ADTs Seen So Far

- Stack
  - Push
  - Pop

- Queue
  - Enqueue
  - Dequeue

- Priority Queue
  - Insert
  - DeleteMin

Then there is decreaseKey…
The Dictionary ADT

• Data:
  – a set of (key, value) pairs

• Operations:
  – Insert (key, value)
  – Find (key)
  – Remove (key)

Dictionary ADT is also called the “Map ADT”

We will tend to emphasize the keys, don’t forget about the stored values
A Modest Few Uses

• Sets
• Dictionaries
• Networks : Router tables
• Operating systems : Page tables
• Compilers : Symbol tables

• Anytime you want to store information according to some key and be able to efficiently retrieve it

**Probably the most widely used ADT!**
## Implementations

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>find</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Unsorted Linked-list</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Unsorted array</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td><strong>Sorted array</strong></td>
<td>$\log n + n$</td>
<td>$O(\log n)$</td>
<td>$\log n + n$</td>
</tr>
</tbody>
</table>

SO CLOSE!

What limits the performance?
Binary Search

Target 4

1 3 4 5 7 8 9 10
Binary Search

1 3 4 5 7 8 9 10
Binary Search

• Today’s idea: Get performance of binary search, but do it using a tree representation

• So, one more recap on trees and their terminology
Binary Trees

• Binary tree is
  – a root
  – left subtree *(maybe empty)*
  – right subtree *(maybe empty)*

• Representation:

<table>
<thead>
<tr>
<th>Data</th>
<th>left pointer</th>
<th>right pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
    A
   / \  /  \
  B   C  F
 / \   /  \
D   E G   H
  / \   / \
 I   J
```
Binary Tree: Representation

Keys required, but also store the value here
Tree Traversals

A traversal is an order for visiting all the nodes of a tree

Three types:

- **Pre-order**: Root, left subtree, right subtree
- **In-order**: Left subtree, root, right subtree
- **Post-order**: Left subtree, right subtree, root
Inorder Traversal

```c
void traverse(BNode t){
    if (t != NULL)
        traverse (t.left);
    process t.element;
    traverse (t.right);
}
```
Binary Tree: Special Cases

Complete Tree

Perfect Tree

“List” Tree

Full Tree
Binary Search Tree

- **Structural property**
  - each node has $\leq 2$ children
  - result:
    - storage is small
    - operations are simple

- **Order property**
  - all keys in left subtree smaller than root’s key
  - all keys in right subtree larger than root’s key
  - result: easy to find any given key
Are these BSTs?

![BSTs](image-url)
Are these BSTs?
Node Find(Object key, 
       Node root) {
    if (root == NULL) 
        return NULL;
    if (key < root.key) 
        return Find(key,  
                     root.left);
    else if (key > root.key) 
        return Find(key,  
                     root.right);
    else 
        return root;
}
Find in BST, Iterative

Node Find(Object key, Node root) {
    while (root != NULL && root.key != key) {
        if (key < root.key)
            root = root.left;
        else
            root = root.right;
    }
    return root;
}
Bonus: FindMin/FindMax

- Find minimum
- Find maximum
Insert in BST

Insertions happen only at the leaves – easy!

- Insert(13)
- Insert(8)
- Insert(31)
Deletion in BST

Why might deletion be harder than insertion?
Deletion

• Removing an item disrupts the tree structure.

• Basic idea: find the node to be removed, then “fix” the tree so that it is still a binary search tree.

• Three cases:
  – node has no children (leaf node)
  – node has one child
  – node has two children
Deletion – The Leaf Case

Delete(17)
Deletion – The One Child Case

Delete(15)
Deletion – The Two Child Case

Delete(5)

What can we replace 5 with?
Deletion – The Two Child Case

Idea: Replace the deleted node with a value guaranteed to be between the two child subtrees

Options:
• *succ* from right subtree: findMin(t.right)
• *pred* from left subtree: findMax(t.left)

Now delete the original node containing *succ* or *pred*
• Leaf or one child case – easy!
Finally...

Original node containing
7 gets deleted

7 replaces 5
BuildTree for BST

- We had BuildHeap, now let's talk about BuildTree

- Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST
  
  - If inserted in given order, what is the tree?
  
  - What big-O runtime for this kind of sorted input?

\[ O(N^2) \]

That ain't good
BuildTree for BST

• Insert keys 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST

• What we if could somehow re-arrange them
  – median first, then left median, right median, etc.
  – 5, 3, 7, 2, 1, 4, 8, 6, 9

  – What tree does that give us?

  – What big-O runtime?
    – $O(N \log N)$, definitely better

  – Does it fix our problem?
Unbalanced BST

- Balancing a tree at build time is insufficient, as sequences of operations can eventually transform that carefully balanced tree into the dreaded list

- At that point, everything is $O(N)$ and nobody is happy
  - Find
  - Insert
  - Delete
Binary Trees: Some Numbers

Recall: height of a tree = longest path from root to leaf.

For binary tree of height $h$:

- max # of leaves: $2^h$
- max # of nodes: $2^{(h+1)} - 1$
- min # of leaves: 1
- min # of nodes: $h - 1$

We’re not going to do better than $\log(n)$ height, and we need something to keep us away from $n$
Balanced BST

Observation
- BST: the shallower the better!
- For a BST with \( n \) nodes
  - Average height is \( O(\log n) \)
  - Worst case height is \( O(n) \)
- Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario

Solution: Require a Balance Condition that
1. ensures depth is \( O(\log n) \) – strong enough!
2. is easy to maintain – not too strong!
Potential Balance Conditions

1. Left and right subtrees of the root have equal number of nodes

Too weak!
Height mismatch example

2. Left and right subtrees of the root have equal \textit{height}

Too weak!
Left chain and Right chain
Potential Balance Conditions

3. Left and right subtrees of every node have equal number of nodes

4. Left and right subtrees of every node have equal height

Too strong!
Only perfect trees