Lecture 3

- Binary search trees
- Toward a binary search tree implementation using C++ templates
- C++ iterators and the binary search tree successor function

Reading: Weiss Ch 4, sections 1-4

Binary search trees

A binary search tree is a data structure with these invariants:

- Structural property: a BST is a binary tree
- Ordering property:
  - Each data item in a BST has a *data item*, sometimes called a *key*, associated with it
  - Keys in a BST belong to an ordered set, which means that...

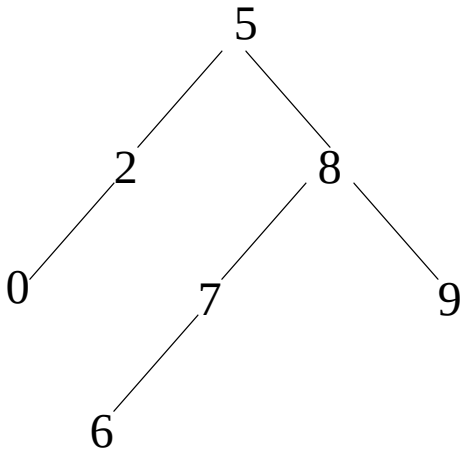
    ... for any two keys k_1 , k_2 exactly one of these is true:

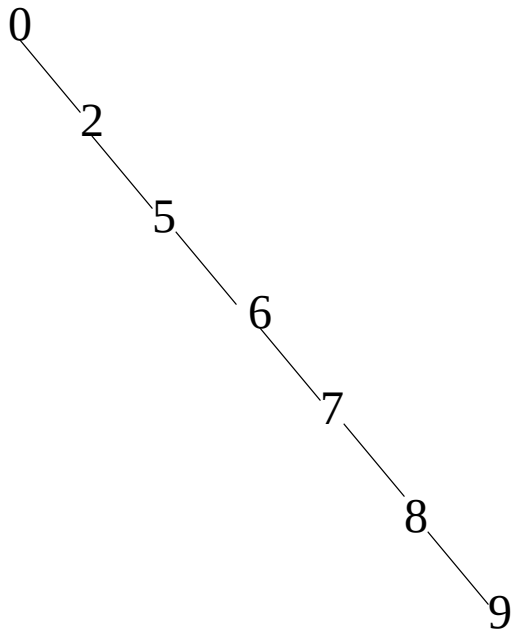
    k_1 is less than k_2 ; k_2 is less than k_1 ; k_1 and k_2 are equal

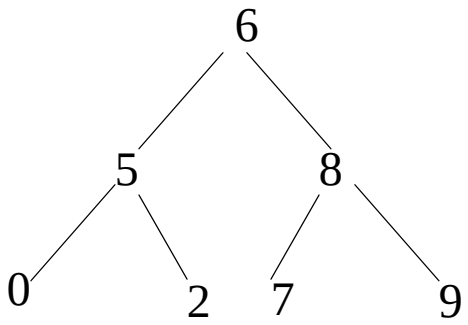
  - Then, for every node X in a BST:
    - the key in X is greater than every key in X's left subtree
    - the key in X is less than every key in X's right subtree
    - (so, a BST does not hold duplicate keys)

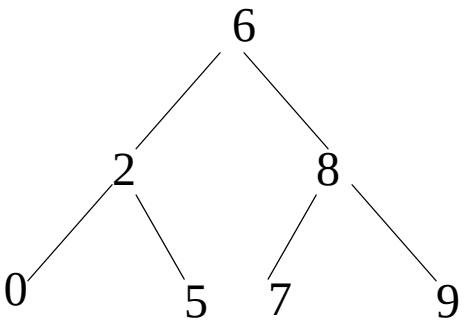
Binary search tree examples

- Which of these are BSTs, and which are not?









Implementing binary search trees

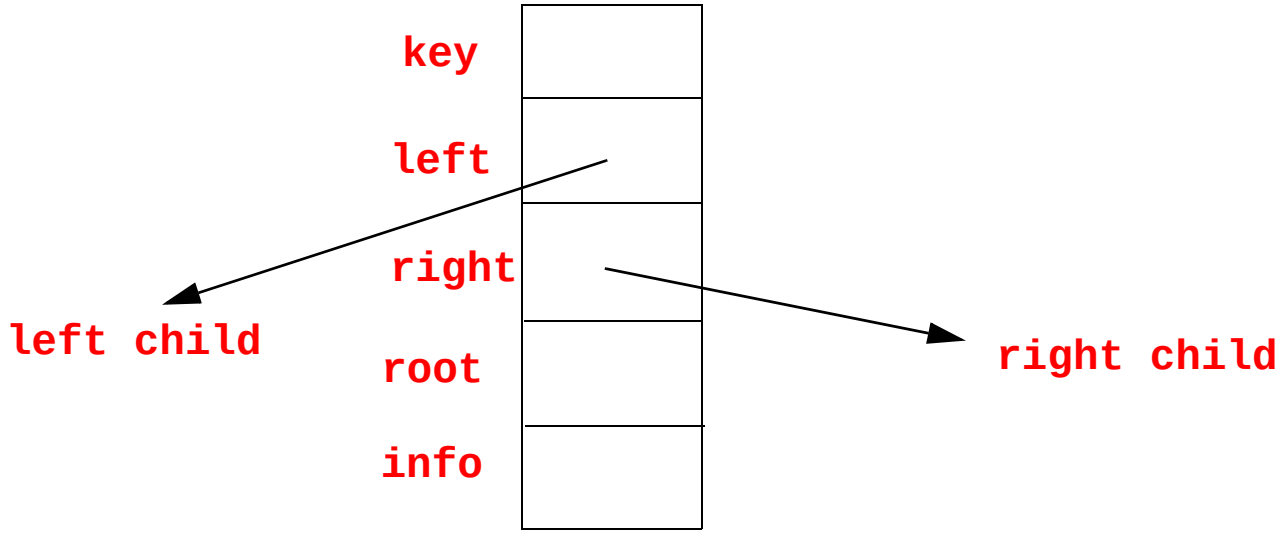
- In an implementation of a BST, nodes should be designed to hold:
  - a pointer to the left child of the node (null if no left child)
  - a pointer to the right child of the node (null if no right child)
  - a pointer to, or copy of, the data item associated with the node

- For some algorithms, it is convenient if the nodes also have:
  - a pointer to the parent (null if this node is the root)
  - a field to hold additional implementation-specific information about the node (balance number, color, priority...)

A Node class template for BST nodes

- Simple C++ Node class template for BST nodes and generic keys:

```
template <typename T>
class Node {
public:
      T key;
      Node<T>* left;
      Node<T>* right;
      Node<T>* parent;
      int info;
};
```



The basic Find operation in a binary search tree

- Idea: exploit the ordering property of BST's; each key comparison either finds the key you are looking for, or tells you which subtree (left or right) to look in next
  - Note: it is a C++ STL convention to use only `<` for key comparisons

- Pseudocode for the basic iterative algorithm to Find key with value `k` in a BST:

1. If `RootNode == NULL`, tree is empty (no root). Return `false`.

2. Set `CurrNode = RootNode`.

3. If `k < CurrNode.key` ... /* key must be in left subtree, if it is in the tree. */
   If `CurrNode.left == NULL`, return `false`.
   else set `CurrNode = CurrNode.left`, and go to 3.

4. else If `CurrNode.key < k` ... /* key must be in right subtree, if it is in the tree . */
   If `CurrNode.right == NULL`, return `false`.
   else set `CurrNode = CurrNode.right`, and go to 3.

5. else Found the key; it is in `CurrNode`. Return `true`.

The Insert operation in a binary search tree

- Again, the idea is to make use of the ordering property of BST's; each key comparison tells you which subtree the key must go in, so the find algorithm can find it later
- But (unlike finds) inserts modify the tree. It is important to maintain all the BST invariants: *If you start with a BST, the result after insertion must still be a BST*!

- Pseudocode for the basic iterative algorithm to Insert key with value `k` in a BST:

1. If `RootNode == NULL`, tree is empty. Set `RootNode = new Node(k)`. Done.

2. Set `CurrNode = RootNode`.

3. If `k < CurrNode.key` ... /* key must go in left subtree */
   If `CurrNode.left == NULL`, set `CurrNode.left = new Node(k)`. Done.
   else set `CurrNode = CurrNode.left`, and go to 3.

4. else If `CurrNode.key < k` ... /* key must go in right subtree. */
   If `CurrNode.right == NULL`, set `CurrNode.right = new Node(k)`. Done.
   else set `CurrNode = CurrNode.right`, and go to 3.

5. else Found the key; Done. /* A BST typically does not contain duplicates */

C++ STL and binary search trees

- The C++ Standard Template Library provides these containers (i.e., data structures):

```
vector
deque
list
stack
queue
priority_queue
set
multiset
map
multimap
bitset
```

- Of these, `set` is one that is implemented using a balanced binary search tree (typically a red-black tree)

- Let's look at some aspects of the interface of set and consider issues in implementing it

C++ STL set find member function

- `set`'s find function has this prototype:

```
template <typename T>

class set {

public:

   iterator find ( T const & x ) const;
```

- Note: find takes an argument that is a reference to a const `T`
  - the actual argument is not copied (it would be copied if the `&` was left out)
  - `x` is a direct reference to the actual argument, not a pointer to it
  - but the `T` that `x` is a reference to must be treated as const by find: it cannot change it
- Note: find is a const member function
  - it is an accessor only
  - it cannot change anything about the set whose find function is being called

C++ STL set find function semantics

- The documentation for set's find function says:

  Searches the container for an element with a value of **x** and returns an iterator to it if found, otherwise it returns an iterator to the element past the end of the container.

- Searching a binary search tree for an element with the value of **x** is easy, as long as you can compare **x** and data items stored in nodes of the tree (using **<**)... just use the standard BST find algorithm

- But then the function is supposed to return an iterator (to the node where **x** is found, or "past the end" of the container if not)

- We will need to study C++ iterators to see how to do that

C++ STL set insert function

- `set`'s insert function has this prototype:

```
template <typename T>

class set {

public:

   std::pair<iterator,bool> insert ( T const & x );
```

- Note: insert takes an argument that is a reference to a const `T`
  - the actual argument is not copied (it would be copied if the `&` was left out)
  - `x` is a direct reference to the actual argument, not a pointer to it
  - but the `T` that `x` is a reference to must be treated as const by find: it cannot change it
- Note: insert is a mutator; it is not a const member function
- Note: insert returns a pair containing an iterator and a bool

C++ STL set insert function semantics

- The documentation for **`set`**'s insert function says:

  The set container is extended by inserting a single new element. This effectively increases the container size by the amount of elements inserted.

  Because set containers do not allow for duplicate values, the insertion operation checks for each element inserted whether another element exists already in the container with the same value, if so, the element is not inserted and an iterator to it is returned.

  The function returns a **`pair`**, with its member **`pair::first`** set to an iterator pointing to either the newly inserted element or to the element that already had its same value in the set. The **`pair::second`** element in the pair is set to true if a new element was inserted or false if an element with the same value existed.

- Inserting a value **`x`** in a BST is easy, as long as you can compare **`x`** (using **`<`**) to data items stored in nodes of the tree... just use the standard BST insert algorithm

- But then again the function is supposed to return an iterator, as an element of a **`std::pair`**

C++ STL iterators

- In the *iterator pattern* of OO design, a container has a way to supply to a client an iterator object which is to be used by the client to access the data in the container sequentially, without exposing the container's underlying representation

- Containers in the STL implement the iterator pattern
- For example here's a typical way client code can iterate over (and print out) all of the data in an STL container, in this case a **set**:

```
set<string> c;
...
// get an iterator pointing to container's first element
set<string>::iterator itr = c.begin();
// get an iterator pointing past container's last element
set<string>::iterator end = c.end();
// loop while itr is not past the last element
while(itr != end) {
   cout << *itr << endl;  // dereference the itr to get data
   ++itr;                 // increment itr to point to next element
}
```

Implementing the iterator pattern in C++

- To implement the iterator pattern for a container in STL fashion, we need to consider:
- How to define the **`begin()`** member function of the container
  - this function must return an iterator object “pointing to” the first element of the container
- How to define the **`end()`** member function of the container
  - this function must return an iterator object “pointing just past” the last element of the container
- How to define these operators for iterators:
  - **`!=`** (not equal test). This operator must return true (1) or false (0) according to whether the two iterators are pointing to the same element or not
  - **`*`**(dereference). This operator must return a reference to (or possibly just a copy of) the data item contained in the element the iterator is currently pointing to
  - **`++`**(pre-increment). This operator must cause the iterator to point to the next element of the container
  - (Note: some iterators also allow decrement **`--`**, and some allow arbitrary pointer arithmetic)
- We’ll consider the iterator operators first

An iterator class template for a BST

- Suppose a BST's nodes are instances of a class template Node as shown before
- At each step in an iteration, an iterator for the BST only needs to keep a pointer to the current Node in the BST
- So, define a BSTIterator class template with one member variable that is a pointer to the current node; then a constructor and overloaded operators for the class are easy to define:

```
template <typename T>
class BSTIterator {

private:
   Node<T>* curr;

public:
   /** Constructor */
   BSTIterator(Node<T>* n) : curr(n) {}
```

An iterator class template for a BST, cont'd

```
public:

   /** Inequality test operator */
   bool operator!=(BSTIterator<T> const & other) const {
      return this->curr != other.curr;
   }

   /** Dereference operator */
   T operator*() const {
      return curr->data;
   }

   /** Pre-increment operator */
   BSTIterator<T>& operator++() {
      curr = curr->successor();  // point to next node
      return *this;
   }
};
```

- How to define the successor() member function of Node?...

The successor relation in binary search trees

- Inorder traversal of a binary tree can be defined recursively:

```
template <typename T>
void inorder(Node<T>* n) {
   if(0 == n) return;
   inorder(n->left);
   visit(n);
   inorder(n->right);
}
```

- An inorder traversal of a binary search tree starting at the root will visit its nodes in sorted order, according to the ordering relation on the data contained in the nodes

- An iterator for a binary search tree backed container should iterate over the container's data in sorted order
  - That is, the iterator should essentially simulate an inorder traversal, one step at a time

- If the iterator is currently pointing to a node X in a BST, when it is incremented it should then point to X's successor, that is, the node that would come immediately after X during an inorder traversal of the BST (if any)

Finding the successor of a BSTNode

- Suppose X is a node in a BST that has just been visited during an inorder traversal. What is the next node that will be visited; that is, what is the successor of X?
- Consider cases:
  - X has a right child
  - X has no right child, and is the left child of its parent
  - X has no right child, and is the right child of its parent
  - X has no right child, and has no parent

A container's end() function

- How to define the **`end()`** member function of the container
  - this function must return an iterator object "pointing just past" the last element of the container
  - For a BST container, with our partial definition of the **`BSTIterator`** template, this function can just return an iterator initialized with the null pointer **`0`**: that will be enough to distinguish it from any iterator that is "pointing to" any actual element

```
template <typename T>
class BST {
private:
   Node<T>* root;

public:
   typedef BSTIterator<T> iterator;

   iterator end() const {
      return BSTIterator<T>(0);
   }
```

A container's begin() function

- How to define the **`begin()`** member function of the container?
  - this function must return an iterator object "pointing to" the first element of the container
  - with our partial definition of the **`BSTIterator`** template, this function can just return an iterator initialized with the **`Node`** holding the first (i.e. smallest) data item in the BST

```
iterator begin() const {
   return BSTIterator<T>( first(root) );
}
```

- Now... how to define the **`first()`** function?
- (It is just a helper function, not part of the interface to the BST class, so it should be private...)
- (And if we define it to take a **`Node`** pointer as argument, it can be a static function; the function will not need to refer to "this" BST...)

Defining a first() function

```
private:

   /** Return a pointer to the node containing the smallest data
    *  item in the BST subtree rooted at n, or 0 if n is 0
    */
   static Node<T>* first(Node<T>* n) {









   }
```

Next time

- Binary search tree average cost analysis
- The importance of being balanced
- AVL trees and AVL rotations
- Insert in AVL trees

Reading: Weiss Ch 4, sections 1-4