CSE100

Advanced Data Structures

Lecture 4

(Based on Paul Kube course materials)
Lecture 4

- Binary search trees
- Toward a binary search tree implementation using C++ templates

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 node 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 reference 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:

```cpp
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; if that subtree is empty, the key is not in the tree.
  - 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 \( \text{RootNode} == \text{nullptr} \), tree is empty (no root). Return \text{false}.
  2. Set \( \text{CurrNode} = \text{RootNode} \).
  3. If \( k < \text{CurrNode.key} \) ... /* key must be in left subtree, if it is in the tree. */
    - If \( \text{CurrNode.left} == \text{nullptr} \), return \text{false}.
    - else set \( \text{CurrNode} = \text{CurrNode.left} \), and go to 3.
  4. else If \( \text{CurrNode.key} < k \) ... /* key must be in right subtree, if it is in the tree. */
    - If \( \text{CurrNode.right} == \text{nullptr} \), return \text{false}.
    - else set \( \text{CurrNode} = \text{CurrNode.right} \), and go to 3.
  5. else Found the key; it is in \( \text{CurrNode} \). Return \text{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 \( \text{RootNode} == \text{nullptr} \), tree is empty. Set \( \text{RootNode} = \text{new Node}(k) \). Done.
2. Set \( \text{CurrNode} = \text{RootNode} \).
3. If \( k < \text{CurrNode.key} \) /* key must go in left subtree */
   - If \( \text{CurrNode.left} == \text{nullptr} \), set \( \text{CurrNode.left} = \text{new Node}(k) \). Done.
   - else set \( \text{CurrNode} = \text{CurrNode.left} \), and go to 3.
4. else If \( \text{CurrNode.key} < k \) /* key must go in right subtree */
   - If \( \text{CurrNode.right} == \text{nullptr} \), set \( \text{CurrNode.right} = \text{new Node}(k) \). Done.
   - else set \( \text{CurrNode} = \text{CurrNode.right} \), 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):
  ```
  array
  vector
  deque
  forward_list
  list
  stack
  queue
  priority_queue
  set
  multiset
  unordered_set
  map
  unordered_map
  multimap
  bitset
  ```

- Of these, `set` and `map` are 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:

  ```cpp
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::find` 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:

```cpp
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::insert` 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

• Again, the function is supposed to return an iterator, as an element of a `std::pair`
std::pair

• Definition of the pair struct template:

```cpp
template <class T1, class T2> struct pair {
    T1 first;
    T2 second;
    pair(const T1& x, const T2& y) : first(x), second(y) {}
};
```

• (Note: in C++, a struct is exactly the same as a class, except that members are public by default; in a class, members are private by default)

• Example of using pair, and of using the function template make_pair:

```cpp
#include <utility>
#include <string>
#include <iostream>

int main () {
    std::pair<std::string, double> pr ("tomatoes", 3.25);
    std::cout << pr.first << ', ' << pr.second << std::endl;
    pr.second = 4.50;
    std::cout << pr.first << ', ' << pr.second << std::endl;
    pr = make_pair("oranges", 2.75);
    std::cout << pr.first << ', ' << pr.second << std::endl;
}
```
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`:

```cpp
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
}
```
Next time

- C++ iterators
- The binary search tree successor function

Reading: Weiss Ch 4, sections 1-4