Last Time

• Binary Search Trees
  – Log time operations for `put()` and `get()`
  – Run time analysis
  – Order-based methods
Order-Based Methods

• Selection?

select(int k) {
    return select(root, k).key;
} //select

select(Node x, int k) {
    if (x == null) {
        return null;
    } //if

    int t = size(x.left);
    if (t > k) {
        return select(x.left, k);
    } else if (t < k) {
        return select(x.right, k-t-1);
    } else {
        return x;
    } //if-else
} //select
Order-Based Methods

delete(Key key) {
    root = delete(root, key);
} //delete

delete(Node x, Key key) {
    if (x == null) { return null; }  
    int cmp = key.compareTo(x.key);
    if (cmp < 0) { x.left = delete(x.left, key); }  
    if (cmp > 0) { x.right = delete(x.right, key); }  
    else {
        if (x.right == null) { return x.left; }  
        if (x.left == null) { return x.right; }  
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    } //if-else
    x.N = size(x.left) + size(x.right) + 1;
    return x;
} //delete
Order-Based Methods

deleteMin(Node x) {
    if (x.left == null) {
        return x.right;
    } //if
    x.left = deleteMin(x.left);
    x.N = size(x.left) + size(x.right) + 1;
    return x;
} //delete
## Can We Do Better?

<table>
<thead>
<tr>
<th>Algorithm (Data Structure)</th>
<th>Worst-Case Cost (After $N$ Inserts)</th>
<th>Average-Case Cost (After $N$ Random Inserts)</th>
<th>Efficiently Support Ordered Operations?</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (unordered linked list)</td>
<td>$N$</td>
<td>$N/2$</td>
<td>no</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\lg N$</td>
<td>$\lg N$</td>
<td>yes</td>
</tr>
<tr>
<td>binary tree search (BST)</td>
<td>$N$</td>
<td>$1.39 \lg N$</td>
<td>yes</td>
</tr>
</tbody>
</table>
2-3 Search Trees

• A 2-3 Search Tree is either empty, or:
  – A 2-node, with one key (and value) and two links: a left link to a 2-3 search tree with smaller keys, and a right link to a 2-3 search tree with larger keys
  – A 3-node, with two keys (and associated values) and three links: a left link to a 2-3 search tree with smaller keys, a middle link to a 2-3 search tree with keys between the node’s keys, and a right link to a 2-3 search tree with larger keys
• Goal: Keep all null links the same distance from the root.
2-3 Trees – Search

- Generalized version of the algorithm for BSTs.
  - Compare the key against the key(s) at the root.
    - If equal, we have a search hit.
    - If not, we follow the link to the subtree corresponding to the interval of key values that could contain the search key.
  - Repeat until the search key is found or until a null node is reached.
2-3 Trees – Search

successful search for H

H is less than M so look to the left

unsuccessful search for B

B is less than M so look to the left

H is between E and J so look in the middle

B is less than E so look to the left

found H so return value (search hit)

B is between A and C so look in the middle link is null so B is not in the tree (search miss)
2-3 Trees – Insert

• We have to consider a lot of different possibilities, to try to keep the tree balanced:
  – Insert into a 2-node
  – Insert into a single 3-node
  – Insert into a 3-node whose parent is a 2-node
  – Insert into a 3-node whose parent is a 3-node
  – Splitting temporary 4-nodes
  – Probably some crying too
2-3 Trees – Insert into a 2-Node

- If we perform an unsuccessful search and then add a new node with a key to the bottom of a tree, the new tree would not remain perfectly balanced.
- So, we will just replace the node with a 3-node containing its key plus the new key to be inserted.
2-3 Trees – Insert into a Single 3-Node

• A tree containing only a single 3-Node is already balanced.

• We can temporarily add a new Key into the Node, then split that temp 4-Node into a 2-3 Tree made up of three 2-Nodes, with the middle key in the root.

• Old tree height = 0, new tree height = 1, all null links still the same distance from the root.
2-3 Trees – Insert into a 3-Node with a 2-Node Parent

• We need to make room for a new node while keeping the tree balanced.
  – Turn the 3-Node into a temporary 4-Node (as before).
  – Instead of splitting the 4-Node into a 2-3 Tree with three nodes, just move the middle key to the parent.
  – Split the pre-insertion 3-Node into separate 2-Nodes.

• The tree remains ordered because the middle key was the one moved to the parent.

• The tree remains balanced, because no null links have changed tree levels.
2-3 Trees – Insert into a 3-Node with a 2-Node Parent

- Inserting Z
  - Search for Z ends at this 3-node
  - Replace 3-node with temporary 4-node containing Z
  - Replace 2-node with new 3-node containing middle key
  - Split 4-node into two 2-nodes, pass middle key to parent
2-3 Trees – Insert into a 3-Node with a 3-Node Parent

• Same goal as before; make room for a new node while keeping the tree balanced.
  – Turn the 3-Node into a temporary 4-Node (as before).
  – Move the middle key to the parent (as before).
  – Now the parent is temporarily a 4-Node, so repeat the process until we either reach a 2-Node, or until reaching a 3-Node at the root.

• As before, the tree remains balanced and no null links have changed tree levels.
2-3 Trees – Insert into a 3-Node with a 3-Node Parent

inserting D

search for D ends at this 3-node

add new key D to 3-node to make temporary 4-node

add middle key C to 3-node to make temporary 4-node

split 4-node into two 2-nodes
pass middle key to parent

add middle key E to 2-node to make new 3-node

split 4-node into two 2-nodes
pass middle key to parent
2-3 Trees – Splitting a 3-Node at the Root

• If we have 3-Nodes along the full path from the insertion point to the root, we end up generating a temporary 4-Node at the root.
  – In this case, follow the steps to replace a tree consisting of a single 3-Node – split the temporary 4-Node into three 2-Nodes.

• The tree remains balanced.
  – All null links have now moved one step further from the root, so the height of the tree increased by 1, but all nodes are still the same distance away.
2-3 Trees – Splitting a 3-Node at the Root

inserting D

search for D ends at this 3-node

add new key D to 3-node to make temporary 4-node

add middle key C to 3-node to make temporary 4-node

split 4-node into two 2-nodes pass middle key to parent

split 4-node into three 2-nodes increasing tree height by 1
Splitting a Temporary 4-Node is Guaranteed to Retain Balance
2-3 Tree Height

- The height of a 2-3 Tree is input-independent.
  - The order of the keys does not matter – the trees will have different structures, but the same height.
  - Left: adding keys S E A R C H X M P L
  - Right: adding keys A C E H L M P R S X
2-3 Tree Analysis

• **Theorem:** Search and insert operations in a 2-3 Tree with $N$ keys are guaranteed to visit at most $\log(N)$ nodes.

• **Proof:**
  – Biggest height of a 2-3 Tree with $N$ Nodes is if all Nodes are 2-Nodes.
    • In this case, height = $\log(N)$.
  – Smallest height of a 2-3 Tree with $N$ Nodes is if all Nodes are 3-Nodes.
    • In this case, height = $\log_3(N)$.
  – Both of these fall in the $O(\log(N))$ complexity class.
2-3 Trees vs BSTs

- The height of a 2-3 Tree with 1 billion keys is guaranteed to be between 19-30.
- The height of a BST with 1 billion keys is guaranteed to be between 30-1,000,000,000.

- Downsides:
  - 2-3 Trees are certainly more complicated structures, and tougher to implement.
  - Because we need to maintain different types of nodes and rebuild the tree structure frequently, a 2-3 Tree may still be slower than a BST.

- 2-3 Trees provide insurance against the worst case.
2-3 Trees vs BSTs
## We Did Better

<table>
<thead>
<tr>
<th>Algorithm (Data Structure)</th>
<th>Worst-Case Cost (After N Inserts)</th>
<th>Average-Case Cost (After N Random Inserts)</th>
<th>Efficiently Support Ordered Operations?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Search</td>
<td>Insert</td>
<td>Search Hit</td>
</tr>
<tr>
<td>Sequential search (unordered linked list)</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>Binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>Binary tree search (BST)</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
</tr>
<tr>
<td>2-3 tree search</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>1.00 lg N</td>
</tr>
</tbody>
</table>
Any Questions?