Algorithms in Systems Engineering
IE170

Lecture 9

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - CLRS Chapter 12
- References
  - D.E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching* (Third Edition), 1998.
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Symbol Tables and Dictionaries

- In the last few lectures, we discussed various methods for sorting a list of items by a specified key.
- We now consider further operations on such lists.
- A *symbol table* is a data structure for storing a list of items, each with a key and satellite data, supporting the following basic operations.
  - Construct a symbol table.
  - Search for an item (or items) having a specified key.
  - Insert an item.
  - Remove a specified item.
  - Count the number of items.
  - Print the list of items.
- Symbol tables are also called *dictionaries* because of the obvious comparison with looking up entries in a dictionary.
- Note that the keys may not have an ordering.

Additional Operations on Symbol Tables

- If the items can be ordered, e.g., by `operator<` and `operator ==`, we may support the following additional operations.
  - Sort the items (print them in sorted order).
  - Return the maximum or minimum item.
  - Select the k^{th} item.
  - Return the successor or predecessor of a given item.
- We may also want to be able to join two symbol tables into one.
- These operations may or may not be supported in various implementations.

Applications of Symbol Tables

- What are some applications of symbol tables?

Symbol Tables with Integer Keys

- Consider a list of items whose keys are small positive integers.
- Assuming no duplicate keys, we can implement such a symbol table using an array.

```
class sybmolTable
{
   private:
      symbolTable(); \\ Disable the default constructor
      Item** st_; \\ An array of pointers to the items
      const int maxKey_; \\ The maximum allowed value of a key
   public:
      symbolTable (const int M); \\ Constructor
      ~symbolTable (); \\ Destructor
      int getNumItems() const;
      Item* search (const int k) const;
      Item* select (int k) const;
      void insert (Item* it);
      void remove (Item* it);
      void sort (ostream& os);
}
```

Implementation

```
symbolTable::symbolTable (const int M)
{
   maxKey_ = M;
   st_ = new Item* [M];
   for (int i = 0; i < M; i++) { st_[i] = 0; }
}

void symbolTable::insert(Item* it)
{ st_[it.getKey()] = it; }

void symbolTable::remove(Item* it)
{ delete st_[it.getKey()]; st_[it.getKey()] = 0; }

Item* symbolTable::search(const int k) const
{ return st_[k]; }
```

Implementation (cont.)

```
Item* select(int k)
{
   for (int i = 0; i < maxKey_; i++)
      if (st_[i])
         if (k-- == 0) return st_[i];
}

Item sort(ostream& os)
{
   for (int i = 0; i < maxKey_; i++)
      if (st_[i])
         os << *st_[i];
}

int getNumItems() const
{
   int j(0);
   for (int i = 0; i<maxKey_; i++) { if (st_[i]) j++; }
   return j;
}
```

Arbitrary Keys

- Note that with arrays, most operations are constant time.
- What if the keys are not integers or have arbitrary value?
- We could still use an array or a linear linked list to store the items.
- However, some of the operations would become inefficient.
- Recall Lab 1
  - If we keep the items in order, searching would be efficient (binary search), but inserting would be inefficient.
  - If we kept the items unordered, inserting would be efficient, but searching would be inefficient (sequential search).
- A *binary search tree* (BST) is a more efficient data structure for implementing symbol tables where the keys are an arbitrary data type.

Binary Search Trees

- To use the BST data structure, the keys must have an order.
- As with heaps, a binary search tree is a binary tree with additional structure.
- In a binary tree, the key value of any node is
  - greater than or equal to the key value of all nodes in its *left subtree*;
  - less than or equal to the key value of all nodes in its *right subtree*.
- For now, we will assume that all keys are unique.
- With this simple structure, we can implement all functions efficiently.

Searching

- Search in a BST can be implemented recursively in a fashion similar to binary search, starting with the root as the current node.
  - If the pointer to the current node is 0, then return 0.
  - Otherwise, compare the search key to the current node's key, if it exists.
  - If the keys are equal, then return a pointer to the current node.
  - If the search key is smaller, recursively search in the left subtree.
  - If the search key is larger, recursively search in the right subtree.
- What is the running time of this operation?

Inserting a Node

- The procedure for inserting a node is similar to that for searching.
- As before, we will assume there is no item with an identical key already in the tree.
- We simply perform an unsuccessful search and insert the node in place of the final 0 pointer at the end of the search path.
- This places it where we would expect to find it the next time we look.
- The running time is the same as searching.
- Constructing a BST from a given list of elements can be done by iteratively inserting each element.

Finding the Minimum and Maximum

- Finding the minimum and maximum is a simple procedure.
- The minimum is the leftmost node in the tree.
- The maximum is the rightmost node in the tree.

Sorting

- We can easily read off the items from a BST in sorted order.
- This involves *walking the tree* in a specified way.
- Walking the tree is done recursively by first walking the left subtree and then the right subtree.
- This leads to three different orders in which we can display the key values in the tree.
  - To display the values in *preorder*, print the value of the current node *before* recursively walking the two subtrees.
  - To display the values in *inorder*, print the value of the current node *after* walking the left subtree, but *before* walking the right subtree.
  - To display the values in *postorder*, print the value of the current node *after* walking both subtrees.
- Which display order will result in the printing of a sorted list?

Finding the Predecessor and Successor

- To find the successor of a node x, think of an inorder tree walk.
- After visiting a given node, what is the next value to get printed out?
- We need to examine two cases.
  - If x has a right child, then the successor is the node with the minimum key in the right subtree (easy to find).
  - Otherwise, the successor is the lowest ancestor of x whose left child is also an ancestor of x (why?).
  - To find such a node, we follow the path to the root until we reach a node that is the left child of its parent.
  - Note that if a node has two children, its successor cannot have a left child (why not?).
- Finding the predecessor works the same way.

Deleting a Node

- Deleting a node z from a BST is more complicated than other operations because of the rigid structure that must be maintained.
- There are a number of algorithms for doing this.
- The most straightforward implementation considers three cases.
  - If z has no children, then simply set the pointer to z in the parent to be 0.
  - If z has one child, then replace z with its child.
  - If z has two children, then delete either the predecessor or the successor and then replace z with it.
- Why does this work?

Performance of BSTs

- Efficiency of the basic operations depends on the depth of the tree.
- Consider the search operation: what is the best case?
- The best case is to make the same comparisons as in binary search.
- However, this can only happen if the root of each subtree is the median element of that subtree, i.e., the tree is balanced.
- Fortunately, if keys are added at random, this should be the case "on average."
  - Like quicksort, the average performance is very good, but worst case behavior is easy to find (where?).
  - In fact, quicksort and BSTs exhibit worst case behavior on the same inputs!
  - As with quicksort, one can show that for a random sequence of keys, the average depth of the tree is $2 \ln n \approx 1.39 \lg n$.
  - Again, the average depth is only 40% higher than the best possible.
  - Building a binary search tree has the same running time as quicksort!

Handling Duplicate Keys

- What happens when the tree may contain duplicate keys?
- To make things easier, we can always insert items with duplicate keys in the right subtree.
- To find all items with the same key, search for the first item and then recursively search for the same item in the right subtree.
- Alternatively, we could maintain a linked list of items with the same key at each node in the tree.