AVL TREES

- Binary Search Trees
- AVL Trees
Binary Search Trees

- A binary search tree is a binary tree $T$ such that
  - each internal node stores an item $(k, e)$ of a dictionary.
  - keys stored at nodes in the left subtree of $v$ are less than or equal to $k$.
  - Keys stored at nodes in the right subtree of $v$ are greater than or equal to $k$.
  - External nodes do not hold elements but serve as place holders.

```
    44
   /   \
  17    88
 /  \
32   97
 /  \
28  54
  /  \
29  76
  |  /\n  |  80
   |  /\n   |  82
```

AVL Trees
Search

• The binary search tree $T$ is a decision tree, where the question asked at an internal node $v$ is whether the search key $k$ is less than, equal to, or greater than the key stored at $v$.

• Pseudocode:

  Algorithm TreeSearch($k$, $v$):
  
  Input: A search key $k$ and a node $v$ of a binary search tree $T$.
  
  Output: A node $w$ of the subtree $T(v)$ of $T$ rooted at $v$, such that either $w$ is an internal node storing key $k$ or $w$ is the external node encountered in the inorder traversal of $T(v)$ after all the internal nodes with keys smaller than $k$ and before all the internal nodes with keys greater than $k$.

  if $v$ is an external node then
    return $v$
  
  if $k = \text{key}(v)$ then
    return $v$
  
  else if $k < \text{key}(v)$ then
    return TreeSearch($k$, $T$.leftChild($v$))
  
  else
    \{ $k > \text{key}(v)$ \}
  
  return TreeSearch($k$, $T$.rightChild($v$))
Search (cont.)

- A picture:
  
  ![AVL Tree Diagram]

**find(25)**

**find(76)**
Insertion in a Binary Search Tree

• Start by calling $\text{TreeSearch}(k, T.\text{root}())$ on $T$. Let $w$ be the node returned by $\text{TreeSearch}$

• If $w$ is external, we know no item with key $k$ is stored in $T$. We call $\text{expandExternal}(w)$ on $T$ and have $w$ store the item $(k, e)$

• If $w$ is internal, we know another item with key $k$ is stored at $w$. We call $\text{TreeSearch}(k, \text{rightChild}(w))$ and recursively apply this algorithm to the node returned by $\text{TreeSearch}$. 
• Insertion of an element with key 78:

a)

b)
Removal from a Binary Search Tree

• Removal where the key to remove is stored at a node (w) with an external child:
Removal from a Binary Search Tree (cont.)

(b)
Removal from a Binary Search Tree (cont.)

- Removal where the key to remove is stored at a node whose children are both internal:

![Diagram of a binary search tree with nodes labeled 17, 28, 29, 54, 82, 76, 80, 78, 44, 88, 97, and 65. The node labeled 76 is the node whose children are both internal, and the removal is illustrated with dashed lines and an arrow labeled w.](image-url)
Removal from a Binary Search Tree (cont.)

(b)
Time Complexity

• Searching, insertion, and removal in a binary search tree is $O(h)$, where $h$ is the height of the tree.

• However, in the worst-case search, insertion, and removal time is $O(n)$, if the height of the tree is equal to $n$. Thus in some cases searching, insertion, and removal is no better than in a sequence.

• Thus, to prevent the worst case, we need to develop a rebalancing scheme to bound the height of the tree to log $n$. 
AVL Tree

- An AVL Tree is a binary search tree such that for every internal node \( v \) of \( T \), the heights of the children of \( v \) can differ by at most 1.

- An example of an AVL tree where the heights are shown next to the nodes:
Height of an AVL Tree

- **Proposition**: The height of an AVL tree $T$ storing $n$ keys is $O(\log n)$.

- **Justification**: The easiest way to approach this problem is to try to find the minimum number of internal nodes of an AVL tree of height $h$: $n(h)$.

  - We see that $n(1) = 1$ and $n(2) = 2$
  - for $n \geq 3$, an AVL tree of height $h$ with $n(h)$ minimal contains the root node, one AVL subtree of height $n-1$ and the other AVL subtree of height $n-2$.
    - i.e. $n(h) = 1 + n(h-1) + n(h-2)$
    - Knowing $n(h-1) > n(h-2)$, we get $n(h) > 2n(h-2)$
      - $n(h) > 2n(h-2)$
      - $n(h) > 4n(h-4)$
      - ...
      - $n(h) > 2^i n(h-2i)$
    - Solving the base case we get: $n(h) \geq 2^{h/2-1}$
  - Taking logarithms: $h < 2\log n(h) + 2$
  - Thus the height of an AVL tree is $O(\log n)$
Insertion

• A binary search tree $T$ is called balanced if for every node $v$, the height of $v$’s children differ by at most one.

• Inserting a node into an AVL tree involves performing an expandExternal$(w)$ on $T$, which changes the heights of some of the nodes in $T$.

• If an insertion causes $T$ to become unbalanced, we travel up the tree from the newly created node until we find the first node $x$ such that its grandparent $z$ is unbalanced node.

• Since $z$ became unbalanced by an insertion in the subtree rooted at its child $y$, $\text{height}(y) = \text{height}($sibling$(y)) + 2$

• To rebalance the subtree rooted at $z$, we must perform a restructuring
  - we rename $x$, $y$, and $z$ to $a$, $b$, and $c$ based on the order of the nodes in an in-order traversal.
  - $z$ is replaced by $b$, whose children are now $a$ and $c$ whose children, in turn, consist of the four other subtrees formerly children of $x$, $y$, and $z$. 
Insertion (contd.)

- Example of insertion into an AVL tree.

```plaintext
          44
         5
        / 
       2   1
      17   32
     /     1
    48
   / 
  50
 /  
48 62
/  
1 54 88
T0  T1  T2  T3

T0  T1  T2  T3

x  y  z

1 2 3
```

AVL Trees 15
Restructuring

• The four ways to rotate nodes in an AVL tree, graphically represented:

  - Single Rotations:

![Diagram of single rotations in AVL trees]

- Single rotation
- Single rotation
Restructuring (contd.)

- double rotations:

- Double rotation:
  - $T_0$
  - $a = z$
  - $b = x$
  - $c = y$
  - $T_2$
  - $T_3$

- Double rotation:
  - $T_0$
  - $a = z$
  - $b = x$
  - $c = y$
  - $T_2$
  - $T_3$

- Double rotation:
  - $T_3$
  - $a = y$
  - $b = x$
  - $c = z$
  - $T_2$
  - $T_1$

- Double rotation:
  - $T_3$
  - $a = y$
  - $b = x$
  - $c = z$
  - $T_1$
  - $T_0$
Restructuring (contd.)

• In Pseudo-Code:

**Algorithm** `restructure(x):`

Input: A node \(x\) of a binary search tree \(T\) that has both a parent \(y\) and a grandparent \(z\)

Output: Tree \(T\) restructured by a rotation (either single or double) involving nodes \(x, y,\) and \(z\).

1. Let \((a, b, c)\) be an inorder listing of the nodes \(x, y,\) and \(z\), and let \((T_0, T_1, T_2, T_3)\) be an inorder listing of the four subtrees of \(x, y,\) and \(z\) not rooted at \(x, y,\) or \(z\)

2. Replace the subtree rooted at \(z\) with a new subtree rooted at \(b\)

3. Let \(a\) be the left child of \(b\) and let \(T_0, T_1\) be the left and right subtrees of \(a\), respectively.

4. Let \(c\) be the right child of \(b\) and let \(T_2, T_3\) be the left and right subtrees of \(c\), respectively.
Removal

- We can easily see that performing a \textit{removeAboveExternal}(w) can cause \( T \) to become unbalanced.

- Let \( z \) be the first \texttt{unbalanced} node encountered while travelling up the tree from \( w \). Also, let \( y \) be the child of \( z \) with the larger height, and let \( x \) be the child of \( y \) with the larger height.

- We can perform operation \textit{restructure}(x) to restore balance at the subtree rooted at \( z \).

- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of \( T \) is reached.
Removal (contd.)

- example of deletion from an AVL tree:
Removal (contd.)

- example of deletion from an AVL tree
Implementation

• A Java-based implementation of an AVL tree requires the following node class:

```java
public class AVLItem extends Item {
    int height;

    AVLItem(Object k, Object e, int h) {
        super(k, e);
        height = h;
    }

    public int height() {
        return height;
    }

    public int setHeight(int h) {
        int oldHeight = height;
        height = h;
        return oldHeight;
    }
}
```
public class SimpleAVLTree
    extends SimpleBinarySearchTree
    implements Dictionary {

    public SimpleAVLTree(Comparator c) {
        super(c);
        T = new RestructurableNodeBinaryTree();
    }

    private int height(Position p) {
        if (T.isExternal(p))
            return 0;
        else
            return ((AVLItem) p.element()).height();
    }

    private void setHeight(Position p) { // called only
        // if p is internal
        ((AVLItem) p.element()).setHeight
            (1 + Math.max(height(T.leftChild(p)),
                height(T.rightChild(p))));
    }
}
private boolean isBalanced(Position p) {
    // test whether node p has balance factor
    // between -1 and 1
    int bf = height(T.leftChild(p)) - height(T.rightChild(p));
    return (-1 <= bf) && (bf <= 1);
}

private Position tallerChild(Position p) {
    // return a child of p with height no
    // smaller than that of the other child
    if(height(T.leftChild(p)) >= height(T.rightChild(p)))
        return T.leftChild(p);
    else
        return T.rightChild(p);
}
private void rebalance(Position zPos) {
    //traverse the path of T from zPos to the root;
    //for each node encountered recompute its
    //height and perform a rotation if it is
    //unbalanced
    while (!T.isRoot(zPos)) {
        zPos = T.parent(zPos);
        setHeight(zPos);
        if (!isBalanced(zPos)) { // perform a rotation
            Position xPos = tallerChild(tallerChild(zPos));
            zPos = ((RestructurableNodeBinaryTree) T).restructure(xPos);
            setHeight(T.leftChild(zPos));
            setHeight(T.rightChild(zPos));
            setHeight(zPos);
        }
    }
}
public void insertItem(Object key, Object element) throws InvalidKeyException {
    super.insertItem(key, element); // may throw an
    // InvalidKeyException
    Position zPos = actionPos; // start at the
    // insertion position
    T.replace(zPos, new AVLItem(key, element, 1));
    rebalance(zPos);
}

public Object remove(Object key) throws InvalidKeyException {
    Object toReturn = super.remove(key); // may throw
    // an InvalidKeyException
    if (toReturn != NO_SUCH_KEY) {
        Position zPos = actionPos; // start at the
        // removal position
        rebalance(zPos);
    }
    return toReturn;
}