CS 361, Lecture 21

Jared Saia
University of New Mexico

Outline

- Class Evaluation
- Binary Trees

Evaluation Results

- Vast majority of students said class pace is "just right", so pace will stay the same as it is now
- Major other problem is "hw is too difficult"

HW Difficulty

- The HW in this class is inherently difficult, this is a difficult class.
- You need to be able to solve problems as hard as the problems in the book to be competitive with students from other schools
- Some of these problems require deep thinking
- However there are things we can do to make things easier

Things you can do

- **Start the hws early!!**
- You have three resources you can use to do well on the hws:
  - Other students - use email,class list, or phone
  - Lab Sections - bring specific questions to lab section
  - Office Hours - come to these

Things I will do

- Answer any HW questions at the beginning of class
- Answer any HW questions emailed to the class mailing list
- Note: You need to start hw early in order to be able to ask me questions about problems you are having

HW Questions

- Are there any questions on the current HW?

Binary Search Trees

- Q: What is a search tree?
- A1: It's yet another data structure for implementing the dictionary ADT
- Q: Don't we already know enough of those?
- A: No

Hash Tables

Hash Tables implement the Dictionary ADT, namely:

- Insert(x) - $O(1)$ expected time, $\Theta(n)$ worst case
- Lookup(x) - $O(1)$ expected time, $\Theta(n)$ worst case
- Delete(x) - $O(1)$ expected time, $\Theta(n)$ worst case

Red-Black Trees

Red-Black trees (a kind of binary tree) also implement the Dictionary ADT, namely:

- Insert(x) - $O(\log n)$ time
- Lookup(x) - $O(\log n)$ time
- Delete(x) - $O(\log n)$ time

Why BST?

- Q: When would you use a Search Tree?
- A1: When need a hard guarantee on the worst case run times (e.g. "mission critical" code)
- A2: When want something more dynamic than a hash table (e.g. don't want to have to enlarge a hash table when the load factor gets too large)
- A3: Search trees can implement some other important operations...

Search Tree Operations

- Insert
- Lookup
- Delete
- *Minimum/Maximum*
- *Predecessor/Successor*

What is a BST?

- It's a binary tree
- Each node holds a key and record field, and a pointer to left and right children
- *Binary Search Tree Property* is maintained

Binary Search Tree Property

- Let x be a node in a binary search tree. If y is a node in the left subtree of x, then key(y)≤key(x). If y is a node in the right subtree of x then key(x)≤key(y)

Example BST

Inorder Walk

- BSTs are arranged in such a way that we can print out the elements in sorted order in $\Theta(n)$ time
- Inorder Tree-Walk does this

Inorder Tree-Walk

```
Inorder-TW(x){
  if (x is not nil){
    Inorder-TW(left(x));
    print key(x);
    Inorder-TW(right(x));
}
```


Example Tree-Walk

Analysis

- Correctness?
- Run time?

Search in BT

```
Tree-Search(x,k){
  if (x=nil) or (k = key(x)){
    return x;
  }
  if (k<key(x)){
    return Tree-Search(left(x),k);
  }else{
    return Tree-Search(right(x),k);
  }
}
```


Analysis

- Let h be the height of the tree
- The run time is $O(h)$
- Correctness???

In-Class Exercise

- Q1: What is the loop invariant for Tree-Search?
- Q2: What is Initialization?
- Q3: Maintenance?
- Q4: Termination?

Tree Min/Max

- Tree Minimum(x): Return the leftmost child in the tree rooted at x
- Tree Maximum(x): Return the rightmost child in the tree rooted at x

Tree-Successor

```
Tree-Successor(x){
  if (right(x) != null){
    return Tree-Minimum(right(x));
  }
  y = parent(x);
  while (y!=null and x=right(y)){
    x = y;
    y = parent(y);
  }
  return y;
}
```

Successor Intuition

- Case 1: If right subtree of x is non-empty, successor(x) is just the leftmost node in the right subtree
- Case 2: If the right subtree of x is empty and x has a successor, then successor(x) is the lowest ancestor of x whose left child is also an ancestor of x.

Insertion

Insert(T,x)

1. Let r be the root of T.
2. Do Tree-Search(r,key(x)) and let p be the last node processed in that search
3. If p is nil (there is no tree), make x the root of a new tree
4. Else if key(x) $\leq$ p, make x the left child of p, else make x the right child of p

Deletion

- Code is in book, basically there are three cases, two are easy and one is tricky
- Case 1: The node to delete has no children. Then we just delete the node
- Case 2: The node to delete has one child. Then we delete the node and "splice" together the two resulting trees

Case 3

Case 3: The node, x to be deleted has two children

1. Swap x with Successor(x) (Successor(x) has no more than 1 child (why?))
2. Remove x, using the procedure for case 1 or case 2.

Analysis

- All of these operations take $O(h)$ time where h is the height of the tree
- If n is the number of nodes in the tree, in the worst case, h is $O(n)$
- However, if we can keep the tree *balanced*, we can ensure that $h = O(\log n)$
- Next time, we'll see how Red-Black trees can maintain a balanced BST