Data Structures

Today's Agenda

- **Continue Discussing Trees**
- Examine the algorithm to insert
- Examine the algorithm to remove
- Begin discussing efficiency of tree
- Are there any alternatives?
  - 2-3
  - 2-3-4 (next time)
  - red-black trees (next time)
  - AVL (next time)

Tree Insert

- From last time...
  - everyone should have prepared an algorithm for insert
  - remember, insert always inserts data at a leaf
  - this is similar, in many regards to inserting data at the end of a linear linked list; the good news is that we don't have to special case the situation where we are trying to rearrange pointers by inserting in the middle

LLL Recursive Insert

- For example, let's review what it would be like to insert into a LLL --- adding at the end all of the time:

```
void insert(node * & head, data & d){
  if (!head) {
        head = new node;
        head->d = d;
        head->next = NULL;
  }
  else insert(head->next, d);
}
```

LLL Recursive Insert

- Why does this work?
- Why does head need to be passed in? Why can't we just use a data member named head?
- Why does head need to be passed by reference?
- How does it connect up the nodes?
- Why was this inefficient for a linear linked list?

LLL Recursive Insert

- Another way to write this:

```
node * insert(node * head, data & d){
  if (!head) {
        head = new node;
        head->d = d;
        head->next = NULL;
        return head;
  }
  head->next = insert(head->next,d);
  return head;
}
```

LLL Recursive Insert

- Is this approach more or less efficient?
- How do the nodes get connected?
- Does it handle the “special case” where head is null to begin with?
- Does it ever dereference a null pointer?
- How about copies being placed on the program stack? How does this compare with the previous recursive solution?

Tree Recursive Insert

- Now let's apply what we have learned to insert into a binary search tree
- Remember, if the data being inserted is less than the root, we want to traverse left
- If the data being inserted is greater than the root, we want to traverse right
- If it is the same, pick a consistent approach to deal with it (either left or right)

Tree Recursive Insert

```
void insert(node * & root, data & d){
  if (!root) {
        root = new node;
        root->d = d;
        root->left = NULL;
        root->right = NULL;
  }
  else if (root->d > d)
        insert(root->left, d);
 else   insert(root->right, d);
}
```

Tree Recursive Insert

```
node * insert(node * root, data & d){
  if (!root) {
          root = new node;
          root->d = d;
          root->left = NULL;
          root->right = NULL;
  }
  else if (root->d > d)
     root->left = insert(root->left, d);
 else root->right=insert(root->right, d);
 return root;
}
```

Tree Recursive Insert

- Do both of these approaches work?
- Which is most efficient?
- How does this compare in terms of efficiency with the linear linked list approach?
- What type of client interface should we provide?
  - insert (data &);

Tree Recursive Insert

- What you should have concluded is that the efficiency of this approach depends greatly on the "shape" of the binary search tree
- For example, what if you entered in 1000 names all in sorted order?
  - what shape would your BST be?
- What if, instead, the data was entered in random order?
  - which is better and why?

Tree Removal

- Now let's discuss removing nodes from a binary search tree
- We will find this is not as simple, because we cannot restrict the removal to just working at the leaf
- There are a number of special cases we need to consider...can you think of them?

Tree Removal: Special Cases

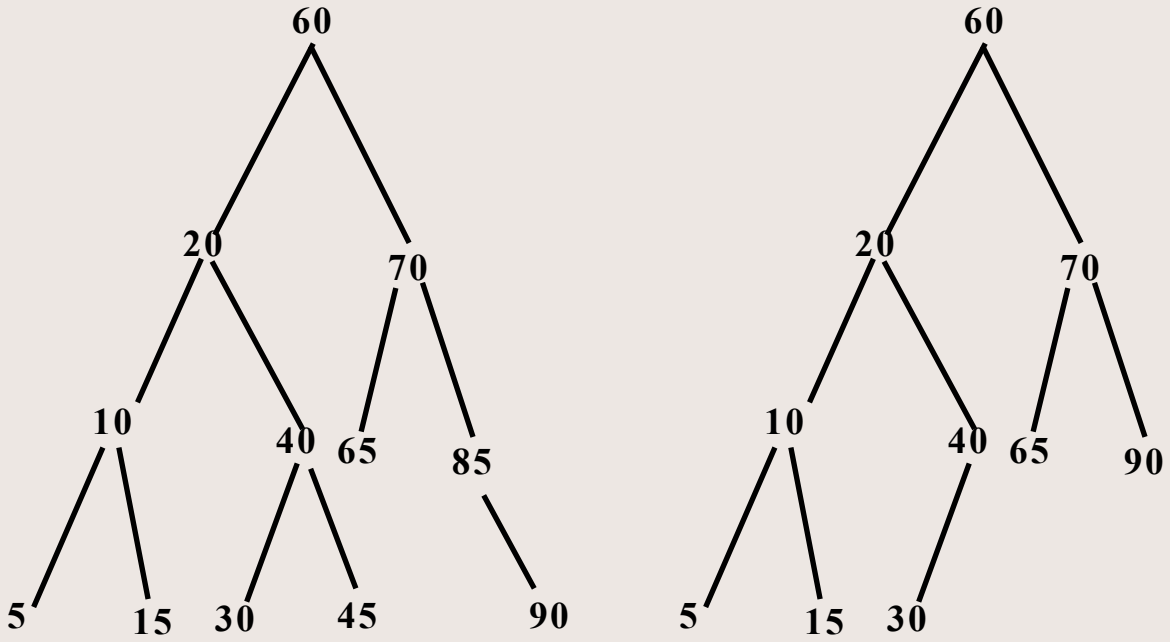
- Tree is empty (never forget this one!)
- The data to be removed is not in the tree
- The node containing the data has no children (i.e., it is a leaf)
- The node containing the data has one child (i.e., it is an internal node with a single child that can be “inherited”)
- The node has **two children**

Tree Removal: Special Cases

- To remove a leaf
  - we simply change the Left or Right pointer in its parent to NULL.
- When there is one child,
  - we end up letting the parent of the node to be deleted adopt the child!
  - It ends up not making a difference if the child was a left or a right child to the node being deleted.

Tree Removal: Special Cases

- Remove 45 (a leaf)
- Remove 85 (one child)



Tree Removal: Special Cases

- Removing a node with 2 children
  - is the most difficult.
  - Both children cannot be "adopted" by the parent of the node to be deleted...this would be invalid for a binary search tree.
  - The parent has room for only one of the children to replace the node being deleted.
  - So, we must take on a different strategy.

Tree Removal: Special Cases

- Removing a node with 2 children
  - One way to do this is to not delete the node; instead replace the data in this node with another node's data...it can come from immediately after or before the search key being deleted.
  - How can a node with a key matching this description be found?
    - Simple.

Tree Removal: Special Cases

- Removing a node with 2 children
  - Remember that traversing a tree INORDER causes us to traverse our keys in the proper sorted order.
  - So, by traversing the binary search tree in order, starting at the to-be-deleted node (i.e., the to-be-replaced node)...we can find the search key to replace the deleted node by traversing the next node INORDER.
  - It is the next node searched and is called the **inorder successor.**

Tree Removal: Special Cases

- Removing a node with 2 children
  - Since we know that the node to be deleted has two children, it is now clear that the inorder successor is the leftmost node of the "deleted nodes" right subtree.
  - Once it is found, you copy the value of the item into the node you wanted to delete and remove the node found to replace this one -- since it will never have two children.

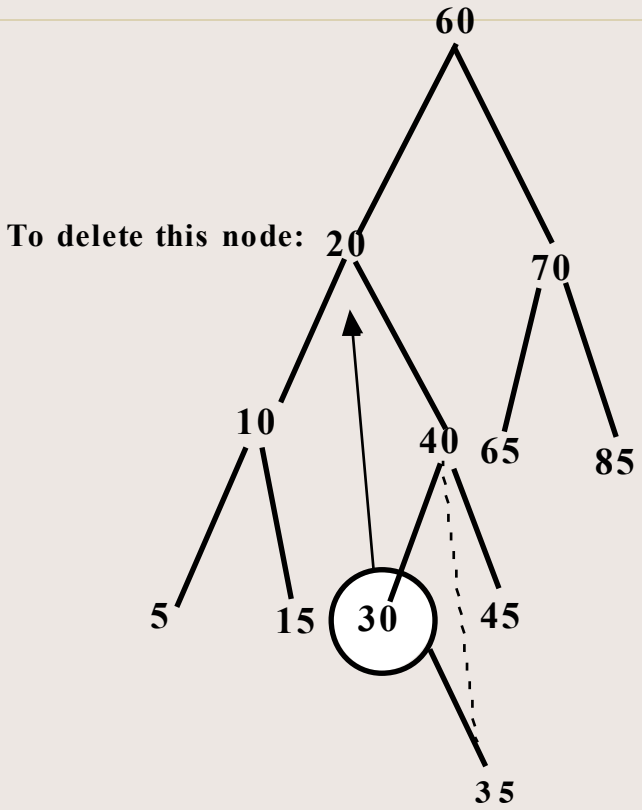
Tree Removal: Special Cases

- Removing a node with 2 children
  - However, there is a special case
  - If the right child has no left children, then the right child becomes the inorder successor
- Should this be done recursively or iteratively?
  - it is common to “find the node who’s data matches the data to be removed” using recursion
  - but, finding the inorder successor **should** be done iteratively, because we simply “loop” until the left pointer is null.

Tree Removal: Special Cases

- Anything else?
  - Yes, as you loop looking for the inorder successor, it is important to either use the “look ahead” approach or keep track of a previous pointer
  - Why? Well, the parent to the inorder successor’s left child pointer must be changed to point to the inorder successor’s right child!
  - yep, that is right. The inorder successor may have a child...just not to the left!!!!!
  - Remember, using a previous pointer is more efficient than a look ahead approach

Tree Removal: Special Cases



Tree Efficiency

- We already know that the maximum height of a binary tree with N nodes is a height of N.
  - And, an N-node tree with a height of N is LLL
- It is interesting to consider how many nodes a tree might have given a certain height.
  - If the height is 3, then there can be anywhere between 3 and 7 nodes in the tree.
  - Trees with more than 7 nodes will require that the height be greater than 3. A full binary tree of height h -- should have 2^h-1 nodes in that tree

Tree Efficiency

- Look at a diagram ... counting the nodes in a full binary tree
  - A full binary tree of height at
    - Level 1: # of nodes = 2^1-1 = 1
    - Level 2: # of nodes = 2^2-1 = 3
    - Level 3: # of nodes = 2^3-1 = 7

Tree Efficiency

- In fact, the height of binary trees can be mathematically predicted
- Given that we need to store N nodes in a binary tree, the maximum height is N
- The minimum height is:
  - $\log_2 N + 1$
- Given a height of a tree, H, the minimum and maximum number of nodes would be:
  - min: H max: $2^H - 1$

Tree Efficiency

- The distance of a node from the root
  - determines how efficiently it can be located
  - the shorter we can make the tree, the easier it is to locate any desired node in the tree
- To determine if a tree is balanced
  - we can calculate its balance factor
  - which is the difference in heights between its left and right subtrees
  - Balance = H_L - H_R

Tree Efficiency

- A tree is balanced
  - if its balance factor is zero and its subtrees are also balanced
  - but, since this definition occurs so seldom, an alternate definition is more generally applied:
  - a binary tree is balanced if the height of its subtrees differs by no more than one (i.e., the balance factors can be -1, 0, or 1) and its subtrees are also balanced.

Tree Efficiency

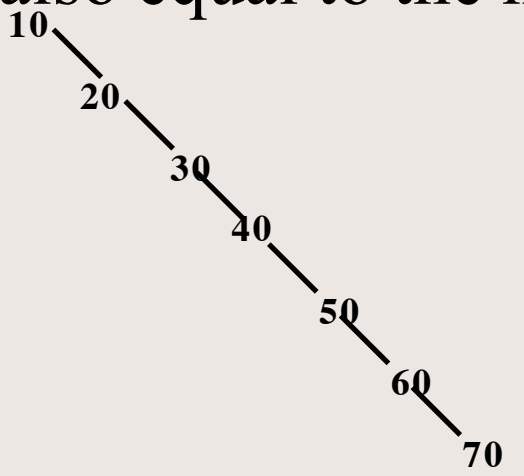
- Using balanced search trees, we can achieve a high degree of efficiency for implementing our ADT Table operations.
- This efficiency depends on the balance of the tree.
- We will find that balanced trees can be searched with efficiency comparable to the binary search.

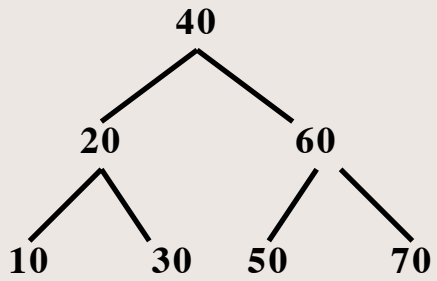
Tree Efficiency

- With a binary search tree,
  - the actual performance of Retrieve, Insert, and Delete actually depends on the tree's height. Why?
  - Because we must follow a path from the root of the tree down to the node that contains the desired item.
  - At each node along the path, we must compare the key to the value in the node to determine which branch to follow.

Tree Efficiency

- With a binary search tree,
  - Because the maximum number of nodes that can be on any path is equal to the height of the tree, we know that the maximum number of comparisons that the table operations can require is also equal to the height.





Tree Efficiency

- Trees that have a linear shape behave no better than a linked list.
- Therefore, it is best to use variations of the basic binary search tree together with algorithms that can prevent the shape of the tree form degenerating.
- Four variations are the 2-3 tree, 2-3-4 tree, red-black tree and the AVL tree.
- The first two are “perfectly balanced” trees

2-3 Trees

- 2-3 trees permit the number of children of an internal node to vary between two and three.
- This feature allows us to "absorb" insertions and deletions without destroying the tree's shape.
  - We can therefore search a 2-3 tree almost as efficiently as you can search a minimum-height binary search tree...and it is far easier to maintain a 2-3 tree than it is to guarantee a binary search tree having minimum height.

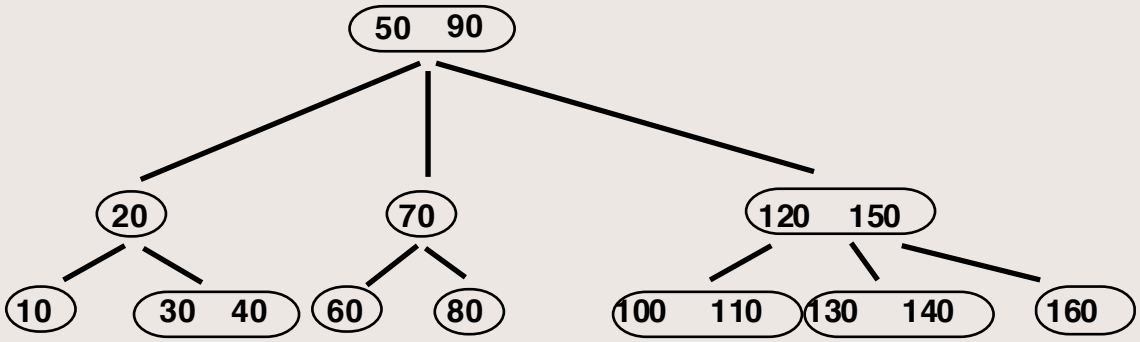
2-3 Trees

- Every node in a 2-3 tree is either a leaf, or has either 2 or 3 children.
  - So, there can be a left and right subtree for each node...or a left, middle, and right subtree.
- To use a 2-3 tree for implementing our ADT table operations
  - we need to create the tree such that the data items are ordered. The ordering of items in a 2-3 search tree is similar to that of a binary search tree. In fact, you will see that to retrieve -- our pseudo code is very similar to that of a binary search tree.

2-3 Trees

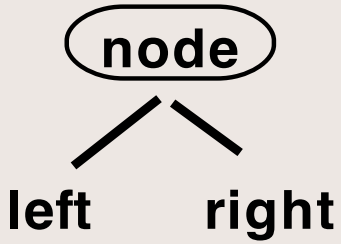
- The big difference is that nodes can contain more than one set of data.
- If a node is a leaf, it may contain either one or two data items!
- If a node has two children, it must only contain 1 data item.
- But, if a node has three children, it must contain 2 data items.

2-3 Trees

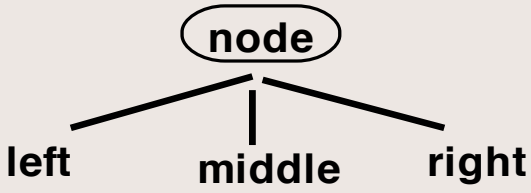


2-3 Trees

- For "nodes" that contain only one data item
  - there can be either no children or 2 children:
  - In this case, the value of the key at the "node" must be greater than the value of each key in the left subtree and smaller than the value of each key in the right subtree.
  - The left and right subtrees must each be a 2-3 tree.



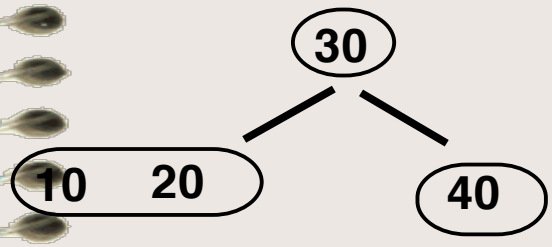
2-3 Trees

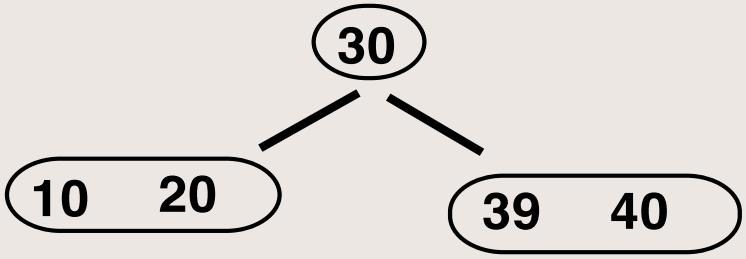


- For "nodes" that contain two data items
  - there can be either no children or 3 children:
  - In this case, the value of the smaller key at the "node" must be greater than the value of each key in the left subtree and smaller than the value of each key in the middle subtree.
  - The value of the larger key at the "node" must be greater than the value of each key in the middle subtree and smaller than the value of each key in the right subtree.

2-3 Trees

- With insertions, since the nodes of a 2-3 tree can have either 2 or 3 children and can contain 1 or two data values --
  - we can make insertions while maintaining a tree that has a balanced shape. That is the goal!
  - try to insert 39 and 40 into the following tree:





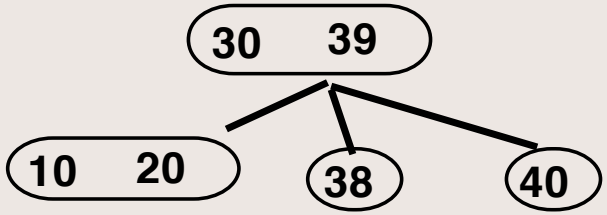
2-3 Trees

- Notice, we still insert at a leaf
  - but now when we reach the last node in a path that node can simply absorb the new data if it has only 1 piece of data in it
  - but, what if there are two pieces of data?
  - the process involves finding the middle data item between the two in the node and the new item, splitting the node, and pushing up to the parent the middle data item to be inserted
  - this process is **very** recursive

2-3 Trees

- For example, now, insert 38.
- Again, we would search the tree to see where the search will terminate if we had tried to find 38 in the tree...this would be at node <39 40>.
- Immediately we know that nodes contain 1 or 2 data items...but NOT THREE!
- So, we can't simply insert this new item into the node.

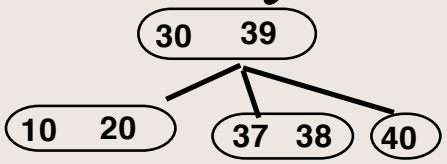
2-3 Trees



- Instead, we find the smallest (38), middle (39) and largest (40) data items at this node.
- You can move the middle value (39) up to the node's parent and separate the remaining values (38,40) into two nodes attached to the parent.
- Notice that since we moved the middle value to the parent -- we have correctly separated the values of its children. See the results:

2-3 Trees

- Now, insert 37.
- This is easy because it belongs in a leaf that currently contains only 1 data value (38). The result is:



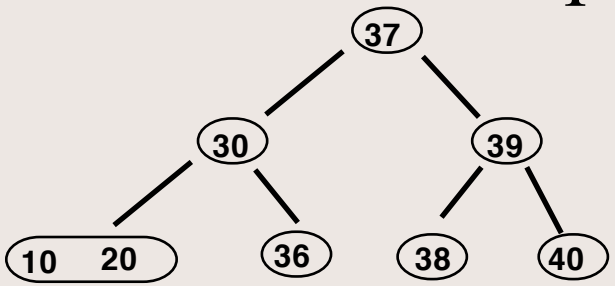
- Now, insert 36.

2-3 Trees

- Inserting 36...
  - We find that this number belongs in node <37 38>.
- But, once again we realize that we can't have 3 values at a node...so we locate the smallest (36), middle (37), and largest (38) values.
- We then move the middle value (37) up to the parent and attach to the parent two nodes (the smallest and the largest).

2-3 Trees

- However, notice that we are not finished. We have now tried to move 37 to the parent --
  - trying to give it 3 data items (think recursion!!) -- and trying to give it 4 children!
- As we did before, we divide the node into the smallest (30), middle (37), and largest (39) values...and move the middle value up to the node's parent.



2-3 Trees

- So, here is the insertion algorithm.
- To insert a value into a 2-3 tree we first must locate the leaf which the search for such a value would terminate.
- If the leaf only contains 1 data value, we insert the new value into the leaf and we are done.
- However, if the leaf contains two data values, we must split it into two nodes (this is called splitting a leaf).

2-3 Trees

- The left node gets the smallest value and the right node gets the largest value.
- The middle value is moved up to the leaf's parent.
- The new left and right nodes are now made children of the parent.
- If the parent only had 1 data value to begin with, we are done.

2-3 Trees

- But, if the parent had 2 data values, then the process of splitting a leaf would incorrectly make the parent have 3 data values and 4 children!
  - So, we must split the parent (this is called splitting an internal node).
  - You split the parent just like we split the leaf...except that you must also take care of the parent's four children.

2-3 Trees

- You split the parent into two nodes.
  - You give the smallest data item to the left node and the largest data item to the right node.
  - You attach the parent's two leftmost children to this new left node and the two rightmost children to the new right node.
  - You move the parent's middle data value to it's parent..and attaching the left and right newly created nodes to it as its two new children.
  - and so on.

2-3 Trees

- This process continues...splitting nodes...moving values up recursively until a node is reached that only has 1 data value before the insertion.
- The height of a 2-3 tree only grows from the top.

2-3 Trees

- An increase in the height will occur if every node on the path from the root of the tree to the leaf where we tried to insert an item contains two values.
  - In this case, the recursive process of splitting a node and moving a value up to the node's parent will eventually reach the root.
  - This means we will need to split the root. You split the root into two new nodes and create a new node that contains the middle value. This new node is the new root of the tree.