Binary Search Trees

- Definition
  - All keys are distinct – why?
  - Key in the root of the left subtree is less than the root.
  - Key in the root of the right subtree is greater than the root.
  - Left and right subtrees are binary search trees.
- Figure 1 is an example:

![Binary Search Tree Diagram](image)

- Where would you add X, B, and E?
- At seats, build tree from the nodes: B R L M T C A N P D
- What about deletion?
  - Since it is easier to delete from a leaf, we first replace the node with its inorder successor, and then delete the node that contained the inorder successor.
- The code for insertion is as follows. I like to create the new node outside the insert routine.

```cpp
template <class Etype>
class BinarySearchTree
{
  protected:
    void makeEmpty( TreeNode<Etype> * & t );
    void insert( const Etype & x, TreeNode<Etype> * & t );
    void printtree( TreeNode<Etype> * t , int indent) const;
    TreeNode<Etype> * copy( const TreeNode<Etype> * t );
    TreeNode<Etype> * root;
    TreeNode<Etype> * lastFind;

  public:
    BinarySearchTree( ) : root( NULL ), lastFind( NULL ) { }
    virtual ~BinarySearchTree( )
    { makeEmpty( root ); lastFind = NULL; }
    const BinarySearchTree &
    operator = ( const BinarySearchTree & Value );
};
```
void makeEmpty( ) { makeEmpty( root ); lastFind = NULL; }
void printtree( ) const { printtree( root,1 ); }
virtual void insert( const Etype & x ) { insert( x, root ); }

template <class Etype>
void BinarySearchTree<Etype>::
insert( const Etype & x, TreeNode<Etype> * & t )
{   if( t == NULL )
    { t = new TreeNode<Etype>( x );
      if( t == NULL ) cout <<"Out of space\n";
    }
    else if( x < t->Element )
      insert( x, t->left );
    else if( x > t->Element )
      insert( x, t->right );
    // Else x is in the tree already. We'll do nothing.
}

• Operations:
  o Search – $O(h)$, where $h$ is the height of the tree. In the best case (balanced), $h$ is $\log n$. In the worst case, $h$ is $n$ (the number of nodes in the tree).
  o Insert – $O(h)$, where $h$ is the height of the tree.
  o Delete – $O(h)$, where $h$ is the height of the tree.
    ▪ What makes deletion difficult?

Balanced Trees

• Binary search trees are a good technique for finding elements in a dictionary.
• But it has a disadvantage:
  o In the worst case, the operations are still $O(n)$.
• There are applications that are time critical, and this amount of time is not acceptable.
• We will consider balanced trees (AVL, red-black, and AA trees) where the worst case operation is $O(\log n)$.
• Go over recursion handout showing trace of recursion.

AVL Trees

• Balanced tree – the height of left subtree and height of right subtree differ by at most one.
• AVL trees are height balanced trees.
• The name comes from the people that came up with the idea – Adelson-Velskii and Landis.
• Definition:
  ▪ An empty binary tree is an AVL tree.
If $T$ is a nonempty binary tree with $T_L$ and $T_R$ as its left and right subtrees, then $T$

is an AVL tree iff

- $T_L$ and $T_R$ are AVL trees and
- $|h_L - h_R| \leq 1$ where $h_L$ and $h_R$ are the heights of $T_L$ and $T_R$, respectively.

- For several examples determine if AVL or not.
- How bad can a tree be and still be AVL?
- An AVL search tree is a binary search tree that is also an AVL tree.
- See Figure 2 for some possible examples. For each tree:
  - Is it an AVL tree?
  - Is it an AVL search tree?

![Figure 2 Possible AVL Trees and AVL Search Trees](image)

- Properties of AVL trees:
  - The height of an AVL tree with $n$ elements/node is $O(\log n)$.
  - For every value of $n$, $n \geq 0$, there exists an AVL tree.
  - An $n$-element AVL search tree can be searched in $O(\text{height}) = O(\log n)$ time.
  - A new element can be inserted into an $n$-element AVL search tree so that the result is an $n+1$ element AVL tree and such an insertion can be done in $O(\log n)$ time.
  - An element can be deleted from an $n$-element AVL search tree so that the result is an $n-1$ element AVL tree and such a deletion can be done in $O(\log n)$ time.

- From here on out, when I refer to an AVL tree, I mean an AVL search tree.
- The bound in the height of an AVL tree can be shown by the recursive routine shown below:
  - worst_tree (h) = /* Worst tree of height h */
  - root left = worst_tree(h-1)
  - root right = worst_tree(h-2)

Searching

- How is this done? (Same way as with the binary search tree.)

Insertion

- Insertions – just like BST, find the leaf where the node needs to go. After that node is placed in the AVL tree, the tree is rebalanced if necessary.
- We need to have a balance factor associated with each node $x$ in the AVL tree.
height of left subtree of $x$ – height of right subtree of $x$

- See Figure 3 for an example of inserting 32 into an AVL tree and the subsequent rebalancing. Notice that the balance factor is contained in each of the nodes shown in the tree.

Figure 3  Example of Inserting into an AVL-Tree

- Observations about the unbalanced tree that results from an insertion.
  - In the unbalanced tree, the balance factors are limited to $-2$, $-1$, $0$, $1$, and $2$.
  - A node with balance factor $2$ has a balance factor of $1$ before the insertion. Similarly, a node with balance factor $-2$, has a balance factor of $-1$ before the insertion.
  - The balance factors of only those nodes on the path from the root to the newly inserted node can change as a result of the insertion.
  - Let $A$ denote the nearest ancestor of the newly inserted node whose balance factor is either $-2$ or $2$. The balance factor of all nodes on the path from $A$ to the newly inserted node was $0$ prior to the insertion.

- In Figure 2(b), what is $A$? (Node 40.)

- There are four cases have to be consider when inserting into an AVL tree. We categorize these starting from $A$.
  - Inserted node is in the left subtree of $A$.
    - New node is a left subtree of the left subtree of $A$ – call LL rotation.
    - New node is a right subtree of the left subtree of $A$ – called LR rotation.
  - Inserted node is in the right subtree of $A$.
    - New node is a left subtree of the left subtree of $A$ – call RL rotation.
    - New node is a right subtree of the left subtree of $A$ – called RR rotation.

- The RR and RL rotations are symmetric to the LL and LR so we don’t need to discuss them.

- A generic LL type imbalance appears in Figure 4.
• A generic LR type imbalance appears in Figure 5.

• The balance factor $b$ in the tree is defined below:
  $b = 0 \Rightarrow bf(B) = bf(A) = 0$ after rotation.
  $b = 1 \Rightarrow bf(B) = 0$ and $bf(A) = -1$ after rotation.
  $b = -1 \Rightarrow bf(B) = 1$ and $bf(A) = 0$ after rotation.

• The complexity of insertion into an AVL tree is $O(\text{height}) = O(\log n)$.

• Notice that a single rotation is sufficient to restore balance if the insertion causes imbalance.

• Consider the tree in Figure 6.
  - Insert 70
  - Insert 22
Deletion

- Deletions: Replace with inorder successor, delete, and rebalance.
- An R0 imbalance at A is rectified by performing the rotation shown in Figure 7.

![Figure 7 R0 Rotation](image)

- Figure 8 shows how to handle an R1 imbalance.

![Figure 8 R1 Rotation](image)
• Following an R1 rotation, we must continue to examine nodes on the path to the root. *Unlike the case of an insertion, one rotation may not suffice to restore balance following a deletion. The number of rotations needed is* $O(\log n)$.

• The transformation needed when the imbalance is of type R–1 appears in Figure 9.

![Figure 9 R–1 Rotation](image)

• R-1 rotations may possibly need to continue to the root.
• See the LR rotation for the definition of b.
• What happens if AVL property is extended so left and right differ by 2 or 3 or $K$?
  o Termed height balanced tree, HB[$K$].
  o As $K$ increases, worst case height increases, but time to rebalance is less.
  o There are symmetric rotations
• Consider the tree shown in Figure 10. We want to delete node 60

![Figure 10 Delete node 60](image)

• Consider the tree shown in Figure 11. We want to delete node 70.
Rotations

- LL and R1 rotations are identical.
- LL and R0 rotations differ only in the final balance factors.
- LR and R–1 rotations are identical.

Red-Black Trees

- A red-black tree is a binary search tree having the following ordering properties:
  - Every node is colored either red or black
  - The root is black
  - If a node is red, its children must be black
  - Every path from a node to a NULL pointer must contain the same number of black nodes
- Figure 12 shows an example of a red-black tree
  - Notice the properties of a red-black tree in this figure
- If every path from the root to a NULL node contains $B$ black nodes, the tree must contain at least $2^B - 1$ black nodes
- If the root is black and there cannot be two consecutive red nodes on a path, the height of a red-black tree is at most $2 \log(N+1)$
- Thus, searching is guaranteed to be a logarithmic operation
• We need to define rotations to keep the tree balanced
  o The single rotation is shown in Figure 13

![Figure 13 Red-black Single Rotation](image)

  o The double rotation is shown in Figure 14

![Figure 14 Red-black Double Rotation](image)

• To avoid the possibility of having to rotate up a tree, we apply a top-down procedure as we are searching for the insertion point
  o We guarantee that, when we arrive at a leaf and insert a node, S is not red
  o Then we can just add a red leaf and if necessary use one rotation (either single or double).
• On the way down, if a node X has two red children, we make X red and its two children black
  o The number of black nodes on the path remains the same
  o If X’s parent is red, we would introduce two consecutive red nodes
  o But we apply either a single or a double rotation as seen in Figure 13 and Figure 14 above
• Consider the following example
  o Add 10, 85, 15 to a red-black tree (see Figure 15)

![Figure 15 Insert 10, 85 15](image)

  o Add 70, 20, 60 (see Figure 16)

![Figure 16 Insert 70, 20, 60](image)
- Add 30 (see Figure 17)
  ![Figure 17 Insert 30]

- Add 50 (see Figure 18)
  ![Figure 18 Insert 50]

- Add 65, 80, 90 (see Figure 19)
  ![Figure 19 Insert 65, 80, 90]

- Add 40 (see Figure 20)
  ![Figure 20 Insert 40]

- Add 5, 55 (see Figure 20)
AA-Trees

- Red-black trees have many different rotations making the implementation fairly tricky.
- The delete operation is also difficult.
- AA-trees are the method of choice when a balanced tree is needed, a casual implementation is acceptable, and deletions are needed.
- The AA-tree adds one extra condition to the red-black tree.
  - Left children may not be red.
- This simplifies things because:
  - Eliminates about half of the restructuring cases.
  - It simplifies the deletion algorithm by removing annoying cases.
- The implementation maintains a level of a node in each node of the tree rather than color (red, black).
- The level of a node represents the number of left links on a the path to the empty tree and is defined as follows:
  - Level 1, if the node is a leaf.
  - The level of its parent, if the node is red.
  - One less than the level of its parent, if the node is black.
- From the above, we know that:
  - Left child must be one level lower than its parent.
  - Right child may be zero or one level lower than it parent (but not more).
- A horizontal link is a connection between a node and a child of equal levels.
- The coloring properties imply that:
  - Horizontal links are right links (because only right children may be red).
  - There may not be two consecutive horizontal links (since there cannot be consecutive red nodes).
  - Nodes at level 2 or higher must have two children.
  - If a node does not have a right horizontal link, its two children are at the same level.
- Figure 22 shows a sample AA-tree.
  - The tree results from the insertion of 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55, and 35.
**Insertion**

- A new item is always inserted at the bottom level.
- In the previous example, inserting 2 will create a horizontal left link.
- In the previous example, inserting 45 generates consecutive right links.
- After inserting at the bottom level, we may need to perform rotations to restore the horizontal link properties.
- To remove a left horizontal link we perform a skew as shown in Figure 23.

**Figure 23 Skew Rotation**

- To remove two consecutive horizontal links we perform a split as shown in Figure 24.

**Figure 24 Split Rotation**

- Skew and split:
  - A skew removes a left horizontal link.
  - A skew might create consecutive right horizontal links.
  - We should first process a skew and then a split, if necessary.
  - After a split, the middle node increases a level, which may create a problem for the original parent.
- Insert a 45 into the AA-tree in Figure 22. When this happens, consecutive horizontal links form. Thus a split is performed and the result is shown in Figure 25.
• After a skew at 50, consecutive horizontal nodes are introduced starting at 40. This is shown in Figure 26.

• After split at 40, 50 is now on the same level as 70, inducing an illegal left horizontal link. This is shown in Figure 27.

• After a skew at 70, consecutive horizontal links are introduced starting at node 30. This is shown in Figure 28.

• After split at 30, the insertion is complete as shown in Figure 29.
Figure 29  After split at 30