Computational Optimization
ISE 407

Lecture 18

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Sections 6.5-6.7
- References
  - CLRS Chapter 22
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Priority Queues

- A priority queue is a data structure for maintaining a list of items that have associated *priorities*.
- The usual operations are
  - construct a queue from a list of items.
  - find the item with the highest priority.
  - insert an item.
  - delete an item.
  - change the priority of an item.
- Any implementation of a priority queue can be used to sort a list of items.
  - Put the items in a priority queue.
  - Delete the maximum item n times.

Heaps

- A *heap* is a balanced binary tree with additional structure that allows it to function efficiently as a priority queue.
- The additional structure needed to support these operations is that the record stored at each node has a higher priority than either of its children.
- Any node with this property is said to satisfy the *heap property*.
- Consider a tree in which all nodes except for the root have the heap property.
- We can easily transform this into a tree in which every node has the heap property (how?).
- This operation is called `heapify()`.
- By calling `heapify()` on each node, starting at the lowest level and working upward, we can transform an unordered binary tree into a heap.

Operations on a Heap

- The node with the highest priority is always the root.
- To delete a record
  - Exchange its record with that of a leaf.
  - Delete the leaf.
  - Call `heapify()`.
- To add a record
  - Create a new leaf.
  - Exchange the new record with that of the parent node if it has a higher priority.
  - Continue to do this until all nodes have the heap property.
- Note that we can change the priority of a record in a similar fashion.

Heap Sort

- Suppose the list of items to be sorted are in an array of size n.
- The heap sort algorithm is as follows.
  - Put the array in heap order as described above.
  - In the i^{th} iteration, exchange the item in position 0 with the item in position $n - i$ and call `heapify()`.
- Why is this algorithm correct?
- How do we analyze the running time?

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 to support more operations (at a cost).
  - Listing the items in sorted order
  - Finding the k^{th} items in sorted order
- In a binary tree, the key value of any node is
  - less than or equal to the key value of all nodes in its *left subtree*;
  - greater 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 `None`, then return `None`.
  - 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 `None` 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.
- Recall that 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 `None`.
  - 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.

Selection

- Recall that the selection problem is that of finding the k^{th} element in an ordered list.
- Selection can be done using an algorithm similar to the quicksort algorithm (notice the connection again).
- However, we need an additional data member `count` in the node class that tracks the size of the subtree rooted at each node.
- With this additional data member, we can recursively search for the k^{th} element.
  - Starting at the root, if the size of the left subtree is $k-1$, return a pointer to the root.
  - If the size of the left subtree is more than $k-1$, recursively search for the k^{th} element of the left subtree.
  - Otherwise, recursively search for the $(k-t-1)^{\text{th}}$ element of the right subtree, where t is the size of the left subtree.
- Note that maintaining the `count` data member can be expensive.

Rotation and Balancing

- To guard against poor performance, we would like to have a scheme for keeping the tree balanced.
- There are many schemes for automatically maintaining balance.
- We describe here a method of manually rebalancing the tree.
- The basic operation that we'll need is that of *rotation*.
- Rotating the tree means changing the root from the current root to one of its children, while maintaining the BST structure.
- To change the right child of the current root into the new root.
  - Make the current root the left child of the new root.
  - Make the left child of the new root the right child of the old root.
- Note that we can make any node the root of the BST through a sequence of rotations.

Partitioning and Rebalancing

- To partition the list around the k^{th} item, select the k^{th} item and rotate it to the root.
- This can be implemented easily in a recursive fashion.
- The left and right subtrees form the desired partition.
- To (re)balance a BST
  - Partition around the middle node.
  - Recursively balance the left and right subtrees.
- This operation can be called periodically
- What is the running time of this operation?

Another Implementation of Delete

- Using the `partition` operation, we can implement delete in a slightly different way.
  - Partition the right subtree of the node to be deleted around its smallest element x.
  - Make the root of the left subtree the left child of x.

Root Insertion and Joining

- Often it is useful to be able to insert a node as the root of the BST.
- This can be done easily by inserting it as usual and then rotating it to the root, i.e., partition around it.
- With root insertion, we can define a recursive method to `join` two BSTs.
  - Insert the root of the first tree as the root of the second.
  - Recursively `join` the pairs of left and right subtrees.

Randomized BSTs

- Recall that we used randomization to guard against the worst case behavior of quicksort.
- We can do the same here.
- The procedure for randomly inserting into a BST of size n is as follows.
  - With probability $1/(n+1)$, perform root insertion.
  - Otherwise, recursively insert into the right or left subtree, as appropriate, using the same method.
- One can prove mathematically that this is the same as randomly ordering the elements first and then inserting them as usual.
- Hence, this should guard against common worst-case inputs.