IE170: Algorithms in Systems Engineering: Lecture 8

Jeff Linderoth

Department of Industrial and Systems Engineering
Lehigh University

January 31, 2007

Taking Stock

Last Time

- Hashes
- (Intro to) Binary Search Trees
- More on Java Collections Interfaces
- `lab == fun`

This Time

- Binary Search Trees
- Java Collections Interfaces: Maps
- Heaps and Heapsort

Binary Search Tree

- A binary search tree is a data structrue that is conceptualized as a binary tree. (Have you read Appendix B-4 yet?)
- Each node in the tree contains:
  - key k. (Or maybe (key, value): (k, v))
  - left l: Points to the left child
  - right r: Points to the right child
  - parent p: Points to the parent

Binary Search Tree Property

If y is in the left subtree of x, then $k(y) \leq k(x)$

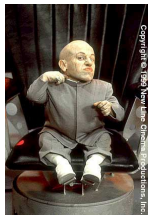
Binary Search Trees

- There are lots of binary trees that can satisfy this property.
- It is obvious that the number of binary tree on n nodes b_n is

$$b_n = \frac{1}{n+1}\begin{pmatrix} 2n \\ n \end{pmatrix} \quad b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}}(1 + O(1/n))$$

- And not all of these (exponentially many) are created equal.
- In fact, we would like to keep our binary search trees "short", because most of the operations we would like to support are a function of the height h of the tree.

Short Is Beautiful



- SEARCH() takes $O(h)$
- MINIMUM(), MAXIMUM() also take $O(h)$
- Slightly less obvious is that INSERT(), DELETE() also take $O(h)$
- Thus we would like to keep out binary search trees "short" (h is small).
- SUCCESSOR(x):
  - If right subtree is not empty: MINIMUM($r(x)$)
  - If right subtree is empty: Walk up tree until you make the first "right" move

Sorted

- We saw in the lab that the Java Tree Set allowed you to iterate through the list in sorted order. How long does it take to do this?

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2    then INORDER-TREE-WALK(ℓ(x))
3         print k(x)
4         INORDER-TREE-WALK(r(x))
```

- What is running time of this algorithm?

Who's Got Next?

- If you would like to find the next largest key value, what do you do?

Basic Idea

- If node x has a right child $r(x)$, then return MINIMUM($r(x)$).
- Otherwise, start walking up until you make the first "right" move.

- Notes that "get next smallest" behaves similarly.

Insert and Delete

- Inserting an element is fairly straightforward: Walk to the bottom, and put it in.
- Deleting an element is harder, but made easier by the following fact:

Binary Search Tree Fact

If a node has two children, then its successor (next largest) has no left child. Its predecessor (next smallest) has no right child

- So to delete, splice a node x with ket $k(x)$, splice out its success and put it in node $x's$ place.

red-black Trees

- red-black trees are simply a way to keep binary search trees short. (Or balanced)
- Balanced here means that no path on the tree is more than twice as long as another path.
- An implication of this is that its maximum height is $2\lg(n+1)$
- SEARCH(), MINIMUM(), MAXIMUM(), all take $O(\lg n)$
- It's implementation is complicated, so we won't cover it
- INSERT(): also runs in $O\lg(n)$
- DELETE(): runs in $O\lg(n)$
  - (but it is more complicated to maintain the "red-black" property)

Map

- A Map is an object that maps keys to values.
- A map cannot contain duplicate keys
- It models the mathematical function abstraction.
- It is like a "set", but each element now holds a (key, value) pair.

Map Implementations

- HashMap
- TreeMap
- LinkedHashMap

Map

```
public interface Map<K,V> {

    // Basic operations
    V put(K key, V value);
    V get(Object key);
    V remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);

    // Collection Views
    public Set<K> keySet();
    public Collection<V> values();
    public Set<Map.Entry<K,V>> entrySet();

    // Interface for entrySet elements
    public interface Entry {
        K getKey();
        V getValue();
        V setValue(V value);
```

Some Sample Java Code

```
public class Freq {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<String, Integer>();
         // Initialize frequency table from command line
        for (String a : args) {
            Integer freq = m.get(a);
            m.put(a, (freq == null) ? 1 : freq + 1);
        }
        System.out.println(m.size() + " distinct words:");
        System.out.println(m);
    }
}
```

- put line is a bit tricky here: has the effect of setting the frequency to one if the word has never been seen before or one more than its current value if the word has already been seen.

Iterating on Maps

```
    for (KeyType key : m.keySet())
        System.out.println(key);

// Filter a map based on some property of its keys.
for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )
    if (it.next().isBogus())
        it.remove();
```

- To iterate over the pairs – you get the entryset:

```
    for (Map.Entry<KeyType, ValType> e : m.entrySet())
        System.out.println(e.getKey() + ": " + e.getValue());
```

Heaps

- A heap is a balanced binary tree with additional structure that allows it to function efficiently as a priority queue.
- There are two types of heaps: $\max$ and $\min$. In lecture, I'll stick to $\max$

Priority Queue (Max)

- INSERT(x)
- MAXIMUM$()$
- $x =$ EXTRACT-MAX$()$
- INCREASE-KEY$(x,\ k)$

Heaps

- Heaps are a bit like binary search trees, however, they enforce a different property

Heap Property: Children are Horrible!

- In a max-heap, the key of the parent node is always at least as big as its children:

$$k(p(x)) \geq k(x) \quad \forall x \neq root$$

- Children are great in min-heaps

How to Keep the Heap Property?

- Consider a tree in which all nodes except for one have the heap property.
- We can transform this into a tree in which every node has the heap property.
- This operation is called HEAPIFY$()$.

Heapify

HEAPIFY(x)

1. Find largest of $k(x)$, $k(\ell(x))$, $k(r(x))$
2. If $k(x)$ is largest, you are done
3. Swap x with largest node, and call HEAPIFY() on the new subtree

- Intuition behind analysis: Heap is binary tree, so $\leq \lg n$ levels. There is a constant amount of work at each level: comparing three items and swapping two.
- $\Rightarrow$ HEAPIFY a node in $O(\lg n)$
- Alternatively, HEAPIFY node of height h is $O(h)$

Next Time?

- Finish Heaps and show how to sort with heaps: Heapsort
- We have covered chapters 1-4, 6, 10-11, and Appendices A and B: That's a lot!

News

- Homework due 2/5 – No late homework – We do review on 2/5
- Quiz on 2/7