Binary Search Trees

A binary tree in which for each node $y$:
- If $x$ is in $y$'s left subtree, then $x \leq y$.
- If $z$ is in $y$'s right subtree, then $y \leq z$.

Binary Search Trees can do:
- Insert$(S,x)$, Delete$(S,x)$
- Search$(S,k)$, Minimum$(S)$, Maximum$(S)$
- Successor$(S,x)$, Predecessor$(S,x)$

- Combine Min- and Max-Priority Queue & Dictionary
  - Can you do “Increase-Key$(S,x,k)$” in a BST?

- “Obvious” implementation BST can be unbalanced.
  $O(\log n)$ expected, $O(n)$ worst case (n random requests).
**Balanced Search Trees**

Basic idea: Ensure tree height is $O(\lg n)$, so each dynamic set operation takes $O(\lg n)$.

$n = \text{size of the set}$

Various implementations:

- Red-Black trees, B - trees (featured in CLRS)
- AVL trees, 2-3 trees, Splay trees (other methods)
- Binomial heaps, Fibonacci heaps (allow merging)
- Patricia trees ( $O(k)$ for length-$k$ keys)

---

**Red-Black Trees**

BST, guaranteed to be nearly balanced.

1. Nodes have key, pointer, & two (possibly NIL) children. "NIL" is really a black leaf – we’ll gloss over this.
2. Every node is colored “red” or “black”.
3. Root is black.
4. Red node can’t have red parent.
5. Paths from root to NIL’s all have the same number of black nodes (the “black-height”).

[Diagram of red-black tree with black and red nodes]
Node depth is $O(lg n)$ in R-B tree

Black Subtree: ignore red nodes
- Every non-leaf has 2, 3, or 4 children.
- Every leaf has the same depth, $k$.
- Thus # black nodes $\geq 2^{k+1} - 1$ ("=" for full binary tree).

- In full red-black tree, $n \geq 2^{k+1} - 1$ ($n = \#$ nodes)
  - So $lg n \geq lg (2^{k+1} - 1) \geq lg 2^{k} = k$; that is, $k < lg n$.
  - Longest path $\leq 2k$ (obvious from picture).
  - So longest path $\leq 2 lg n$.

<table>
<thead>
<tr>
<th>bent</th>
<th>straight</th>
<th>bent</th>
<th>straight</th>
</tr>
</thead>
<tbody>
<tr>
<td>bent</td>
<td>straight</td>
<td>bent</td>
<td>straight</td>
</tr>
</tbody>
</table>

Insertions into R-B tree

- Insert as new red leaf in correct spot.
- May create "red menace" (red node with red parent)
- Eliminate, promote, or straighten red menace.
  - Details on next page.
  - After at most $2 \times$ node depth moves, red menace will be gone
- If root is red, color it black
Moving Red Menaces

• **Case 1**: Uncle (grandparents other child) is red.
  - Recolor parent, uncle & grandparent.
  - Grandparent may be a red menace now.

• **Case 2**: Uncle is black, red menace is bent.
  - Rotate red menace with parent.
  - Red menace is now straight.
  - Proceed to Case 3

• **Case 3**: Uncle black, red menace straight.
  - Rotate & recolor parent & grandparent.
  - Red menace disappears.

Your turn

• Insert 1, 2, 3, ... into empty tree.

Other Operations

• Search
  
  Compare to root, go obvious direction, recursively.

• Successor
  
  If node has non-empty right subtree, find its MIN.
  Else, return closest ancestor that’s bigger (if any).

• Delete
  
  Complicated (like insert).
Augmenting Search Trees (chapter 14)

Basic idea:

Suppose we want to maintain additional information about data in a dynamic set.

suppose this information can be computed at each node using only the data at that node and its immediate children.

Such information is called a synthesized attribute.

We can do so without increasing the big-O complexity of Insert or Delete.

Why? Because the only information that is modified is on the (original) path from leaf to root.

the tree may shift a bit when we Insert or Delete

(Trivial) Example

Suppose satellite data includes a “size” field

At each node \( n \) we can maintain:

1. maximum size of any node in \( n \)'s subtree, and/or
2. total size of all nodes in \( n \)'s subtree.
Total Size Example

Example (continued)

Previous slide. Still has red menace.
Is rank a synthesized attribute? ("Rank" means place in sorted list)

In both these trees, root has rank-1 and rank-4 child. But it’s rank is different.

So rank can’t be computed just from children’s ranks.

But ...

Search Tree & Rank in $O(\lg n)$ time

“Number of nodes in subtree” is synthesized.

Rank can be computed by adding
1 + nodes in left subtree
+ # of “left ancestors”
+ # nodes in their left subtrees.
while going from root to the node.

Rank of node “17” is the number of nodes inside dotted line.
**Interval Trees**

(Closed) interval \([a,b]\) is \(\{ x \in \mathbb{R} | a \leq x \leq b \}\).

**Interval tree** is a search tree with:
- key = left-hand endpoint of interval
- satellite data = right-hand endpoint of interval
- synthesized attribute = max r-h endpoint of subtree

**Overlapping Interval Problem**

(Two intervals overlap if they have any points in common.)

Return an interval in tree that \([x,y]\) overlaps, or “none”.

- **Case 1**: \([x,y]\) overlaps root: you’re done.
- **Case 2**: \([x,y]\) is entirely left of root: search left subtree.
  - It can’t possibly overlap root or anything in right subtree.
  - (Try \([6,10]\) in previous example.)
- **Case 3**: \([x,y]\) is entirely right of root & max right-hand endpoint in root’s left subtree: search right subtree.
  - It can’t overlap root or anything in left subtree. (Try \([17,18]\).)
- **Case 4**: \([x,y]\) is right of root, but not of max right-hand endpoint in root’s left subtree: overlap in left subtree.
  - Continue with \([17,18]\) example - this case holds at second step.