Some Search Structures

- **Sorted Arrays**
  - Advantages: Search in $O(\log n)$ time (binary search)
  - Disadvantages:
    - Need to know size in advance
    - Insertion, deletion $O(n)$ – need to shift elements

- **Lists**
  - Advantages:
    - No need to know size in advance
    - Insertion, deletion $O(1)$ (not counting search time)
  - Disadvantages:
    - Search is $O(n)$, even if list is sorted

Balanced Search Trees

- **Best of both!**
  - Search, insert, delete in $O(\log n)$ time
  - No need to know size in advance

- **Several flavors**
  - AVL trees, 2-3 trees, red-black trees, skip lists, random treaps, ...

Review – Binary Search Trees

- Every node has a *left child*, a *right child*, both, or neither
- Data elements are drawn from a totally ordered set (e.g., Comparable)
- Every node contains one data element
- Data elements are ordered in *inorder*

A Binary Search Tree

In any subtree:
- all elements smaller than the element at the root are in the *left* subtree
- all elements larger than the element at the root are in the *right* subtree
Search

To search for an element x:

• if tree is empty, return false
• if x = object at root, return true
• if x < object at root, search left subtree
• if x > object at root, search right subtree

Example: search for 13
Search

boolean treeSearch(T x) { // T extends Comparable<T>
    int c = x.compareTo(datum);
    if (c == 0) return true; // found
    if (c < 0 && left != null)
        return left.treeSearch(x);
    return right != null && right.treeSearch(x);
}

Insertion

To insert an element x:
• search for x – if there, just return
• when arrive at a leaf y, make x a child of y
  – left child if x < y
  – right child if x > y

Example: insert 15

```plaintext
  6
 / \
/   \ 
1     25
 \    /  \
 20   29
  /   /  \
13 54 80
 /  /  / \
48 91 48 91
```

```plaintext
  6
 / \
/   \ 
1     25
 \    /  \
 20   29
  /   /  \
13 54 80
 /  /  / \
48 91 48 91
```

```plaintext
  6
 / \
/   \ 
1     25
 \    /  \
 20   29
  /   /  \
13 54 80
 /  /  / \
48 91 48 91
```

```plaintext
  6
 / \
/   \ 
1     25
 \    /  \
 20   29
  /   /  \
13 54 80
 /  /  / \
48 91 48 91
```

```plaintext
  6
 / \
/   \ 
1     25
 \    /  \
 20   29
  /   /  \
13 54 80
 /  /  / \
48 91 48 91
```
**Insertion**

void insert(T x) {
    // T extends Comparable<T>
    int c = x.compareTo(datum);
    if (c == 0) return;
    if (c < 0) {
        if (left != null) left.insert(x);
        else left = new TreeNode<T>(x);
    } else {
        if (right != null) right.insert(x);
        else right = new TreeNode<T>(x);
    }
}

**Deletion**

To delete an element x:
- remove x from its node – this creates a hole
- if the node was a leaf, just delete it
- find greatest y less than x in the left subtree (or least y greater than x in the right subtree), move it to x’s node
- this creates a hole where y was – repeat
Deletion
To find greatest y less than x:
• follow right children as far as possible in left subtree

Example: delete 25

Deletion

Deletion

Deletion

Deletion

Deletion

Deletion
Example: delete 47
Example: delete 29
Observation
• These operations take time proportional to the height of the tree (length of the longest path)
• O(n) if tree is not sufficiently balanced

Bad case for search, insertion, and deletion – essentially like searching a list

Solution
Try to keep the tree balanced (all paths roughly the same length)

Balanced Trees
• Size is exponential in height
• Height $= \log_2(\text{size})$
• Search, insert, delete will be $O(\log n)$

Creating a Balanced Tree
Creating one from a sorted array:
• Find the median, place that at the root
• Recursively form the left subtree from the left half of the array and the right subtree from the right half of the array

Keeping the Tree Balanced
• Insertions and deletions can put tree out of balance – we may have to rebalance it
• Can we do this efficiently?

AVL Trees
Adelson-Velsky and Landis, 1962

AVL Invariant:
The difference in height between the left and right subtrees of any node is never more than one
An AVL Tree

- Nonexistent children are considered to have height $-1$
- Note that paths can differ in length by more than 1 (e.g., paths to 2, 48)

AVL Trees are Balanced

The AVL invariant implies that:
- Size is at least exponential in height
  - $n \geq \varphi^d$, where $\varphi = (1 + \sqrt{5})/2 \approx 1.618$, the golden ratio!
- Height is at most logarithmic in size
  - $d \leq \log n / \log \varphi \approx 1.44 \log n$

AVL Invariant:
The difference in height between the left and right subtrees of any node is never more than one

To see that $n \geq \varphi^d$, look at the smallest possible AVL trees of each height

\[
\begin{align*}
A_0 &= 1 \\
A_1 &= 2 \\
A_2 &= A_{d-1} + A_{d-2} + 1, \quad d \geq 2
\end{align*}
\]
AVL Trees are Balanced

\[ A_0 = 1 \]
\[ A_1 = 2 \]
\[ A_d = A_{d-1} + A_{d-2} + 1, \quad d \geq 2 \]

1 2 4 7 12 20 33 54 88 ...

The Fibonacci sequence

Rebalancing

• Insertion and deletion can invalidate the AVL invariant
• May have to rebalance

Rotation

• A local rebalancing operation
• Preserves inorder ordering of the elements
• The AVL invariant can be reestablished with at most \( O(\log n) \) rotations up and down the tree

Example: delete 27

\begin{align*}
\text{Before} & \quad \text{After} \\
\begin{array}{c}
A \quad w \quad y \\
\end{array} & \quad \begin{array}{c}
A \quad A \\
\end{array} \\
\begin{array}{c}
\text{rotate} \\
\end{array} & \quad \begin{array}{c}
\text{rotate} \\
\end{array}
\end{align*}
Another balanced tree scheme
- Data stored only at the leaves
- Ordered left-to-right
- All paths of the same length
- Every non-leaf has either 2 or 3 children
- Each internal node has smallest, largest element in its subtree (for searching)
2-3 Trees

- smallest 2-3 tree of height $d = 3$
  $2^d = 8$ data elements
- largest 2-3 tree of height $d = 3$
  $3^d = 27$ data elements

- number of elements satisfies $2^d \leq n \leq 3^d$
- height satisfies $d \leq \log n$

Insertion in 2-3 Trees

- want to insert new element here

- want to insert new element here
Insertion in 2-3 Trees

Deletion in 2-3 Trees

want to delete this element

Deletion in 2-3 Trees

want to delete this element
Deletion in 2-3 Trees

If neighbor has 3 children, borrow one

This may cascade up the tree!
Deletion in 2-3 Trees

This may cascade up the tree!

Deletion in 2-3 Trees

This may cascade up the tree!

Conclusion

Balanced search trees are good
• Search, insert, delete in $O(\log n)$ time
• No need to know size in advance
• Several different versions
  – AVL trees, 2-3 trees, red-black trees, skip lists, random treaps, Huffman trees, ...
  – find out more about them in CS4820