Today

Data Structures: Search Trees
- Binary Search Trees,
- Non-Binary Search Trees,
- Balanced Trees,
- 2-3 Trees.

Revised 10/29/07; See Page 4.
**Dynamic Sets**

**Operations**

- **CREATE()**: Creates empty set
- **SEARCH(x, S)**: Returns element with key $x$, in $S$ if such entry exists;
- **Succ(x, S)**: Returns el't with key $> x$;
- **Pred(x, S)**: Returns el't with key $< x$;
- **INSERT(x, S)**: Adds el't with key $x$ to $S$;
- **DELETE(x, S)**: Deletes el't with key $x$;

Would like all operations in time $f(n)$

where $n = |S|$.

What is the best $f(n)$?
Naive Solutions

Sorted Array

SEARCH = O(log n)

INSERT = O(n) ←

Sorted Linked List

SEARCH = O(n)

INSERT = O(n) + SEARCH time

Heap

SEARCH = O(n)

Can we do better than the above? This lecture & next ... O(lg n).
**Binary Search Tree**

- Rooted tree where every node has 0/1 left child, and 0/1 right child.

- Every node has one key.

Search property: for every node

```
(key(L) < key(node)) 
(key(node) < key(R))
```

(REVISED) 10/29/07  
& every L in left subtree  
& every R in right subtree
Examples

```
       6
      / \
     3   8
    / \
   2   5
  / \  / \
 2   3 6   8
  / \
 5   6
```

```
       8
      / \
     3   11
    / \
   2   8
  / \
 5   6
```
Basic Algorithms

1. TREE-WALK (node);
   
   if node = NIL return;
   
   else TREE-WALK (left child);
   
   print (key);
   
   TREE-WALK (right child);

Print keys in tree in sorted tree;

2. SEARCH (node, x);

   if node = NIL return (NOT FOUND);
   
   else if key(node) = x return (node);
   
   else if key(node) < x SEARCH (right child);
   
   else SEARCH (left child);
\textbf{INSERT} \((x, \text{node})\) \\
\text{Tree with } x \leftarrow \text{ in it;}
\begin{align*}
\text{if } & \text{ node = NIL return } (\text{NIL}); \\
\text{else if } & \text{ key(node) = } x \text{ then return } (\text{"ERROR");} \\
\text{else if } & \text{ } x < \text{ key(node) then} \\
& \{ \text{ if leftchild = NIL then} \\
& \quad \text{add } x \leftarrow \text{ as leftchild;} \\
& \quad \text{else SEARCH } (x, \text{ leftchild}) \} \\
\text{else } & \{ \text{ similar stuff with right child } \} \\
\end{align*}
**Example**

- **Insert (10)**

```
  6
 / 
3   8
/ \
2   5
```

- **Successor = ?**
- **Predecessor = ?**
- **Delete = ?**
Good News: Running time of all algorithms
= $O(d)$ where $d = \text{depth}$.

Bad News: Without "maintenance", and
with worst-case sequence of INSERT's,
$\text{depth} = \Omega(n)$.

Random News: If INSERT's are random, then
$\text{depth} = \Theta(\log n)$ whp.

Better News: With some maintenance,
can ensure $d = \Theta(\log n)$.
Maintaining BST balance

Two approaches:

1. Keep trees Binary & Rotate

2. Allow flexibility in # children
   (e.g. 0/2/3 children - 2-3 tree)
   (0/2/3/4 children - 2-3-4 tree)
   (0/3/3+1/.../2B children - B-trees)

& SPLIT/MERGE nodes.
**Rotate** (for binary trees)

Recall Tree = Root + left subtree + right subtree

Subtree = Subroot + left subtree + right subtree

Say subtree is imbalanced if

\[ | \text{left height} - \text{right height} | > 1 \]
Let's look at $A$ further. Suppose the tree looks like this.

Then can LEFT-ROTATE($x$) to get...
Resulting tree is more balanced than before. Still has Search property.

**Theorem**: [AVL]: Can follow up on INSERT/DELETE with \textsc{Rotate} to maintain height balance at every node. Consequently, all ops run in time $\Theta(\log n)$. 
Approach 2 (Ours):

2-3 Trees + Split / Merge.

Definition of 2-3 tree

1. Every node has
   0 children (leaf)
   or 2 children or 3 children (internal)

2. Leaf stores 1 or 2 keys.

   Internal node stores 1 less key than
   # children.

3. All leaves at same level!
Example

- SEARCH/SUCC/PRED takes $\Theta(d)$ time.

- $d =$ ?

- Can we do INSERT/DELETE while maintaining the 2-3 tree property?
Depth of 2-3 Tree

- 2-3 tree of depth $d$
  has at least $2^d$ nodes

- 2-3 tree of depth $d$
  has at most $3^d$ nodes

- $2^d \leq n \leq 3^d$

$\Rightarrow \quad \log_3 n \leq d \leq \log_2 n$

$\Rightarrow \quad d = \Theta(\log n)$.

INSERT/DELETE: Next lecture.