In this document, we will learn about the calculations and traversals of binary search trees. Let's start with the tree calculations.

**Recall:** height is max number of edges from root to a leaf.

To find the height of the tree, we can use a recursive approach. For example, consider the tree shown below:

```
        B
       / \  
      D   E
     / \   \ 
    F   G  H
   /   /   / 
  I   J   K   L
```

The height of this tree is 3, as there are 3 edges from the root (B) to a leaf (L).

### More Recursive Tree Calculations: Tree Traversals

A **traversal** is an order for visiting all the nodes of a tree. There are three types of traversals:

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

For example, the pre-order traversal of the above tree would be: BDEFGHLJKI.

### Traversals

```c
void traverse(BNode t) {
    if (t != NULL) {
        traverse(t.left);
        print t.element;
        traverse(t.right);
    }
}
```

Now let's discuss binary trees.

- **Binary tree is**:
  - a root
  - left subtree (maybe empty)
  - right subtree (maybe empty)

#### Binary Trees

![Binary Tree Diagram]

- **Representation:**
  - Data
  - left
  - right
Binary Tree: Representation

Binary Tree: Special Cases

Binary Tree: Some Numbers!
For binary tree of height $h$:
- max # of leaves:
- max # of nodes:
- min # of leaves:
- min # of nodes:

ADTs Seen So Far

- Stack
  - Push
  - Pop
  - Priority Queue
    - Insert
    - DeleteMin
  - What about decreaseKey?

- Queue
  - Enqueue
  - Dequeue

The Dictionary ADT

- Data:
  - a set of (key, value) pairs
- Operations:
  - Insert (key, value)
  - Find (key)
  - Remove (key)

A Modest Few Uses

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

Probably the most widely used ADT!
Implementations
- Unsorted Linked-list
- Unsorted array
- Sorted array

Binary Search Tree Data Structure
- Structural property
  - each node has 2 children
  - result:
    - storage is small
    - operations are simple
    - average depth is small
- 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

What must I know about what I store?

Example and Counter-Example

Find in BST, Recursive

Find in BST, Iterative

Insert in BST

Insertions happen only at the leaves – easy!
BuildTree for BST
- Suppose keys 1, 2, 3, 4, 5, 6, 7, 8, 9 are inserted into an initially empty BST.
  - Runtime depends on the order!
    - in given order
    - in reverse order
    - median first, then left median, right median, etc.

Bonus: FindMin/FindMax
- Find minimum
- Find maximum

Delete Operation
- Delete is a bit trickier...Why?
- Suppose you want to delete 10
- Strategy:
  - Find 10
  - Delete the node containing 10
- Problem: When you delete a node, what do you replace it by?

Delete Operation
- Problem: When you delete a node, what do you replace it by?
- Solution:
  - If it has no children, by NULL
  - If it has 1 child, by that child
  - If it has 2 children, by the node with the smallest value in its right subtree (the successor of the node)

Delete “5” - No children
- Find 5 node
- Then Free the 5 node and NULL the pointer to it

Delete “24” - One child
- Find 24 node
- Then Free the 24 node and replace the pointer to it with a pointer to its child
Delete “10” - two children

Find 10, Copy the smallest value in right subtree into the node

Then recursively Delete node with smallest value in right subtree
Note: it does not have two children

Delete “11” - One child

Remember 11 node

Then Free the 11 node and replace the pointer to it with a pointer to its child

Runtimes

- Find? Insert? Delete?
- What is the average height of a BST?
- What is the maximum height?
- What happened when we insert nodes in sorted order?

Balanced BST

Observation
- BST: the shallower the better!
- 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

2. Left and right subtrees of the root have equal height

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
The AVL Balance Condition
Left and right subtrees of every node have equal heights differing by at most 1

Define: $\text{balance}(x) = \text{height}(x.\text{left}) - \text{height}(x.\text{right})$

AVL property: $-1 \leq \text{balance}(x) \leq 1$, for every node $x$

- Ensures small depth
  - Will prove this by showing that an AVL tree of height $h$ must have a lot of (i.e. $O(2^h)$) nodes
- Easy to maintain
  - Using single and double rotations

The AVL Tree Data Structure

Structural properties
1. Binary tree property
2. Balance property: balance of every node is between -1 and 1

Result:
- Worst case depth is $O(\log n)$

Ordering property
- Same as for BST