ABSTRACT SORTED LIST

• Previously, we discussed Abstract Lists: the objects are explicitly linearly ordered by the programmer.

• We will now discuss the Abstract Sorted List:
  • The relation is based on an implicit linear ordering.

• Certain operations no longer make sense:
  • push_front and push_back are replaced by a generic insert.

• Queries that may be made about data stored in a Sorted List ADT include:
  • Finding the smallest and largest entries.
  • Finding the kth largest entry.
  • Find the next larger and previous smaller objects of a given object which may or may not be in the container.
  • Iterate through those objects that fall on an interval [a, b].
BINARY SEARCH TREES

DEFINITION

• Graphically, we may relationship

- Each of the two sub-trees will themselves be binary search trees
- Notice that we can already use this structure for searching: examine the root node and if we have not found what we are looking for:
  - If the object is less than what is stored in the root node, continue searching in the left sub-tree
  - Otherwise, continue searching the right sub-tree

• With a linear order, one of the following three must be true:
  \[ a < b \quad a = b \quad a > b \]

BINARY SEARCH TREES

DEFINITION

• Thus, we define a non-empty binary search tree as a binary tree with the following properties:
  - The left sub-tree (if any) is a binary search tree and all elements are less than the root element,
  - The right sub-tree (if any) is a binary search tree and all elements are greater than the root element
Binary Search Trees

DEGENERATE BINARY TREE

- Unfortunately, it is possible to construct degenerate binary search trees

- This is equivalent to a linked list, i.e., $O(n)$

EXAMPLES

- All these binary search trees store the same data
BST

DUPLICATE ELEMENTS

- We will assume that in any binary tree, we are not storing duplicate elements unless otherwise stated
  - In reality, it is seldom the case where duplicate elements in a container must be stored as separate entities

- You can always consider duplicate elements with modifications to the algorithms we will cover

BST

IMPLEMENTATION

- We will look at an implementation of a binary search tree in the same spirit as we did with our Single_list class
  - We will have a Binary_search_nodes class
  - A Binary_search_tree class will store a pointer to the root

- We will use templates, however, we will require that the class overrides the comparison operators
- Any class which uses this binary-search-tree class must therefore implement:
  
  ```cpp
  bool operator<=( Type const &, Type const & );
  bool operator< ( Type const &, Type const & );
  bool operator==( Type const &, Type const & );
  ```

- That is, we are allowed to compare two instances of this class
- Examples: int and double
#include "Binary_node.h"

template <typename Type>
class Binary_search_tree;

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;

        Type front() const;
        Type back() const;
        bool find( Type const & ) const;

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

    friend class Binary_search_tree<Type>;
};
**BST**

**CONSTRUCTOR**

- The constructor simply calls the constructor of the base class
  - Recall that it sets both left_tree and right_tree to nullptr
  - It assumes that this is a new leaf node

```cpp
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  
}
```

**BST**

**ACCESSORS**

- Because it is a derived class, it already inherits the function:
  ```cpp
  Type retrieve() const;
  ```
- Because the base class returns a pointer to a Binary_node, we must recast them as Binary_search_node:

```cpp
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() );  
}
```
**Binary Search Trees**

**INHERITED MEMBER FUNCTIONS**

- The member functions

  ```
  bool empty() const
  bool is_leaf() const
  int size() const
  int height() const
  ```

- are inherited from the base class `Binary_node`

**Finding the Minimum Object**

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

**FINDING THE MINIMUM OBJECT**

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

- The extreme values are not necessarily leaf nodes

```cpp
    11
   / \  39
  8   29
 / \  / \  \
3  19 24 44
```

**BST**

**FIND**

- The implementation is similar to front and back:

```cpp
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 );
}
```
Recall that a Sorted List is implicitly ordered
- It does not make sense to have member functions such as `push_front` and `push_back`
- Insertion will be performed by a single `insert` member function which places the object into the correct location

An insertion will be performed at a leaf node:
- Any empty node is a possible location for an insertion

The values which may be inserted at any empty node depend on the surrounding nodes

---

For example, this node may hold 48, 49, or 50

An insertion at this location must be 35, 36, 37, or 38
Binary Search Trees

INSERT

- Like find, we will step through the tree
  - If we find the object already in the tree, we will return
    - The object is already in the binary search tree (no duplicates)
  - Otherwise, we will arrive at an empty node
    - The object will be inserted into that location
    - The run time is \( O(h) \)

INSERT

- In inserting the value 52, we traverse the tree until we reach an empty node
  - The left sub-tree of 54 is an empty node

- A new leaf node is created and assigned to the member variable `left_tree`
In inserting 40, we determine the right sub-tree of 39 is an empty node.

A new leaf node storing 40 is created and assigned to the member variable right_tree.

```
BST
INSERT

• In inserting 40, we determine the right sub-tree of 39 is an empty node

• A new leaf node storing 40 is created and assigned to the member variable right_tree

BST
INSERT

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;
    }
}
```
**BST INSERT**

- It is assumed that if neither of the conditions:

  \[
  \text{obj} < \text{retrieve()}
  \]

  \[
  \text{obj} > \text{retrieve()}
  \]

- then \(\text{obj} == \text{retrieve()}\) and therefore we do nothing
  - The object is already in the binary search tree

---

**BST INSERT**

- Blackboard example:
  - In the given order, insert these objects into an initially empty binary search tree:
    
    \[
    31 \ 45 \ 36 \ 14 \ 52 \ 42 \ 6 \ 21 \ 73 \ 47 \ 26 \ 37 \ 33 \ 8
    \]

  - What values could be placed:
    - To the left of 21?
    - To the right of 26?
    - To the left of 47?

  - How would we determine if 40 is in this binary search tree?
  - Which values could be inserted to increase the height of the tree?
A node being erased is not always going to be a leaf node.
There are three possible scenarios:
- The node is a leaf node,
- It has exactly one child, or
- It has two children (it is a full node)

A leaf node simply must be removed and the appropriate member variable of the parent is set to nullptr.
- Consider removing 75

The node is deleted and left_tree of 81 is set to nullptr.
Erasing the node containing 40 is similar:

- The node is deleted and `right_tree` of 39 is set to `nullptr`.

Removing a node with one child:

- If a node has only one child, we can simply promote the sub-tree associated with the child.
  - Consider removing 8 which has one left child.

- The node 8 is deleted and the `left_tree` of 11 is updated to point to 3.
There is no difference in promoting a single node or a sub-tree.

To remove 39, it has a single child 11.

The node containing 39 is deleted and left_node of 42 is updated to point to 11.

Notice that order is still maintained.

Consider erasing the node containing 99.
Finally, we will consider the problem of erasing a full node, e.g., 42.

We will perform two operations:
- Replace 42 with the minimum object in the right sub-tree
- Erase that object from the right sub-tree

In this case, we replace 42 with 47.
- We temporarily have two copies of 47 in the tree.
- We now recursively erase 47 from the right sub-tree.
  - We note that 47 is a leaf node in the right sub-tree.
Leaf nodes are simply removed and `left_tree` of 51 is set to `nullptr`

- Notice that the tree is still sorted:
- 47 was the least object in the right sub-tree

Suppose we want to erase the root 47 again:

- We must copy the minimum of the right sub-tree
- We could promote the maximum object in the left sub-tree and achieve similar results
We copy 51 from the right sub-tree

We must proceed by delete 51 from the right sub-tree

In this case, the node storing 51 has just a single child

We delete the node containing 51 and assign the member variable left_tree of 70 to point to 59
Note that after seven removals, the remaining tree is still correctly sorted.

In the two examples of removing a full node, we promoted:
- A node with no children
- A node with right child
- Is it possible, in removing a full node, to promote a child with two children?
Recall that we promoted the minimum element in the right sub-tree

- If that node had a left sub-tree, that sub-tree would contain a smaller value

In order to properly remove a node, we will have to change the member variable pointing to the node

- To do this, we will pass that member variable by reference

- Additionally: We will return 1 if the object is removed and 1 if the object was not found

```
template<typename Type>
bool Binary_search_node<Type>::erase( Type const &obj, Binary_search_node *&ptr_to_this ) {
    if ( empty() ) {
        return false;
    }
    else if ( obj == retrieve() ) {
        if ( is_leaf() ) {  // leaf node
            ptr_to_this = nullptr;
            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() ) ? left() : right();
            delete this;
        }
        return true;
    } else if ( obj < retrieve() ) {
        return left()->erase( obj, left_tree );
    } else {
        return right()->erase( obj, right_tree );
    }
}
```
**BST**

**IMPLEMENTATION**

```cpp
template <typename Type>
class Binary_search_tree {
    private:
        Binary_search_node<Type> *root_node;
        Binary_search_node<Type> *root() const;
    public:
        Binary_search_tree();
        ~Binary_search_tree();
        bool empty() const;
        int size() const;
        int height() const;
        Type front() const;
        Type back() const;
        int count( Type const &obj ) const;
        void clear();
        bool insert( Type const &obj );
        bool erase( Type const &obj );
};
```

**CONSTRUCTOR, DESTRUCTOR, AND CLEAR**

```cpp
template <typename Type>
Binary_search_tree<Type>::Binary_search_tree():
    root_node( nullptr ) { // does nothing
}

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

template <typename Type>
void Binary_search_tree<Type>::clear() {
    root()->clear( root_node );
}
```
BST
ROOT, EMPTY AND SIZE

template<typename Type>
Binary_search_tree<Type> *Binary_search_tree<Type>::root() const {
    return tree_root;
}

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

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

BST
HEIGHT AND COUNT

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

template<typename Type>
bool Binary_search_tree<Type>::find( Type const &obj ) const {
    return root()->find( obj );
}
// If root() is nullptr, 'front' will throw an underflow exception
template <typename Type>
Type Binary_search_tree<Type>::front() const {
    return root()->front();
}

// If root() is nullptr, 'back' will throw an underflow exception
template <typename Type>
Type Binary_search_tree<Type>::back() const {
    return root()->back();
}

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

template <typename Type>
bool Binary_search_tree<Type>::erase( Type const &obj) {
    return root()->erase( obj, root_node );
}
OTHER RELATION-BASED OPERATIONS

- We will quickly consider two other relation-based queries that are very quick to calculate with an array of sorted objects:
  - Finding the previous and next entries, and
  - Finding the k\textsuperscript{th} entry

PREVIOUS AND NEXT OBJECTS

- All the operations up to now have been operations which work on any container: count, insert, etc.
  - If these are the only relevant operations, use a hash table

- Operations specific to linearly ordered data include:
  - Find the next larger and previous smaller objects of a given object which may or may not be in the container
  - Find the k\textsuperscript{th} entry of the container
  - Iterate through those objects that fall on an interval [a, b]

- We will focus on finding the next largest object
  - The others will follow
PREVIOUS AND NEXT OBJECTS

• To find the next largest object:
  • If the node has a right sub-tree, the minimum object in that sub-tree is the next-largest object.

If, however, there is no right sub-tree:
  • It is the next largest object (if any) that exists in the path from the root to the node.

PREVIOUS AND NEXT OBJECTS

• More generally: what is the next largest entry of an arbitrary object?
  • This can be found with a single search from the root node to one of the leaves—an O(h) operation.
  • This function returns the object if it did not find something greater than it.

```cpp
template<typename Type>
Type Binary_search_node<Type>::next( Type const &obj ) const {
  if ( empty() ) {
    return obj;
  } else if ( retrieve() == obj ) {
    return ( right()EMPTY() ) ? obj : right()->front();
  } else if ( retrieve() > obj ) {
    Type tmp = left()->next( obj );
    return ( tmp == obj ) ? retrieve() : tmp;
  } else {
    return right()->next( obj );
  }
}
```
FINDING THE \( k \)TH OBJECT

Another operation on sorted lists may be finding the \( k \)th largest object
- Recall that \( k \) goes from 0 to \( n - 1 \)
- If the left-sub-tree has \( \ell = k \) entries, return the current node,
- If the left sub-tree has \( \ell < k \) entries, return the \( k \)th entry of the left sub-tree,
- Otherwise, the left sub-tree has \( \ell > k \) entries, so return the \((k - \ell - 1)\)th entry of the right sub-tree

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

```cpp
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);
    }
}
```
FINDING THE Kth OBJECT

- We must now update insert(...) and erase(...) to update it

```cpp
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 ) ? ++tree_size : false;
    }
    else if ( obj > retrieve() ) {
        return right()->insert( obj, right_tree ) ? ++tree_size : false;
    }
    else {
        return false;
    }
}
```

Clever trick: in C and C++, any non-zero value is interpreted as true