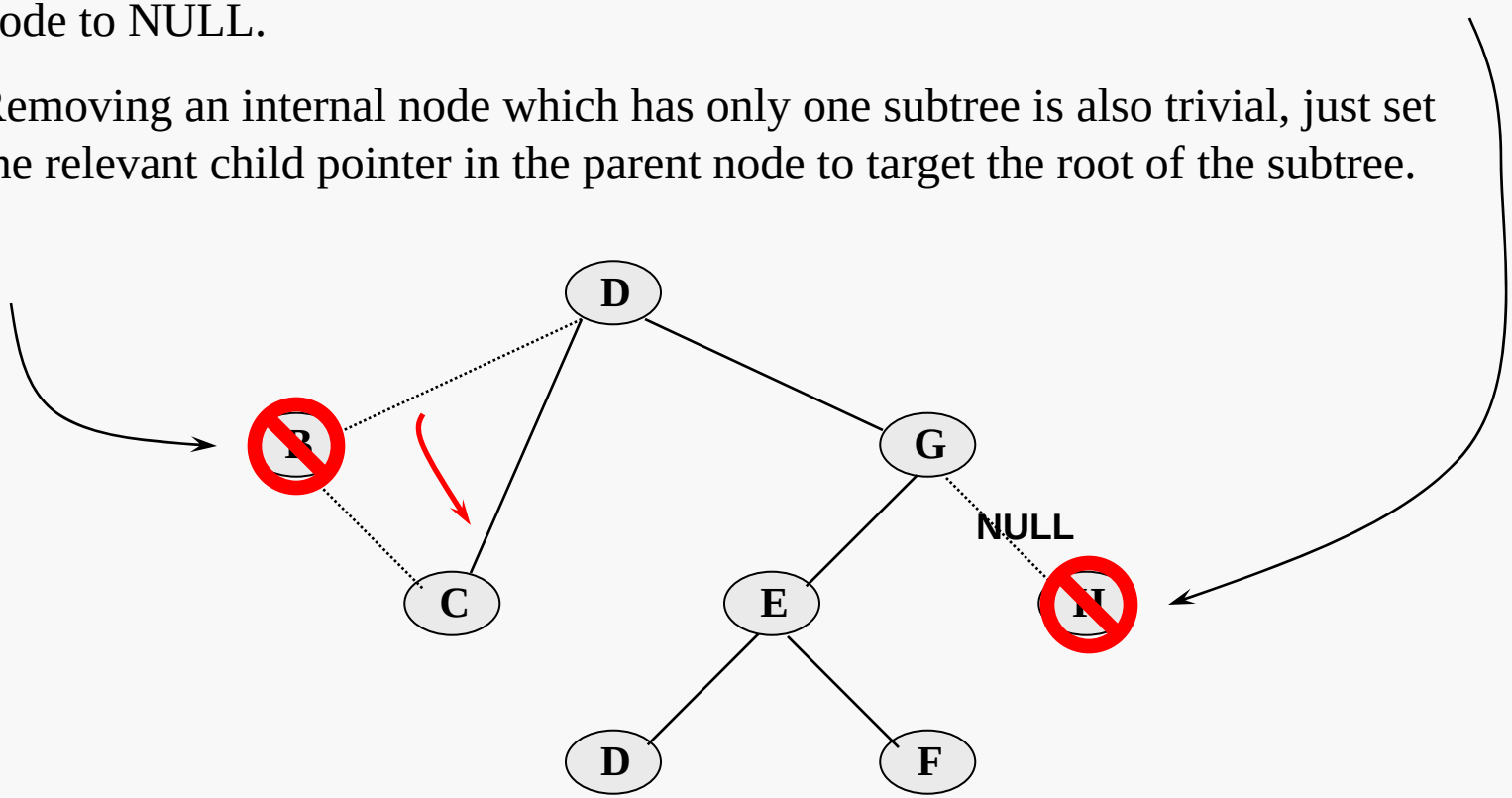


Trace searching for the key value E.

BST Deletion

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 being 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.

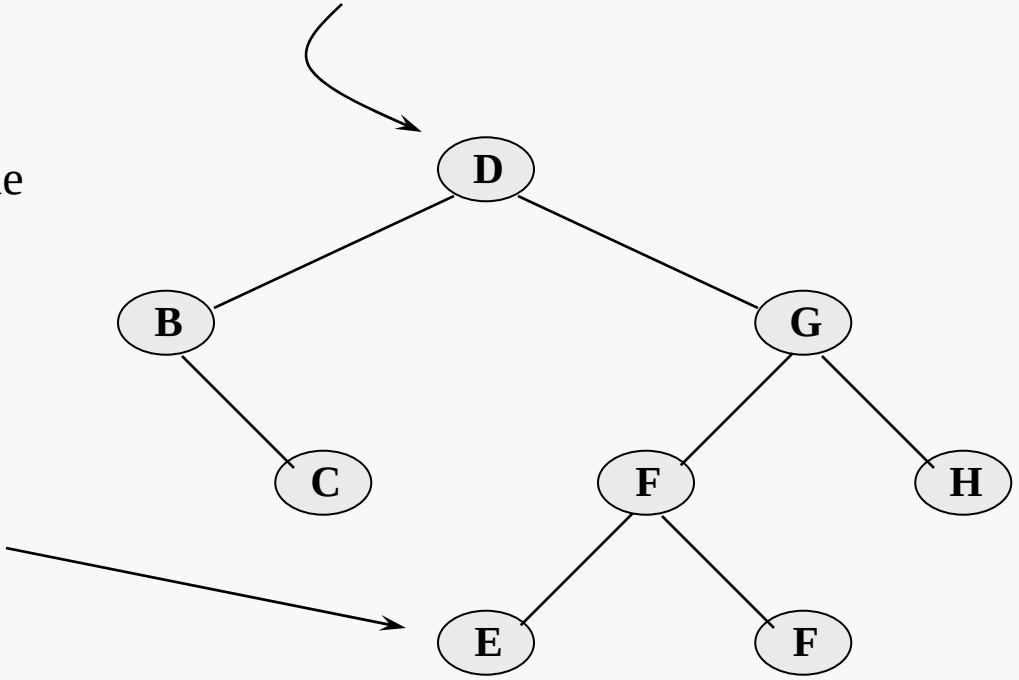


BST Deletion

- 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 may take the smallest value from the right subtree, which would be the closest succcessor of the value being deleted.

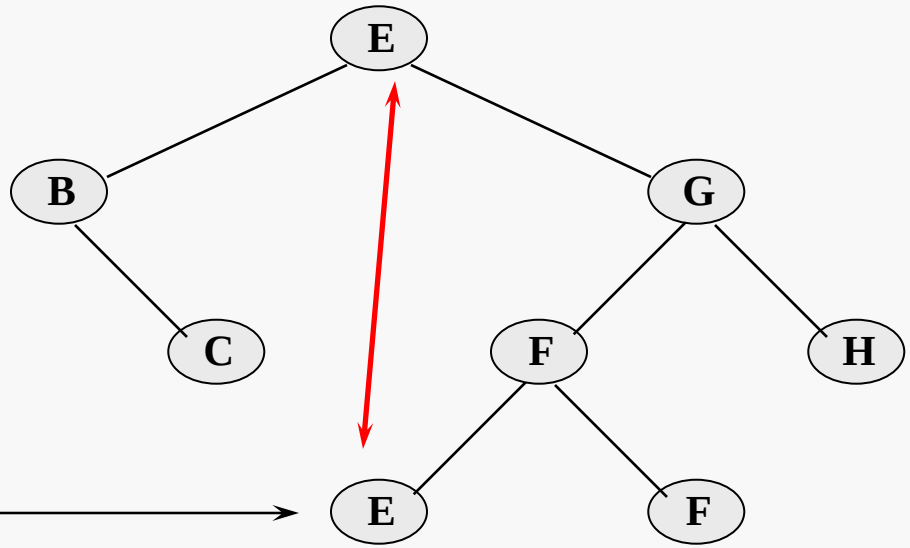


Fortunately, the smallest value will always lie in the left-most node of the subtree.

BST Deletion

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 copied value from the right subtree.

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

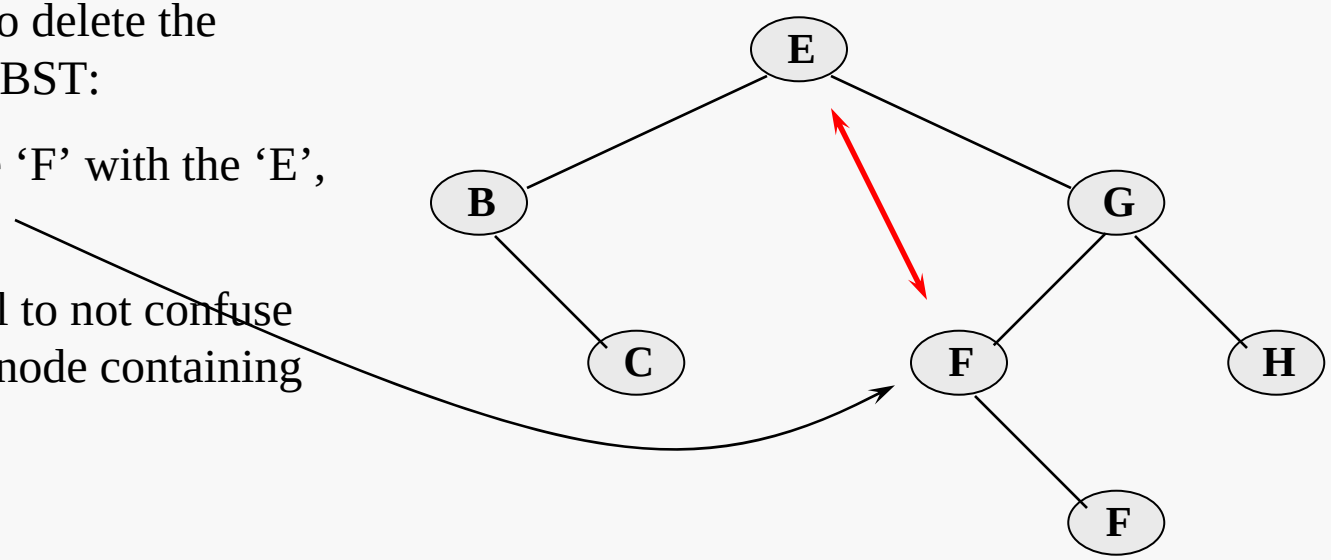
- the node will NOT be a leaf in all cases
- the occurrence of duplicate values is a complicating factor
- so we might want to have a helper function to clean up at this point

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.

Generic BST Interface

Here's a partial generic BST interface, adapted from Weiss:

```
public class BST extends Comparable<? super AnyType>> {

    private static class BinaryNode<AnyType> {
       . . .
    }

    private BinaryNode<AnyType> root;

    public BST( ) { . . . }
    public void    clear( ) { . . . }
    public boolean isEmpty( ) { . . . }
    public boolean contains( AnyType x ) { . . . }
    public AnyType find( AnyType x ) { . . . }
    public AnyType findMin( ) { . . . }
    public AnyType findMax( ) { . . . }
    public void    insert( AnyType x ) { . . . }
    public void    delete( AnyType x ) { . . . }
    public void    printTree( ) { . . . }
    . . .
}
```

Generic BST Interface

Here's a partial generic BST interface, adapted from Weiss:

```
public class BST extends Comparable<? super AnyType>> {
    . . .
}
```

```
public int compareTo(Object o)
```

Returns: a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

See Weiss 1.5.5

Generic BST Node

Here's a partial generic BST interface, adapted from Weiss:

```
private static class BinaryNode<AnyType> {
   // Constructors
   BinaryNode( AnyType theElement ){

      this( theElement, null, null );
   }

   BinaryNode( AnyType theElement, BinaryNode<AnyType> lt,
               BinaryNode<AnyType> rt ){

      element  = theElement;
      left     = lt;
      right    = rt;
   }

   AnyType              element;  // The data in the node
   BinaryNode<AnyType> left;     // Left child
   BinaryNode<AnyType> right;    // Right child
}
```

BST `contains()` Implementation

The BST `contains()` function takes advantage of the BST data organization:

```
public boolean contains( AnyType x ) {
   return contains( x, root );
}
```

```
private boolean contains( AnyType x, BinaryNode<AnyType> t ) {
   if ( t == null )
      return false;

   int compareResult = x.compareTo( t.element );

   if ( compareResult < 0 )
      return contains( x, t.left );
   else if ( compareResult > 0 )
      return contains( x, t.right );
   else
      return true;    // Match
}
```

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

BST `find()` Implementation

The BST `find()` function provides client access to data objects within the tree:

```
public AnyType find( AnyType x ) {
   return find( x, root );
}
```

```
private AnyType find( AnyType x, BinaryNode<AnyType> t ) {
   if ( t == null )
      return null;

   int compareResult = x.compareTo( t.element );

   if ( compareResult < 0 )
      return find( x, t.left );
   else if ( compareResult > 0 )
      return find( x, t.right );
   else
      return t.element;    // Match
}
```

Warning:

be sure you understand the potential dangers of supplying this function… and the benefits of doing so…

BST `insert()` Implementation

The public `Insert()` function is just a stub to call the recursive helper:

```
public void insert( AnyType x ) {

   root = insert( x, root );
}
```

Warning:

the BST definition in these notes does not allow for duplicate data values to occur, the logic of insertion may need to be changed for your specific application.

The stub simply calls the helper function..

The helper function must find the appropriate place in the tree to place the new node.

The design logic is straightforward:

- locate the parent "node" of the new leaf, and
- hang a new leaf off of it, on the correct side

BST `insert()` Helper

The `insert()` helper function:

```
private BinaryNode<AnyType>
insert( AnyType x, BinaryNode<AnyType> t ) {
   if ( t == null )
      return new BinaryNode<AnyType>( x, null, null );

   int compareResult = x.compareTo( t.element );

   if ( compareResult < 0 )
      t.left = insert( x, t.left );
   else if ( compareResult > 0 )
      t.right = insert( x, t.right );
   else
      ;  // Duplicate; do nothing
   return t;
}
```

When the parent of the new value is found, one more recursive call takes place, passing in a NULL pointer to the helper function.

Note that the insert helper function must be able to modify the node pointer parameter, and that the search logic is precisely the same as for the find function.

BST `delete()` Implementation

The public `delete()` function is very similar to the insertion function:

```
public void delete( AnyType x ) {

   root = delete( x, root );
}
```

The `delete()` helper function design is also relatively straightforward:

- locate the parent of the node containing the target value
- determine the deletion case (as described earlier) and handle it:
  - parent has only one subtree
  - parent has two subtrees

The details of implementing the delete helper function are left to the reader…

Parent Pointers

Some binary tree implementations employ parent pointers in the nodes.

- increases memory cost of the tree (probably insignificantly)
- increases complexity of insert/delete/copy logic (insignificantly)
- provides some unnecessary alternatives when implementing insert/delete
- may actually simplify the addition of iterators to the tree (later topic)

Some Refinements

The given BST template may also provide additional features:

- a function to provide the size of the tree
- a function to provide the height of the tree
- a function to display the tree in a useful manner

It is also useful to have some instrumentation during testing. For example:

- log the values encountered and the directions taken during a search

This is also easy to add, but it poses a problem since we generally do not want to see such output when the BST is used.

I resolve this by adding some data members and mutators to the template that enable the client to optionally associate an output stream with the object, and to turn logging of its operation on and off as needed.