Lecture 4: Overview

- Binary Search Trees (Chapter 12 & 13)
- Programming Assignment 1 due next week
Binary Search Trees

- Store data in a tree
- Each node can have up to two child nodes
  - one left child
  - one right child
Binary Search Trees

- All left descendants have smaller values
- All right descendants have larger values
- Nodes w/o descendants are called leaf node
Binary Search Trees

1. To find a value in a tree start at the root
2. If the value is less than the current node look at its left child
3. If the value is greater, look at its right child
Binary Search Trees

1. To find a value in a tree start at the root
2. If the value is less than the current node look at its left child
3. If the value is greater, look at its right child
4. Repeat 2 until either we find the node or we reach the bottom of the tree
5. If we reach the bottom of the tree, it doesn’t contain the value
Searching Tree

- Search for 5
- Start at root
Searching Tree

- Since 5 > 4 look at right child
Searching Tree

- Since 5 < 6 look at left child
Searching Tree

- Found value!!!
In-order Traversal

- In addition to pre-order and post-order traversal, we have an in-order traversal
- Algorithm:
  1. Visit left child
  2. Process/print current node
  3. Visit right child
- Visits nodes in sorted order
Pre/post-order traversals

- Pre-order
  - current node
  - recursive left
  - recursive right
- Post-order
  - recursive left
  - recursive right
  - current node
Tree-minimum/maximum

- **Minimum**
  
  while left[x]! = null
  
  do x <- left[x]

  return x

- **Maximum**

  while right[x]! = null

  do x <- right[x]

  return x

- **Time complexity: O(h)**
Successor/predecessor

Consider in-order traversal
Want next node
1. If right[x]!=NIL
2. return minimum(right[x])
3. y<-p[x]
4. while y!=NULL and x==right[y]
5. do x<-y
6. y<-p[y]
7. return y

Predecessor - mirror of algorithm
Inserting a Node into a Tree

• Find where to place node:
  Use same algorithm as search algorithm
• Two possible outcomes:
  • Find value – then simply do a replacement
  • End up at node where we can’t search any further:
    • If less than leaf node add to left side
    • If greater than leaf node add to right side
Removing a Node from the Tree

• Three cases:
  • Leaf node - just remove node
  • Single child - replace with child node
  • Two children - copy value with next value in an in-order traversal (will have at most one child, so can use previous two cases on it)
Time Complexity

- Depends on the depth of the tree
- If the tree is balanced, depth $\propto \log(N)$
- If tree isn’t balanced, depth can grow as $N$
Order Preserving Rotations

Left rotation

Right rotation
Red-Black Tree

- Every node is either red or black
- The root is black
- (conceptual) Leaf is black - all nodes have two children, but one may be a null pointer
- A red node has two black children
- All simple paths from a node to a descendant leaf have the same number of black nodes
Red Black Tree

- Definition: Black Height $bh(x)$
  - # of black nodes on a path to a leaf node (not including $x$ itself)
- Height $h$-node $x$ has black height $bh(x) \geq h/2$
  - Can’t have more than 2 red nodes in a row
- Number of vertices $n$ is bound by height of tree
  - upper bound: full tree w/ red/blacks
    - $n \leq 2^h - 1$
  - lower bound: black only $n \geq 2^{bh(x)} - 1$
    - $n \geq 2^{h/2} - 1$
    - $h \leq 2 \log(n+1)$
Insertion

• Same insertion as before, assign new node the color red
• May need to fix up things...
• If the parent of the new node is black, everything is still fine (we haven’t violated any properties)
Fixup

RB-Insert-Fixup(T,z)
1 while color[p[z]] = RED
2   do if p[z] = left[p[p[z]]]
3     then y<-right[p[p[z]]]
4       if color[y]=RED
5         then color[p[z]]<-BLACK
6           color[y]<-BLACK
7           color[p[p[z]]]<-RED
8           z<-p[p[z]]
9     else if z=right[p[z]]
10       then z<-p[z]
11         left-rotate(T,z)
12       color[p[z]]<-BLACK
13       color[p[p[z]]]<-RED
14       right-rotate(T,p[p[z]])
15   else same as then clause w/right left exchanged
16   color[root[T]]<-BLACK
CASE 1

• Z’s uncle y is red
• p[p[z]] is black (rb tree invariant)
• Push red color up one level
• Possibly need to fix further up
CASE 2

- Z’s uncle y is black and z is a right child
- Make into case 3 by left rotation
CASE 3

- Z’s uncle y is black and z is a left child
- Do a right rotation of Z’s parent and great grandparent
Rotations

Case 2

Case 3
Removals

- Standard remove procedure
- If we remove a red node -> invariants are fine
- If we remove a black node -> need to fix up tree (changed black count)
Removal Fix-up

• Problem - current path has black count that is 1 too low
• Idea - either
  • keep pushing black count problem upward until entire tree has black count 1 less (coloring uncles red) (case 1/2)
  • rotate/recolor nodes so that we locally fix black count (case 3/4)
• Chapter 13.4 gives details - read, but won’t be on any tests
Runtime

- $O(\log n)$