6.1 Binary Search Trees

A binary search tree is a data structure that can be used for storing sorted data. We will begin by discussing an Abstract Sorted List or Sorted List ADT and then proceed to describe the structure.

6.1.1 Abstract Sorted Lists

An abstract sorted list or Sorted List ADT is a model for storing implicit linearly ordered data. Given a collection of objects that have an implicit linear ordering, it may be useful to ask:

1. What is the largest and smallest objects?
2. What is the k^{th} object in the collection? (The smallest is the first.)
3. Given an interval [*a*, *b*], what are all the objects in the container that fall in that interval?
4. Given an object in the container, what are the previous smaller or next larger objects?

Because the linear ordering is implicit, new objects should be automatically located in the correct positions based on their properties. Therefore, we will have an arbitrary

```
void insert( Type const & )
```

member function and explicit insertion functions such as

```
void push_front( Type const & )
```

No longer make sense—this would only be appropriate if it was already known that the object being inserted is smaller than the smallest object in the container.

6.1.2 Description

Assuming that it is intended that new objects be inserted into a container storing sorted data or that objects may be removed from that container, it is not possible to use a simple array or linked list to store sorted data. Either the insertion will be fast (the object is just appended to the data structure) but then operations such as accessing the front are $\Theta(n)$ or the data is stored sorted internally in which case accessing the front is $\Theta(1)$, but the insertions and removals themselves are $O(n)$.

Recall that with a binary tree, we identify the two sub-trees as being either the *left* or *right* sub-trees. Consequently, there is already an order between these children, but we can also include the node itself in this relationship: Given any node,

1. All entries in the left sub-tree are less than the value stored in the node, and
2. All entries in the right sub-tree are greater than the root.

For example, if the root stored 42 and the binary search tree contained the entries 3, 15, 22, 23, 29, 40, 46, 50, 57, 59, 65, 73, 88 and 91, they would appear, relative to 42 in the sub-trees shown in Figure 1.

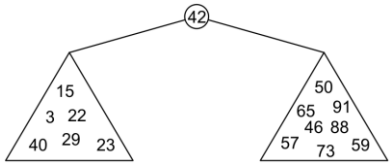

Figure 1. The locations relative to 42 of fourteen other numbers.

Because of the linear ordering, it is possible to conduct a search based on the value at a node:

1. If the value we are searching for equals what is stored in the current node, we are done,
2. If the value is less than the current node, search the left sub-tree, otherwise
3. The value is greater than the current node, so search the right sub-tree.

This recursive algorithm continues until the object is found or the left or right sub-tree is a *null sub-tree*.

Examples of search trees are shown in Figure 2.

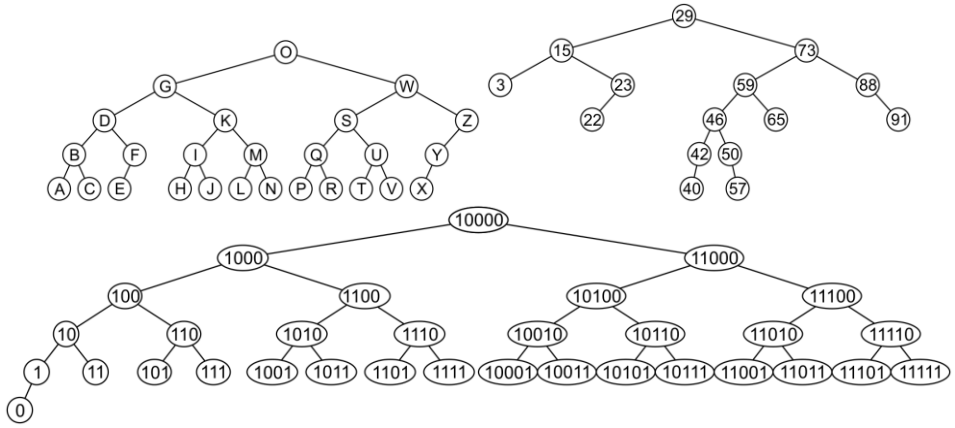

Figure 2. Examples of binary search trees.

Unfortunately, it is possible to create *degenerate* binary search trees, as is shown in Figure 3.

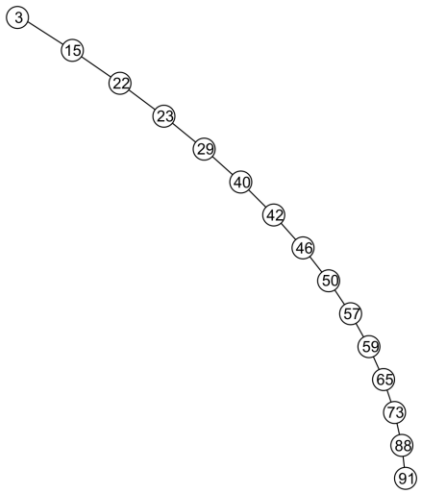

Figure 3. A degenerate binary search tree.

A degenerate tree is equivalent to a linked list and therefore the run time of all operations would be equal to the corresponding operations in a linked list. Consequently, there are many binary search trees that could store the same data. Some will have desirable characteristics while others will be sub-optimal. Figure 4 shows four binary search trees that all store the same 15 objects.

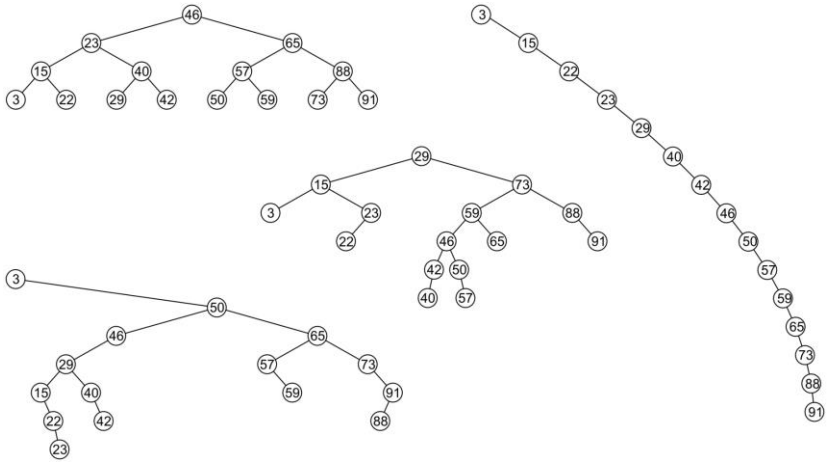

Figure 4. Four binary search trees.

6.1.3 Duplicate Elements

Reminder: we will assume that all entries of a binary search tree are unique. If a duplicate element is inserted into a binary search tree, that insertion will be ignored.

6.1.4 Implementation

We will implement a binary search tree by deriving from the `Binary_node` class:

```
#include "Binary_node.h"

template <typename Type>
class Binary_search_tree;

// The ":public Binary_node<Type>" indicates that this class is derived from the Binary_node
// class.  Every 'binary search node' is a 'binary node'.
template <typename Type>
class Binary_search_node:public Binary_node<Type> {
    using Binary_node<Type>::element;
    using Binary_node<Type>::left_tree;
    using Binary_node<Type>::right_tree;

    public:
        Binary_search_node( Type const & );

        Binary_search_node *left() const;
        Binary_search_node *right() const;

        bool empty() const;
        int size() const;
        int height() const;
        bool leaf() const;
        Type front() const;
        Type back() const;
        bool find( const Type & ) const;

        void clear();
        bool insert( Type const &, Binary_search_node *& );
        bool  erase( Type const &, Binary_search_node *& );

        friend class Binary_search_tree<Type>;
};
```

Because this class is derived from the Binary_node class, it inherits the member functions

```
        Type retrieve() const;
        Binary_search_node *left() const;
        Binary_search_node *right() const;
```

The constructor simply calls the constructor of the base class:

```
template <typename Type>
Binary_search_node<Type>::Binary_search_node( Type const &obj ):
Binary_node<Type>( obj ) {
    // Just calls the constructor of the base class
    //  - 'element' is assigned the value of obj
    //  - Both left_tree and right_tree are assigned nullptr.
}
```

We already inherit the function `Type retrieve() const;` and thus we need not re-implement this function; however, the return type of left and right now change and therefore we must re-implement these:

```
template <typename Type>
Binary_search_node<Type> *Binary_search_node<Type>::left() const {
        return reinterpret_cast<Binary_search_node *>( Binary_node<Type>::left() );
}

template <typename Type>
Binary_search_node<Type> *Binary_search_node<Type>::right() const {
        return reinterpret_cast<Binary_search_node *>( Binary_node<Type>::right() );
}
```

NOTE: You will not be required to understand the nuances of derived template classes as shown in these examples. They are given for completeness.

6.1.4.1 Front

The minimum or front entry of a binary search tree with no left sub-tree is the element of the current node. Otherwise, the minimum entry is the minimum entry of the left sub-tree. An example is shown in Figure 7.

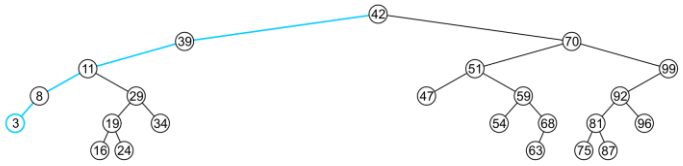

Figure 7. The minimum (front) element is 3.

This can be implemented as follows:

```
template <typename Type>
Type Binary_search_node<Type>::front() const {
    return ( left()->empty() ) ? retrieve() : left()->front();
}
```

6.1.4.2 Back

The maximum or back entry of a binary search tree with no right sub-tree is the element of the current node. Otherwise, the maximum entry is the maximum entry of the right sub-tree. An example is shown in Figure 8.

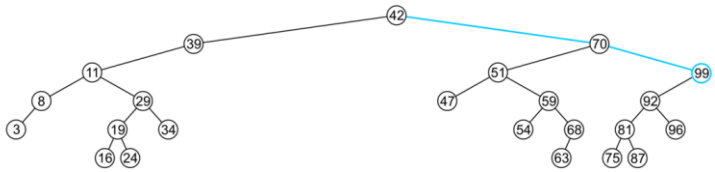

Figure 8. The maximum (back) element is 99.

This can be implemented as follows:

```
template <typename Type>
Type Binary_search_node<Type>::back() const {
    return ( right()->empty() ) ? retrieve() : right()->back();
}
```

6.1.4.3 Find

Finding an object in a binary search tree begins at the root: if the object is at the root, we are finished; otherwise, we compare the object with the root:

1. If the object is less than the root, we search the left sub-tree, and
2. If the object is greater than the root, we search the right sub-tree.

If the indicated sub-tree is empty, we are finished—the object was not found. Otherwise, we will recurse on this new node. The run time is O(*h*). Figures 9 and 10 demonstrate this.

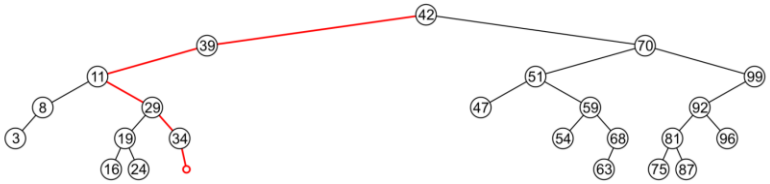

Figure 9. A search for 36 ultimately ends in an empty node—the tree does not contain 36.

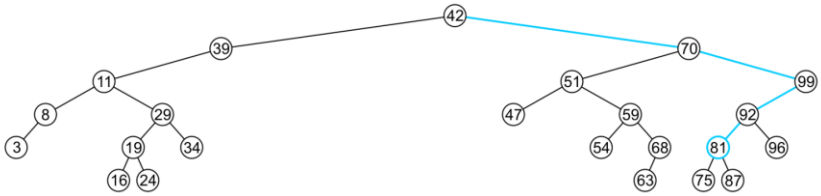

Figure 10. A search for 81 finds the node after four steps.

The implementation is similarly recursive:

```
template <typename Type>
bool Binary_search_node<Type>::find( Type const &obj ) const {
    if ( empty() ) {
        return false;
    } else if ( retrieve() == obj ) {
        return true;
    }

    return ( obj < retrieve() ) ? left()->find( obj ) : right()->find( obj );
}
```

6.1.4.4 Insert

To insert a new object into a binary search tree, we will follow the same algorithm as `bool find( Type const & ) const`; however, when we reach the empty node, we will add a new node in that location storing the object. Because we are assuming uniqueness, if we ever find that object during our search, we return without inserting the copy. A new insertion could occur at any of the empty nodes in a tree, as is shown in Figure 11.

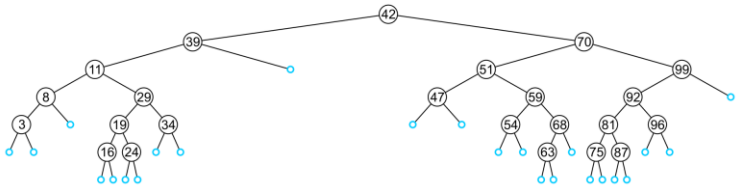

Figure 11. The $n + 1$ empty nodes where new insertions could occur.

For example, the indicated nodes in Figures 12, 13, and 14 can only store specific values.

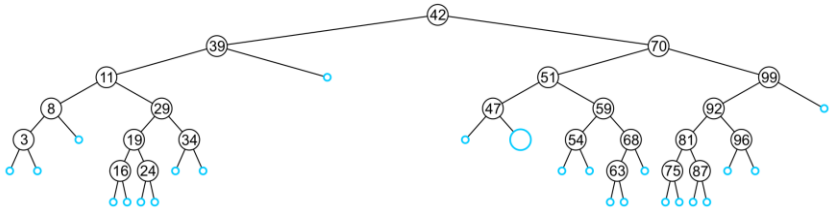

Figure 12. Values between 47 and 51 could be inserted at the indicated node.

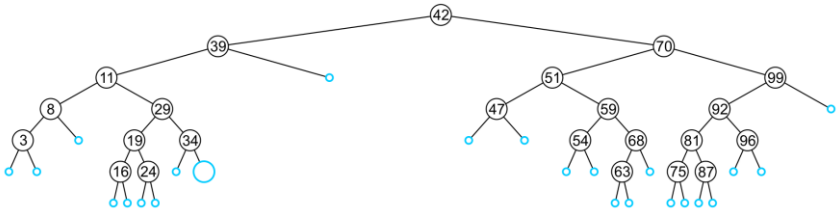

Figure 13. Values between 34 and 39 could be inserted at the indicated node.

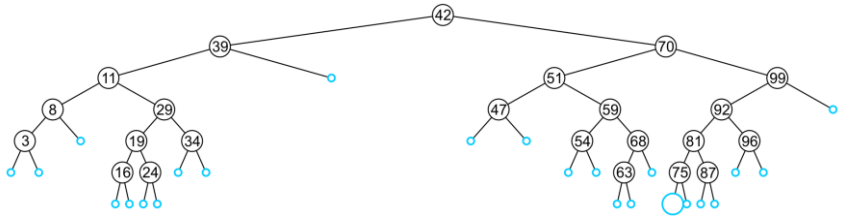

Figure 14. Values between 70 and 75 could be inserted at the indicated node.

As two examples of insertions,

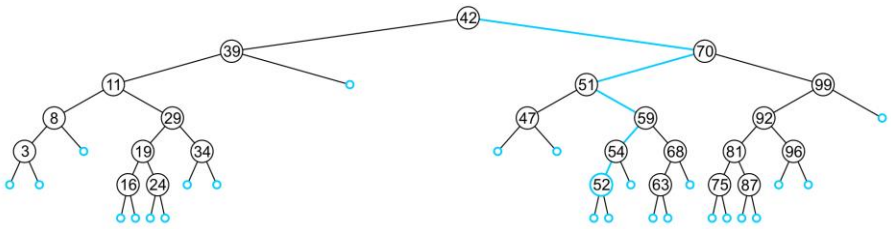

Figure 15. The value 52 would be inserted as a new left child of node 54.

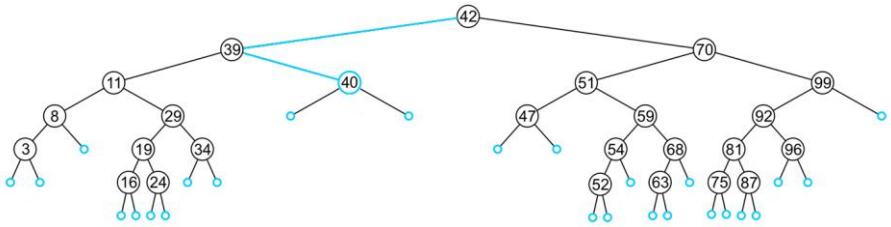

Figure 16. The value 40 would be inserted as a new right child of node 39.

As indicated, the implementation is very similar to that of find. Note that if neither condition is met, that indicates that `obj == retrieve()` and therefore we simply return without continuing the recursion.

In implementing this function, there is one critical point we should note: if we traverse into an empty node and then determine that we must now insert a new node at this location, we must update a member variable of the parent which called insert on this particular null sub-tree. Consequently, we must have a mechanism for changing that value in the parent while insert is being called on this node.

Now, by default, C++ is pass-by-value: each parameter is a copy of the argument and any modifications to that parameter will not affect the original argument. In C++, however, an argument can by passed-by-reference by placing an `&` immediately in front of the parameter name. Now, up to this point, we have already used this with an argument `obj`; however, here we added the modifier const. By passing-by-reference but as a constant, the procedure is not allowed to modify the value. In the function `erase`, we will pass an argument with the parameter name `ptr_to_this` that is not constant and therefore the value can be modified appropriately. Figure 17 shows how, if erase is called on the node indicated as this, the parameter `ptr_to_this` would be the argument `left_tree` of the parent.

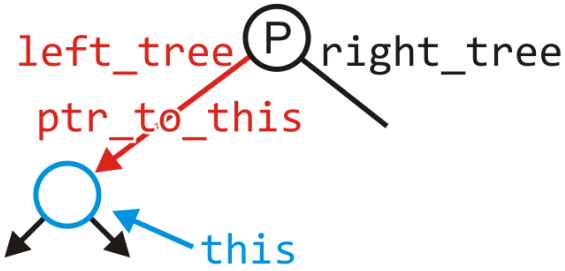

Figure 17. The member variable `left_tree` would be passed as the pass-by-reference parameter `ptr_to_this` when calling erase on the node marked `this`.

Our implementation is:

```
template <typename Type>
bool Binary_search_node<Type>::insert( Type const &obj, Binary_search_node *&ptr_to_this ) {
    if ( empty() ) {
        ptr_to_this = new Binary_search_node<Type>( obj );
        return true;
    } else if ( obj < retrieve() ) {
        return left()->insert( obj, left_tree );
    } else if ( obj > retrieve() ) {
        return right()->insert( obj, right_tree );
    } else {
        return false;
    }
}
```

6.1.4.5 Erase

Unlike insertions where any new object can always be inserted at an appropriately located empty node, an object being erased from a binary search tree could occur either at an internal or leaf node. If the object is a leaf node, it is easy: simply set the pointer in the parent that is storing the address of that node to zero and delete the node. Figure 18 shows the result of removing node 75 and Figure 19 shows the result of removing node 40.

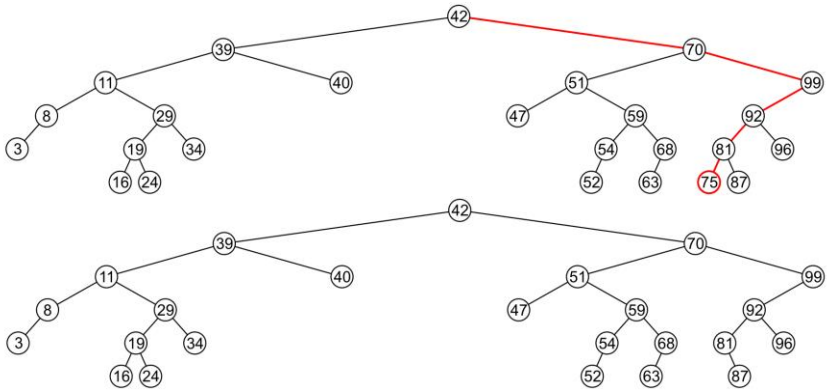

Figure 18. Removing 75 from this binary search tree.

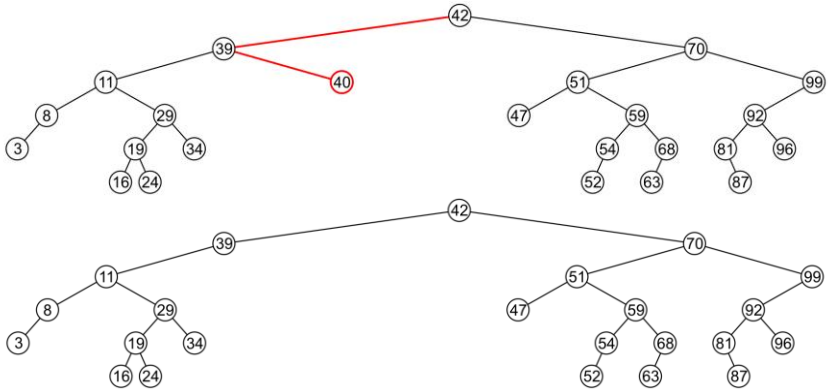

Figure 19. Removing 40 from this binary search tree.

If the node being removed has only one child—that child can be promoted to being the child of the parent of the object being removed. Figures 20 and 21 show the result of removing 8 and 39, respectively, from a binary search tree.

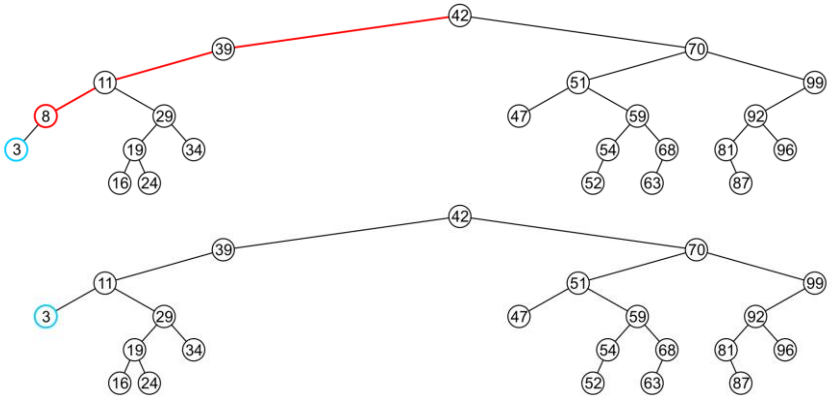

Figure 20. Removing 8 from a binary search tree. The sub-tree rooted at 3 is promoted to be the left child of 11.

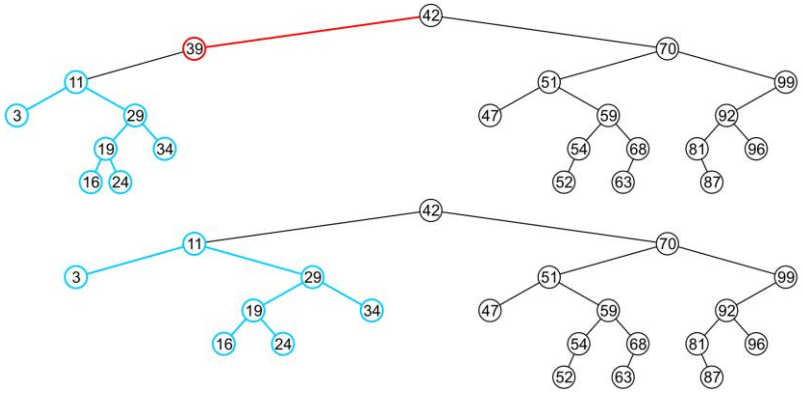

Figure 21. Removing 39 from a binary search tree. The sub-tree rooted at 11 is promoted to be the left child of 42.

If we remove 99 from the binary search tree in Figure 21, we get the tree in Figure 22.

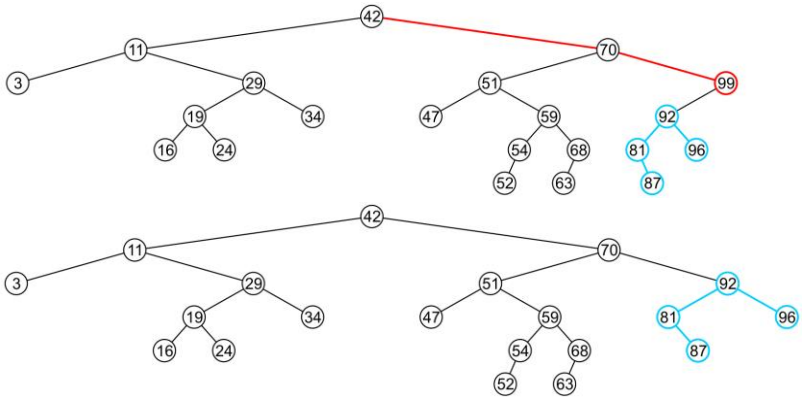

Figure 22. Removing 99 from a binary search tree. The sub-tree rooted at 92 (originally the left sub-tree of 99) is promoted to the right sub-tree of 70.

The last case we must consider is removing a full node. If you look at Figure 23 and ask what could you do to remove 42, you will quickly realize that you cannot simply promote one of the children.

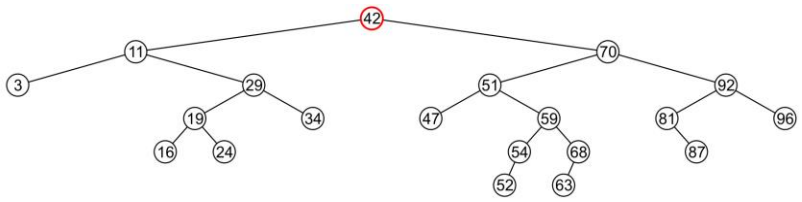

Figure 23. Removing the root from a binary search tree.

If we were to promote 70 to the root, we would have to deal with the left sub-tree of 70: with 70 in the root, that sub-tree would now have to be moved to the right side of the binary search tree.

Instead, the next smaller entry prior to the root is the largest entry of the left sub-tree: 34. The next larger entry after the root is the smallest entry of the right sub-tree: 47. One simple solution is to replace the root with one of these entries and then proceed to remove that node from the corresponding sub-tree. We will use the following rule:

> If the node being removed is a full node, promote the smallest element from the right sub-tree and then proceed to remove that smallest element from the right sub-tree.

In our example, we would promote 47 to the root, as is shown in Figure 24 and then proceed to remove 47 from the right sub-tree.

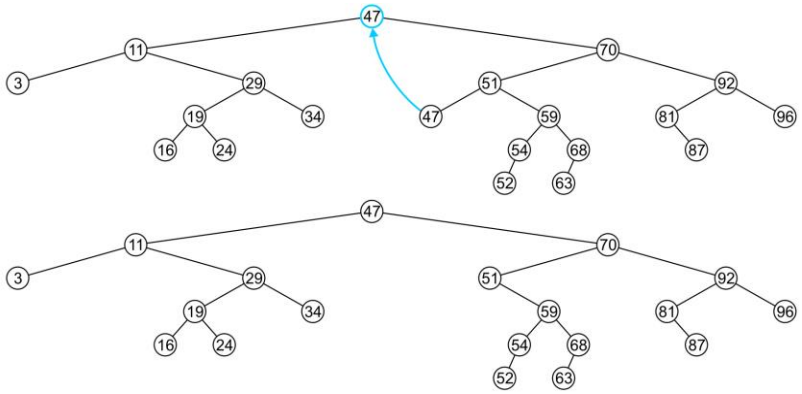

Figure 24. Promoting 47 and removing it from the right sub-tree.

Not meaning to choose just the root, but removing 47 now requires us to promote the smallest entry of the right sub-tree, 51, to the root. The node 51, however, is not a leaf node, and therefore the sub-tree rooted at 59 must be promoted to be the left sub-tree of 70.

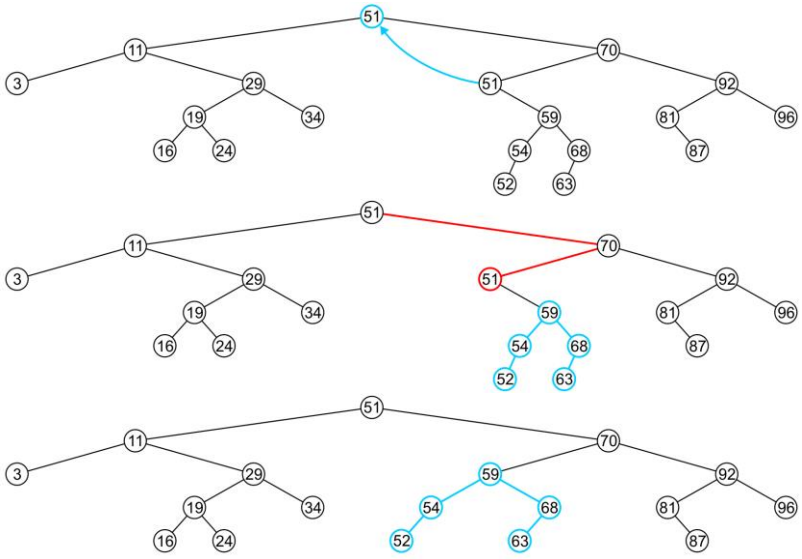

Figure 25. Removing the root a second time.

The first example had us removing a promoted node that was a leaf node and the second had that promoted node with one child. It should be obvious why the promoted node will never be a full node.

Like the implementation of insert, this member function must also possibly modify the appropriate member variable of the parent. Thus, like the insert, we will pass the member variable storing the address of the node as a second argument.

```
template <typename Type>
bool Binary_search_node<Type>::erase( Type const &obj, Binary_search_node<Type> *&ptr_to_this ) {
    if ( empty() ) {
        return false;
    } else if ( obj == retrieve() ) {
        if ( leaf() ) {                                 // leaf node
            ptr_to_this = 0;
            delete this;
        } else if ( !left()->empty() && !right()->empty() ) {    // full node
            element = right()->front();
            right()->erase( retrieve(), right_tree );
        } else {                                        // only one child
            ptr_to_this = ( left()->empty() ) ? right() : left();
            delete this;
        }

        return true;
    } else if ( obj < retrieve() ) {
        return left()->erase( obj,  left_tree );
    } else {
        return right()->erase( obj, right_tree );
    }
}
```

6.1.5 The `Binary_search_tree` Class

Recall how `Single_node` required a container class `Single_list` to store the `list_head` and `list_tail` member variables. We will do the same for this class, that is, we will create a `Binary_search_tree` class; however, unlike the `Single_list` class where all the work was done in the `Single_list` class, here, all the work is already done by the node class: for the most part, the member functions of the `Binary_search_tree` class will simply call the appropriate member function of the root node.

```
template <typename Type>
class Binary_search_tree {
    private:
        Binary_search_node<Type> *root_node;
    public:
        Binary_search_tree();
        ~Binary_search_tree();

        bool empty() const;
        int size() const;
        int height() const;
        Type front() const;
        Type back() const;
        bool find( Type const & obj ) const;

        void clear();
        void insert( Type const & obj );
        bool erase( Type const & obj );
};

template <typename Type>
Binary_search_tree<Type>::Binary_search_tree():
root_node( 0 ) {
    // does nothing
}

template <typename Type>
Binary_search_tree<Type>::~Binary_search_tree() {
    clear();
}

template <typename Type>
    void Binary_search_tree<Type>::clear() {
    root_node ->clear();
    root_node = 0;
}

template <typename Type>
bool Binary_search_tree<Type>::empty() const {
    return root_node->empty();
}

template <typename Type>
int Binary_search_tree<Type>::size() const {
     return root_node->size();
}
```

```
template <typename Type>
int Binary_search_tree<Type>::height() const {
     return root_node->height();
}

template <typename Type>
bool Binary_search_tree<Type>::find( Type const &obj ) const {
     return root_node->find( obj );
}

template <typename Type>
Type Binary_search_tree<Type>::front() const {
    if ( empty() ) {
        throw underflow();
    }

    return root_node->front();
}

template <typename Type>
Type Binary_search_tree<Type>::back() const {
    if ( empty() ) {
        throw underflow();
    }

    return root_node->back();
}

template <typename Type>
int Binary_search_tree<Type>::insert( Type const &obj ) {
    if ( empty() ) {
        root_node = new Binary_search_node<Type>( obj );
    } else {
        root_node->insert( obj );
    }
}

template <typename Type>
bool Binary_search_tree<Type>::erase( Type const &obj ) {
    if ( empty() ) {
        return false;
    }

    return root_node->erase( obj, root_node );
}
```

6.1.6 Other Relation-based Operations

To this point, we have now considered how to create and maintain a binary search tree. The purpose of this was to create a data structure that can store sorted data where insertions and removals were not O(*n*)—in this case, they are all O(*h*).

With the tree structure, however, it is necessary to ask: what are the run times of the operations that are very fast with an array:

1. Given an entry, what are the previous and next entries?
2. What is the k^{th} entry?

Recall, if we are not interested in such operations, there is no significance to storing linearly ordered data in either an array or a binary search tree—use a hash table (all operations Θ(1))!

6.1.6.1 Previous and Next Entries

We will quickly discuss finding the next largest entry:

1. If the right sub-tree is not empty, the next largest entry is the minimum entry of the right sub-tree,
2. Otherwise, it is the next largest entry along the path to the node from the root.

These are shown in Figures 26 and 27.

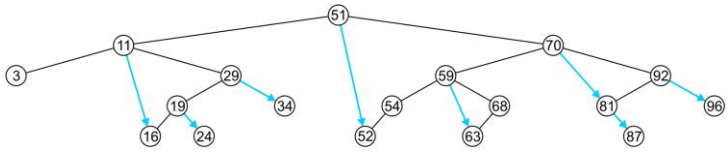

Figure 26. If the right sub-tree is not empty, the next largest entry is the minimum of the right sub-tree.

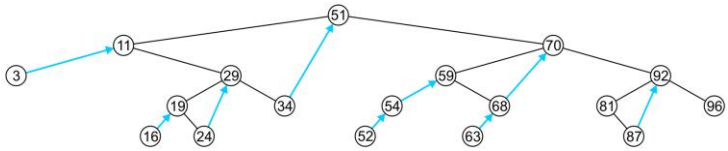

Figure 27. If the right sub-tree is empty, the next largest entry is the next largest value along the path from the root node to the node in question.

In both cases, the worst case requires one to follow a path from the root to a leaf node: O(*h*).

6.1.6.2 The k^{th} Entry

In order to find the k^{th} entry, it is necessary to know how many entries there are in each sub-tree.

If we are looking for the k^{th} entry (recall that k goes from 1 to $n - 1$):

1. If the left-sub-tree has $\ell = k$ entries, return the current node,
2. If the left sub-tree has $\ell < k$ entries, return the k^{th} entry of the left sub-tree,
3. Otherwise, the left sub-tree has $\ell > k$ entries, so return the $(k - \ell - 1)^{th}$ entry of the right sub-tree.

We would have to improve the size function so that it runs in $\Theta(1)$ time, in which case, the run time of this operation would be $O(h)$.

```
template <typename Type>
Type Binary_search_tree<Type>::at( int k ) const {
     return ( k <= 0 || k > size() ) ? Type() : root_node->at( k );
                                                    // Need to go from 0, ..., n - 1
}

template <typename Type>
Type Binary_search_node<Type>::at( int k ) const {
     if ( left()->size() == k ) {
          return retrieve();
     } else if ( left()->size() > k ) {
          return left()->at( k );
     } else {
          return right()->at( k - left()->size() – 1 );
     }
}
```

6.1.7 Maintaining an $\Theta(\ln(n))$ Height

Almost all operations on a binary search tree are $O(h)$. Unfortunately, if the tree degenerates into a linked list, the run times are no better than linked lists: $O(n)$. It is really easy to construct such a degenerate binary search tree, too: just insert the numbers 1 through n in order into an empty binary search tree.

We have seen, however, that the best case trees, perfect and complete binary trees, have heights that are $\Theta(\ln(n))$. This will be our *ideal case*. This suggests that we should look for tree structures that allow us to construct binary search trees, but to also perform operations to maintain a logarithmic height.

To this end, we will look at:

1. AVL trees,
2. B+ trees

There are, however, other descriptions of trees including red-black trees and BB[α] trees.