

Trees

CIS*2520 Summer 2006

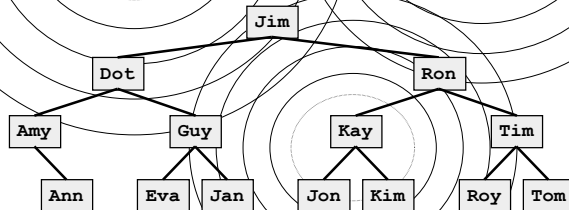
Overview

- Recall:
 - linked lists: only linear access permitted (vs. array)
 - can we find a method to rearrange the nodes of a linked list so that we can search in time $O(\log_2(n))$ instead of $O(n)$?
- Trees
 - non-linear structure (2-dimensional)
 - formal definition:
 - an acyclic graph with vertices and edges such that a path exists between any pair of vertices

CIS*2520 (506)

Binary Search Revisited

- Consider:
 - binary search on:
 - Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom
 - resulting comparison tree:



CIS*2520 (506)

Binary Search Revisited (cont.)

- Idea:
 - store nodes in structure of comparison tree itself
 - advantages of linked storage
 - obtain speed of binary search for retrieval

CIS*2520 (506)

Terminology

- **Nodes (vertices), Arcs (edges)**
- **Root**
 - the unique node with no incoming edges
 - there is a unique path from the root to every other node in the tree
- **Path**
 - a sequence of consecutive edges
- **Leaf**
 - a vertex with no outgoing edges
- **Children**
 - the vertices immediately below vertex v are the children of v

CIS*2520 (506)

Terminology (cont.)

- **Parent**
 - the vertex immediately above node v is the parent of v
- **Siblings**
 - vertices with the same parent
- **Descendent, Ancestor**
- **Subtree**
 - each child of the root is the root of a smaller tree called a subtree
 - NOTE: recursive property of trees

CIS*2520 (506)

Binary Trees

- **Binary Tree**
 - a binary tree is either empty, or consists of a node called the **root**, together with two binary trees called the **left subtree** and the **right subtree** of the root
- **Binary Search Tree**
 - a binary search tree is a binary tree that is either empty or in which each node contains a key that satisfies the following conditions:
 1. all keys (if any) in the left subtree of the root precede the key in the root
 2. the key in the root precedes all keys (if any) in its right subtree
 3. the left and right subtrees are again binary search trees

CIS*2520 (506)

Binary Trees (cont.)

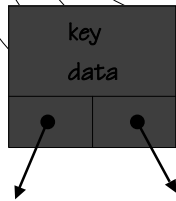
- **Contrast binary trees with 2-trees (considered previously)**
 - a node in a 2-tree has *either* 0 or 2 children
 - a node in a binary tree has *at most* 2 children
- **Consider: a binary tree with 3 nodes**
 - one of them must be the root of the tree
 - how many different trees are there?

CIS*2520 (506)

Binary Tree Implementation

- As with linked lists, trees are built using many elements (or nodes) all with the same basic structure

- Basic building block:



```
typedef struct TreeNode_t
{
    KeyType *key;
    DataType *data;
    TreeNode *left;
    TreeNode *right;
} TreeNode;
...
KeyType *getKey() {...}
DataType *getData() {...}
TreeNode *getLeft() {...}
TreeNode *getRight() {...}
/* note there are other possible
representations for a tree */
```

CIS*2520 (506)

Binary Tree Implementation (cont.)

- Empty subtrees can be marked with a NULL pointer
 - similar to the end of a list in some of the examples we have examined
- Consider this as an ADT**
 - note:** we will discuss many algorithms in terms of the tree structure directly
 - this does not imply that the tree would not be implemented as a proper ADT with methods such as these defined internally (recall discussion of recursive list searching)
 - all such functions we present here are named with a leading underscore as they would not be called externally in a TreeADT

CIS*2520 (506)

Searching Binary Trees

- To search for a target key we first compare it to the key at the root of the tree, if it is not the same
 - if the target is smaller than the key at the root then we search the left subtree;
 - if the target is larger than the key at the root then we search the right subtree
- Repeat this search in the subtree
- Stopping condition?
 - find the key (success)
 - empty subtree (failure)

CIS*2520 (506)

Binary Tree Search

- Consider a treeSearch method
 - receives a pointer to the root of the tree (or subtree) as one of its arguments
 - PRE:
 - pointer to root of tree must be the root of a binary search tree
 - POST:
 - method will return a reference to the data associated with the target key if the search was successful, null otherwise
 - note:** could define the function to return 1/0 (true/false) and fill in a value parameter with the data if found, however this is generally more cumbersome
 - recall:** convention of using *lessThan*, *greaterThan* and *equals* functions for generic comparisons

CIS*2520 (506)

Binary Tree Search Implementation

```
DataType *_treeSearch(TreeNode *root, KeyType *target)
{
    if (root != NULL)
    {
        if (equals(target, getKey(root))
            return(getData(root));
        else if (lessThan(target, getKey(root))
            return(_treeSearch(getLeft(root), target));
        else
            return(_treeSearch(getRight(root), target));
    }
    return(NULL);
} /* _treeSearch*/
```

- This is still tail recursion (don't let the if statement confuse you)
 - easily removed with loop added

CIS*2520 (506)

Binary Tree Search Implementation (cont.)

- Iterative implementation (tail recursion removed)
- Note: root is passed by value (what is the significance of this again?)

```
DataType *_treeSearch(TreeNode *root, KeyType *target)
{
    while (root != NULL)
    {
        if (equals(target, getKey(root))
            return(getData(root));
        else if (lessThan(target, getKey(root))
            root = getLeft(root);
        else
            root = getRight(root);
    }
    return(NULL);
} /* _treeSearch */
```

CIS*2520 (506)

Traversal of Binary Trees

- Most applications require not only that we be able to find nodes, but that we be able to move through all the nodes in the tree, visiting each one in turn.
- If there are N nodes, then there are N! different orders in which they could be visited
 - most of these are irregular
 - we almost always want to proceed such that the same rule is applied at each node
- At a given node there are generally three tasks we wish to do in some order:
 1. visit the node itself (i.e. perform some operation)
 2. traverse the left subtree
 3. traverse the right subtree

CIS*2520 (506)

Traversal of Binary Trees (cont.)

- There are only 3 ways to do this such that we consider the left subtree before the right (named according to the step at which the node is visited)
 - PREORDER
– visit, traverse left, traverse right
 - INORDER
– traverse left, visit, traverse right
 - POSTORDER
– traverse left, traverse right, visit
- Traversal functions for each of these are particularly easy to specify thanks to the recursive structure of the tree

CIS*2520 (506)

Tree Traversal Implementation

```
void _preorder(TreeNode *root)
{
    if (root != NULL)
    {
        visit(root);
        _preorder(getLeft(root));
        _preorder(getRight(root));
    }
} /* _preorder */
```

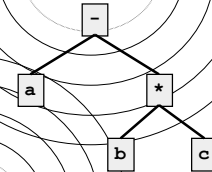
```
void _postorder(TreeNode *root)
{
    if (root != NULL)
    {
        _postorder(getLeft(root));
        _postorder(getRight(root));
        visit(root);
    }
} /* _postorder */
```

```
void _inorder(TreeNode *root)
{
    if (root != NULL)
    {
        _inorder(getLeft(root));
        visit(root);
        _inorder(getRight(root));
    }
} /* _inorder */
```

CIS*2520 (506)

Example

- Consider expression trees
 - built-up from simple operators and operands of an arithmetic or logical expression by placing the operators as interior nodes and operands as the leaves
 - e.g.
 $a - (b * c)$
 - apply traversal algorithms to these trees
 - PREORDER == prefix notation
 - INORDER == infix notation
 - POSTORDER == postfix notation



CIS*2520 (506)

Notes

- Recall previous example---searching names
 - INORDER traversal of a binary search tree gives ascending order (alphabetic in this case)
- Food for thought:
 - treeSort
 - given unordered ordinal values (numbers, strings, etc.)
 - build a binary search tree
 - inorder traversal of tree to get sorted list
 - merits? disadvantages?

CIS*2520 (506)

Tree ADT

- Before considering issues such as building trees, we have enough material at our disposal to consider the tree as an ADT.
- What kinds of operations do we want?
 - insertion (must be able to build a tree)
 - retrieval (find a value in a tree - i.e. treeSearch)
 - deletion (may wish to remove elements from a tree - not always easy)
 - output, size, etc.

CIS*2520 (506)

Tree ADT Specification

- **DEF'N:**
 - *In general:*
A tree is a connected acyclic graph of vertices (element of type T) and edges (links between elements) such that there is a path between any two vertices
 - *Binary Search Tree (BST ADT):*
A binary search tree is a unique vertex called the root together with two binary search trees called the LEFT SUBTREE and RIGHT SUBTREE of the root. The tree is either empty, or each vertex contains a key and satisfies the following conditions:
 1. all keys (if any) in the left subtree of the root precede the key in the root
 2. the key in the root precedes all keys (if any) in the right subtree
 3. the left and right subtrees of the root are again search trees

note: the general definition subsumes BST

CIS*2520 (506)

Tree ADT Specification (cont.)

- **OPERATIONS:**
 - **create**
 - create a new tree initialized to be empty
 - PRE: size > 0
 - POST: tree is initialized and empty
 - **destroy**
 - destroy a tree
 - PRE: n/a
 - POST: tree is destroyed
 - **insert**
 - add a node to the tree
 - n/a
 - new node inserted into tree such that above properties are preserved

CIS*2520 (506)

Tree ADT Specification (cont.)

- **OPERATIONS (cont.):**
 - **retrieve**
 - obtain the data associated with a given key stored in a tree
 - PRE: a node containing the key exists in the tree
 - POST: provides data associated with the given key
 - **delete**
 - remove the node containing a given key from the tree
 - PRE: a node containing the key exists in the tree
 - POST: node containing the given key is removed from the tree such that the above properties are preserved
 - **height**
 - obtain the current height of the tree (length of longest path from the root to any leaf)
 - PRE: n/a
 - POST: provides current height of the tree

CIS*2520 (506)

Tree ADT Specification (cont.)

- **OPERATIONS (cont.):**
 - **isEmpty**
 - determine if a tree is empty
 - PRE: n/a
 - POST: true if tree is empty, false otherwise
 - **size**
 - obtain the number of nodes currently in the tree
 - PRE: n/a
 - POST: provides the number of nodes in the tree
 - **write**
 - **read**
 - **others?**

CIS*2520 (506)

Notes

- We are primarily discussing binary search trees
 - understand the distinction between these and binary trees more generally (2 separate definitions)
 - e.g. expression trees are binary trees, but not BSTs (why?)
- “Key and data” model we are using here is meant to be general (as was the case with List elements previously)
 - key and data may not be separate items
- Binary Search Trees are not unique

CIS*2520 (506)

Insertion Into a Binary Search Tree

- Insertion only has to ensure that the properties of a binary search tree are preserved at each step
 - for the time being we will assume no duplicate keys are permitted (what happens with duplicate keys is implementation specific)
 - BASE CASE: insert into empty tree
 - make root reference the new node
 - OTHERWISE: compare target key with the one in the root
 - if it is less: *new node should be inserted in left subtree*
 - if it is more: *new node should be inserted in right subtree*
 - if it is equal: *duplicate key*
- It should be obvious that this is a perfect recursive definition that permits a direct implementation

CIS*2520 (506)

Insertion Into a Binary Search Tree (cont.)

- Must be careful with dynamic allocation and recursive functions
 - only do allocation in the base case (if allocating at each recursive call by mistake there is a lot of leaking memory...probably not a good thing).
- CAUTION
 - how do we “make root reference the new node”?
 - we cannot change this value directly
 - option:
 - function returns reference to (possibly changed) root
 - assume the convention that insert is called as `root = insert(root);`
 - NOTE: this is not an issue at the level of the ADT operations (why?)
 - implementation alternatives?

CIS*2520 (506)

Insertion Implementation

```
TreeNode *TreeNodeCreate(KeyType *key_val, DataType *data_val)
{
    TreeNode *new = NULL;

    if (new = calloc(1, sizeof(TreeNode)))
        return(NULL);

    key = key_val;
    data = data_val;
    left = NULL;
    right = NULL;

    return(new);
} /* TreeNodeCreate */
```

- as noted previously, key and data may not be separate arguments

CIS*2520 (506)

Insertion Implementation (cont.)

```
TreeNode *_insert(TreeNode *root, KeyType *key, DataType *data)
{
    if (root == NULL)
        return(TreeNodeCreate(key, data));
    else if lessThan(key, getKey(root))
        setLeft(root, _insert(getLeft(root), key, data));
    else if greaterThan(key, getKey(root))
        setRight(root, _insert(getRight(root), key, data));
    else
        /* duplicate key */

    return(root);
} /* _insert */
```

CIS*2520 (506)

Insertion Notes

- Must be careful that recursion is being handled correctly
 - especially with respect to the return value when you are essentially “re-writing” a structure in place
 - very common to need to adjust a value by assignment in a recursive call
- Can recursion be removed here?
 - it isn't quite tail recursive (there is an assignment operation after the recursive call)
 - the local variables do not change throughout the hierarchy of recursive calls, so still easy to remove recursion
 - use a tmp reference for traversal
 - still need to return the root argument

CIS*2520 (506)

Some Observations

- Complexity of insertion?
- Complexity of search?
- What will happen if data to be inserted is presented in alphabetical order?

CIS*2520 (506)

Deletion From a Binary Search Tree

- We indicated that an advantage of linked storage is “dynamic” structure
 - i.e. the ability to insert and delete
 - insertion is trivial
 - deletion is a bit trickier
- Consider cases
 1. node to be deleted is a leaf
 2. node to be deleted has only one non-empty subtree
 3. node to be deleted has two (both) non-empty subtrees

CIS*2520 (506)

Deletion From a Binary Search Tree (cont.)

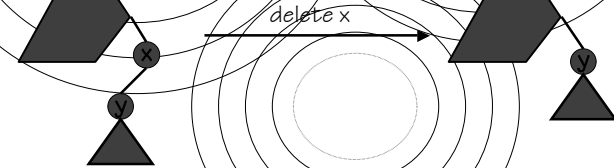
- Node to be deleted is a leaf
 - replace link to deleted node with null (or whatever indicates empty subtree)



CIS*2520 (506)

Deletion From a Binary Search Tree (cont.)

- Node to be deleted has only one non-empty subtree
 - adjust link from parent of deleted node to reference its subtree
 - recall properties of a BST --- does this preserve it?



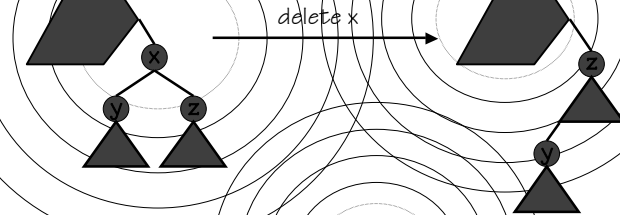
CIS*2520 (506)

Deletion From a Binary Search Tree (cont.)

- Node to be deleted has two (both) non-empty subtrees
 - to which of the subtrees should parent of deleted node now refer?
 - what is done with the other subtree?
 - Solution: attach the right subtree in place of the deleted node and then hang the left subtree on an appropriate node of the right subtree (which one?)
 - every key in the left subtree precedes every key of the right subtree
 - it must be as far left as possible (i.e. take left branches until an empty left subtree is found and hang the subtree at this point)
 - verify that this preserves the properties of a BST

CIS*2520 (506)

Deletion From a Binary Search Tree (cont.)



- Implementation note:
 - we will do the delete in 2 steps: 1) find the node to delete, 2) call another function to remove it
 - could combine this into one, but would complicate the function (recall concept of COHESION)

CIS*2520 (506)

Deletion Implementation

```
TreeNode *_delete(TreeNode *root, KeyType *key)
{
    if (root != NULL)
    {
        if (equals(key, getKey(root)))
            return(_removeNode(root));
        else if (lessThan(key, getKey(root)))
            setLeft(root, _delete(getLeft(root), key));
        else if (greaterThan(key, getKey(root)))
            setRight(root, _delete(getRight(root), key));
    }

    /* otherwise nothing to delete */

    return(root);
} /* _delete */
```

CIS*2520 (506)

Deletion Implementation (cont.)

```
TreeNode *_removeNode(TreeNode *node)
{
    TreeNode tmp; /* used to find place for left subtree */

    /* what if node is null? */

    if (getRight(node) == NULL)
        /* attach left subtree in place of deleted node */
        return(getLeft(node));
    else if (getLeft(node) == NULL)
        /* attach right subtree in place of deleted node */
        return(getRight(node));

    /* cont... */
}
```

CIS*2520 (506)

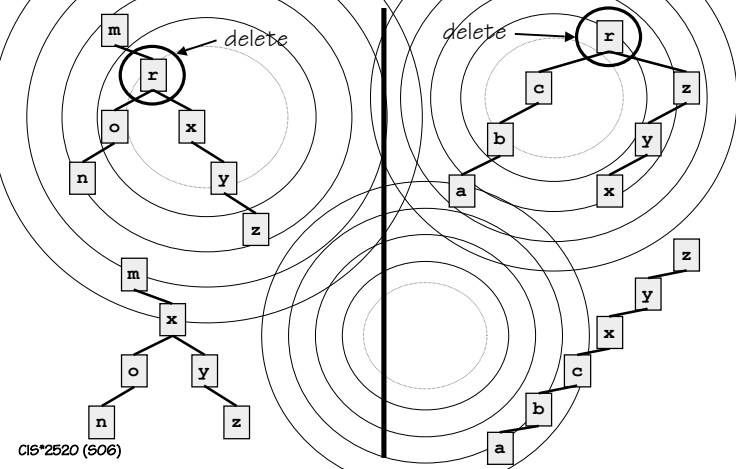
Deletion Implementation (cont.)

```
else /* neither subtree is empty */
{
    /* go to farthest left of the right subtree of the
    deleted node: this is attachment point for left
    subtree of deleted node --- attach right subtree of
    deleted node in place of the deleted node */
    tmp = getRight(node);
    while (getLeft(tmp) != NULL)
        tmp = getLeft(tmp);
    setLeft(tmp, getLeft(node));
    return(getRight(node));
} /* removeNode */
```

- Consider carefully how these functions are working in terms of the assumed means of invocation, $root = \text{function}(root)$, in order to ensure consistency

CIS*2520 (506)

Deletion Examples



CIS*2520 (506)

Properties of Trees (algorithms)

- Determine height of a binary tree (not necessarily a search tree)
 - recall: height (or depth) is the length of the longest path from the root to a leaf
 - **BASE CASE:**
 - empty tree - height is 0
 - **INDUCTION STEP:**
 - height is 1 + greater of height of left and right subtrees
- Note: we'll simplify this by defining a function "max"
 - what about equality?

CIS*2520 (506)

Height Implementation

```
int max(int v1, int v2)
{
    return((v1 < v2) ? v2 : v1);
} /* max */

int _height(TreeNode *root)
{
    if (root == NULL)
        return(0);
    else
        return(1 + max(_height(getLeft(root)),
                       _height(getRight(root))));
} /* _height */
```

CIS*2520 (506)

Other Property Based Operations

- Assuming all data values are integers, write a function to determine the sum of all data values in the tree
- Write a function to count the number of leaves in the tree
- Write a function that returns the length of the shortest path from the root to a leaf node
- All of these are not hard to implement recursively
 - you must take care defining the induction steps in terms of what you need to get done (and when it should be done)

CIS*2520 (506)

Tree Balance and Height

- Dynamic construction of BSTs is very sensitive to the order of the keys
 - best case depth $O(\log_2 n)$
 - worst case is $O(n)$!
- Height is ultimately related to "balance" of the tree
 - how well distributed are the keys throughout the tree
 - perfect balance implies no more than $\log_2 n$ comparisons are needed for any search

CIS*2520 (506)

Tree Balance and Height (cont.)

- Some numbers...
 - assume the $N!$ possible orderings of N keys are equally likely
 - average number of comparisons needed in the average BST with N nodes is approximately $2\log_2 n = 2(\log_2 2)(\log_2 n)$
 - average BST requires approximately $2\ln 2$ (≈ 1.39) times as many comparisons as a completely balanced tree
 - so average cost of not balancing a BST is approximately 39% more comparisons --- is this important?
 - not necessarily true that all key orders are equally likely
 - what is the expectation of “worst case” or “poor case”?
 - is optimality an issue? --- must balance cost of maintaining a balanced tree vs cost of additional comparisons (cost in terms of compute time, but also programming time)

CIS*2520 (506)

AVL Trees

- Russian mathematicians Adelson-Veskil and Landis (AVL) - 1962
 - achieves goal that searches, insertions and deletions in a tree with N nodes can all be achieved in time that is $O(\log_2 n)$, even in the worst case
 - in most cases, actual length of search is nearly $\log_2 n$
 - behaviour of AVL tree closely approximates that of a completely balanced tree

CIS*2520 (506)

AVL Tree Definition

- An AVL Tree is a binary search tree in which the heights of the left and right subtrees of the root differ by at most 1, and in which the left and right subtrees are again AVL trees
- Balance Factor
 - each node in an AVL tree has an associated balance factor which is
 - **LEFT HIGH** : left subtree has height greater than right subtree
 - **EQUAL** : left and right subtrees have equal height
 - **RIGHT HIGH** : right subtree has height greater than left subtree
 - in drawings we indicate these as ‘/’, ‘\’, ‘\’ respectively
- Does this definition imply that all leaves are on the same or adjacent levels?

CIS*2520 (506)

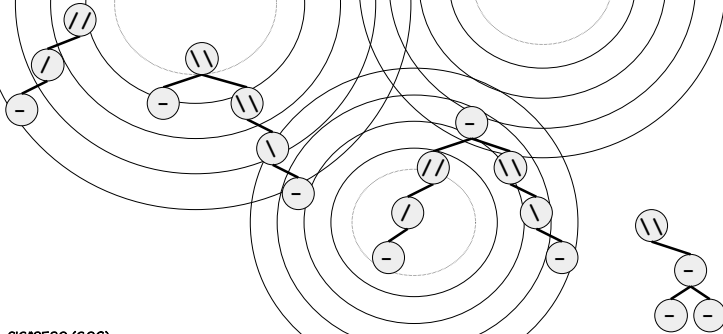
AVL Tree Examples



CIS*2520 (506)

AVL Tree Examples (cont.)

The following are not AVL trees:



CIS*2520 (506)

AVL Tree Implementation

- First we modify the node type to incorporate the balance factor

```
typedef enum { LHIGH, EQUAL, RHIGH } balance_t;

typedef struct TreeNode_t
{
    KeyType *key;
    DataType *data;
    TreeNode *left;
    TreeNode *right;
    balance_t bf;      /* balance factor of this node */
} TreeNode;

...

balance_t getBF() { ... }
```

CIS*2520 (506)

AVL Tree Implementation (cont.)

- We also need some additional initialization in the create operation

```
...

TreeNode *TreeNodeCreate(KeyType *keyval, DataType *dataval)
{
    key = keyval;
    data = dataval;
    left = right = NULL;
    bf = EQUAL;
} /* TreeNodeCreate */

...
```

CIS*2520 (506)

Insertion Into an AVL Tree

- Generally:
 - use the usual binary search tree insertion algorithm
 - compare new key to that in the root and insert into left or right subtree as appropriate
 - need to track when an insertion increases the height of a subtree (i.e. change balance factor)
 - CASES:
 - height of subtree does not change* (do nothing)
 - shorter subtree grows* (only balance factor of the root changes)
 - new node added to subtree of the root that is strictly taller than the other subtree* (height of subtree has increased --- need to FIX tree to restore balance)
- e.g. consider growth of AVL tree, with balance factors, on addition of the following (in order):
 - k, t, e, v, p, a, m, u, h

CIS*2520 (506)

Insertion Into an AVL Tree (cont.)

- If new node inserted into taller subtree of the root and height increases
 - subtree has height 2 more than the other
 - tree no longer satisfies AVL constraints
 - must rebuild part of tree to restore balance
- Require additional subroutine to perform rebalancing
 - this will be different depending on whether we are balancing with respect to a left or right subtree
- Need additional boolean parameter to permit communication of rebalancing information up recursive chain
 - change in height is with respect to a given subtree root -- may be distant from point of origin
 - for expedience we will use an integer value parameter (other options?)

CIS*2520 (506)

AVL Insertion Implementation

```
TreeNode
*_insertAVL(TreeNode *node, KeyType *key, DataType *data,
            int *taller)
{
    if (root == NULL)
    {
        *taller = 1;
        return (TreeNodeCreate(key, data));
    }
    else if (lessThan(key, getKey(root)))
    {
        setLeft(root, *_insertAVL(getLeft(root),key,data,taller));

        if (*taller) /* left subtree is taller */
        {
            switch(getBF(root))
            {
```

CIS*2520 (506)

AVL Insertion Implementation (cont.)

```
        LHIGH: /* node was already left high */
            root = _leftBalance(root);
            *taller = 1;
            break;
        EQUAL: /* node is now left high */
            setBf(root, LHIGH);
            break;
        RHIGH: /* node now has balanced height */
            setBf(root, EQUAL);
            *taller = 0;
            break;
    }
}
else if (greaterThan(key, getKey(root)))
{
    setRight(root, *_insertAVL(getRight(root),key,data,taller));

    if (*taller) /* right subtree is taller */
    {
```

CIS*2520 (506)

AVL Insertion Implementation (cont.)

```
        switch(getBF(root))
        {
            LHIGH: /* node now has balanced height */
                setBf(root,EQUAL);
                *taller = 0;
                break;
            EQUAL: /* node is now right high */
                setBf(root, RHIGH);
            RHIGH: /* node was already right high */
                root = _rightBalance(root);
                *taller = 0;
                break;
        }
    }
}
else
    /* duplicate key */
} /* *_insertAVL */
```

CIS*2520 (506)

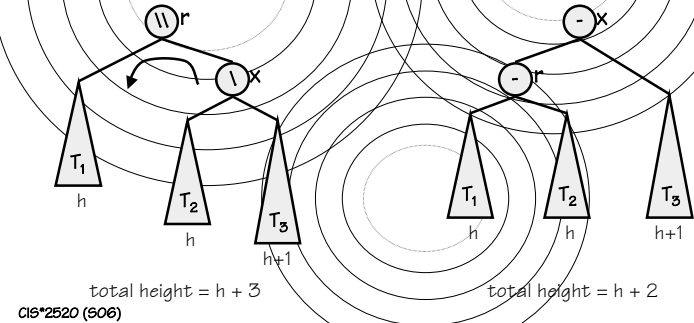
Balance-Right: Rotation Operation

- Consider
 - new node inserted into right subtree
 - height has increased
 - original tree was right high
 - i.e. this is the case covered by `balanceRight()` function
- Let r be the root of the tree and x be the root of the right subtree
- There are 3 cases depending on the balance factor of x

CIS*2520 (506)

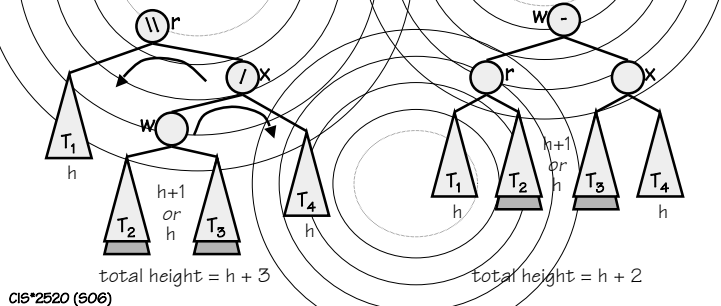
Case 1: x is RHIGH

- Requires LEFT ROTATION
 - resulting height decreases by 1 (height imbalance is redressed)
 - adjustment of balancing is "in general" -- do explicitly in implementation



Case 2: x is LHIGH

- Requires DOUBLE ROTATION
 - necessary to move 2 levels, to the node w that roots the left subtree of x , to find the new root
 - rotate subtree rooted at x to the right, making w the root of that subtree, then rotate the subtree rooted at r to the left, making w the new root of the whole subtree



Case 2: x is LHIGH (cont.)

- What about resulting balance factors?
 - new balance factors for r and x depend on previous balance factor for w
 - the previous slide suggests that the subtrees of w have equal height, however it is possible that w may be either left or right high

old w	new r	new x
-	-	-
/	-	\
\	/	-

CIS*2520 (506)

Case 3: x is EQUAL

Can never happen

- recall: just inserted a new node into subtree rooted at x
 - this subtree now has height 2 more than left subtree of the root (r)
 - new node either went into the left or right subtree of x , thus its insertion increased the height of only one subtree of x
 - if these subtrees had equal height after the insertion then the height of the full subtree rooted at x was not changed by the insertion which is a contradiction

Note (In General):

- after a "balance" operation, the increase in height induced by the insertion is undone (no further rotations necessary)
- thus only one balance operation is ever needed

CIS*2520 (506)

Left Rotation Implementation

```
/*
 * -----
 *  * _leftRotate
 *  *   - rotate a binary tree to the left
 *  * -----
 *  * PRE:
 *  *   right subtree of the root is non-empty
 *  * -----
 *  * POST:
 *  *   returns the new root of the subtree, the former
 *  *   root's right subtree; former root's right subtree
 *  *   is the left subtree of the former root's right
 *  *   subtree; new root's left subtree is the former root
 *  * -----
 *  * ERRORS:
 *  *   (right subtree of the root is empty)
 *  * -----
 */
```

CIS*2520 (506)

Left Rotation Implementation (cont.)

```
TreeNode *_leftRotate(TreeNode *root)
{
    TreeNode *tmp;

    if (getRight(root) == NULL)
        /* ERROR - can't rotate an empty subtree */
    else
    {
        tmp = getRight(root);
        setRight(root, getLeft(tmp));
        setLeft(tmp, root);
        return tmp;
    }
}

/* _leftRotate */
```

CIS*2520 (506)

Right Balance Implementation

```
/*
 * -----
 *  * _rightBalance
 *  *   - rebalance an AVL tree that is right heavy
 *  * -----
 *  * PRE:
 *  *   balance factor of right subtree of the root is not
 *  *   EQUAL
 *  * -----
 *  * POST:
 *  *   returns the new root of the subtree, which now
 *  *   satisfies the constraints of an AVL tree
 *  * -----
 *  * ERRORS:
 *  *   (right subtree of the root has EQUAL balance)
 *  * -----
 */
```

CIS*2520 (506)

Right Balance Implementation (cont.)

```
TreeNode *_rightBalance(TreeNode *root)
{
    switch(getBf(getRight(root)))
    {
        RHIGH: /* single rotation left */
        setBf(root, EQUAL);
        setBf(getRight(root), EQUAL);
        return(_leftRotate(root));
        break;
        LHIGH: /* double rotation left */
        switch(getBf(getLeft(getRight(root))))
        {
            RHIGH:
            setBf(root, LHIGH);
            setBf(getRight(root), EQUAL);
            break;
        }
    }
}
```

CIS*2520 (506)

Right Balance Implementation (cont.)

```
        EQUAL:
        setBf(root, EQUAL);
        setBf(getRight(root), EQUAL);
        break;
        LHIGH:
        setBf(root, EQUAL);
        setBf(getRight(root), RHIGH);
        break;
    }

    setBf(getLeft(getRight(root)), EQUAL);
    setRight(root, _rightRotate(getRight(root)));
    return(_leftRotate(root));
    break;
    EQUAL:
    /* ERROR - right subtree cannot have equal balance */
}
} /* _rightBalance */
```

CIS*2520 (506)

Deletion From an AVL Tree

- Similar idea to insertion
 - standard BST deletion will often be sufficient
 - where height changes sufficiently, rotations will be required to preserve balance
- i.e. consider the cases, etc.
 - slightly more complex than insertions (more cases to consider)
- We will not discuss this further at this point, however you are encouraged to consider AVL trees more fully when you have the interest or need

CIS*2520 (506)

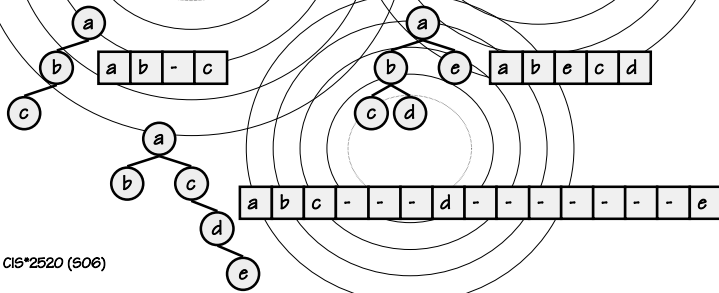
AVL Tree Notes

- Some numbers
 - worst case height for sparsest possible AVL tree
 - approx $1.44 \log_2 n$ (comparable to average case for vanilla BST)
 - worst case AVL tree is uncommon
 - empirical studies suggest average number of comparisons to be about $\log_2 n + 0.25$
 - difficult to approach analytically

CIS*2520 (506)

Contiguous Representation of Binary Trees

- Similar to contiguous representation of linked lists
- Use an array (as before, 2 general approaches)
 1. structure using integers as indirection to indices of right/left subtrees
 2. for node at index k , left/right subtrees are at $2k+1$, $2k+2$
 - positions beyond end of the array do not exist
 - must mark empty subtree positions



Other Tree Structures (see lab)

- Tries
 - fast alphabetic storage and searching
- N-ary Trees
 - N subtrees at each node
- B-Trees
 - external/indexed searching
- etc.

CIS*2520 (506)