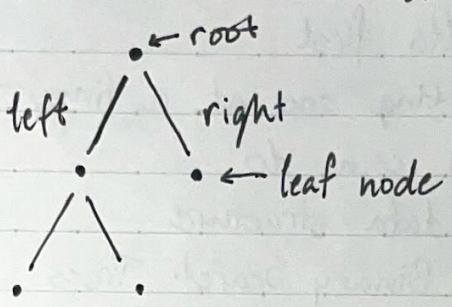
Date 2025.04.16

Binary Tree

- Think of it as a linked structure



Binary Search Tree (BST)

- Binary tree
- Follows the binary search tree property
- (This involves more operations)

Examples

We are asking the program too much

- Build a PQ. Is N in there? ← Very Important
- Build a hash table. Output is in ascending order

Operations for BST

- insert
- delete
- search

] Basic Dictionary Functions

- min/max
- predecessor/successor
- traversals

- Typical performance of a bst function is $O(n)$

... h - height of tree

hope for $h \cong O(\log n)$
worst case $h \cong O(n)$

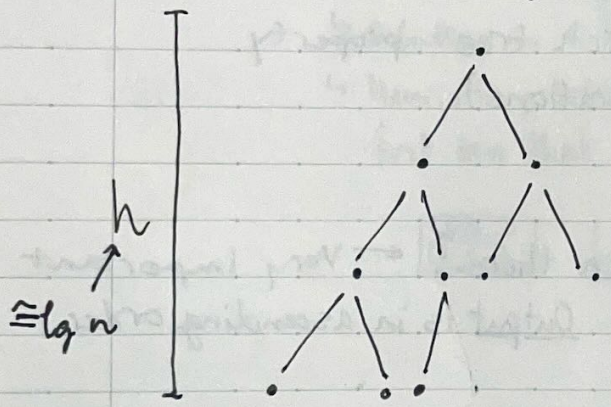
] Depends on construction of the tree
WC: Sorted

Date

WC IS COMMON

- What to do?
  - ↳ Randomize delta first
    But if inserting one at a time — there's nothing you can do
  - ↳ Use another data structure
    - ↳ Balancing Binary Search Trees
      - ↳ A binary tree that tries to stay as balanced as it can be

"Best Case" "WC



- ↳ Examples:
  - ↳ AVL Tree
  - ↳ Red/Black Trees

Date

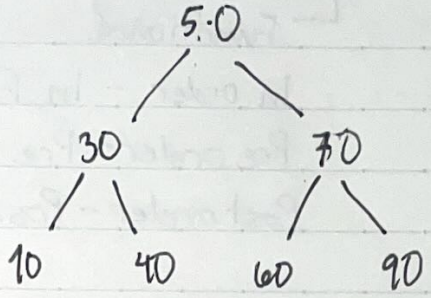
Our BST will just use Ints

BST Property

- key(x.right) > key(x)
- key(x.left) ≤ key(x)
- Assume unique keys

There won't be repeating keys in test cases

BST Example



BST Traversals

- A traversal is a "walk" of that "touches" each node once in some order
  - O(n) → 'touch' n nodes

★ Memorize These ★

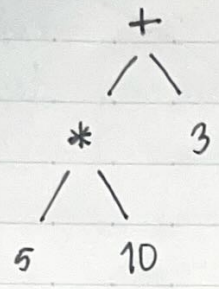
- Preorder: this, left, right
- In order: left, this, right
- Post order: left, right, this

```
inOrder(x) {        node reference/pointer
  if(x == null) return
  inOrder(x.left);
  print x; //or whatever
  inOrder(x.right);
}
```

- Preorder: 50, 30, 10, 40, 70, 60, 90
- InOrder Example w/ BST above: 10, 30, 40, 50, 60, 70, 90
- Post order: 10, 40, 30, 60, 90, 70, 50

Double Check!

EXAMPLE: 5 * 10 + 3



- Pre Order: + * 5 10 3
- In order: 5 * 10 + 3
- Post-Order: 5 10 * 3 +
- Algebraic form / in fix
- RPN (Reverse Polish Notation / Post Fix Not.)
- Functional

In order - In Fix
Pre order - Pre Fix
Post order - Post Fix

DON'T MAKE THIS MISTAKE
- ↳ Write other functions on scratch
- ↳ Don't cut and paste, if you do make sure to rename them

RECURSIVE SEARCH
- Must be (at least one) public wrapper function (User doesn't have access to internal reference/pointers in the tree, including root)

node ref, key value — *k is target value

Private
```
search(x, k) {
  if (x == null || x.key == k)  // found & not found
    return x;
  if (k < x.key)
    return search(x.left, key);
  else
    return search(x.right, key);
```

ITERATIVE SEARCH

Private
O(h)

```
search (x, k) {
   while (x != null && x.key != k) {
      if( k < x.key)
         x = x.left;
      else
         x = x.right;
   }
   return x;
}
```

MIN ← node ref / node you're examining ← can be the root

Private
O(h)

```
min(x)                          *PRE: x is NOT null
   if(x == null) return null;
   while (x.left != null)
      x = x.left;
   return x;
```

MAX

Private
O(h)

```
max (x) {
   if(x == null) return null;
   while (x.right != null)
      x = x.right;
   return x;
```

t can be null

O(h)

Successor.

```
successor(x)
   if (x == null) return null;
   if (x.right != null)                } Case #1
      return min(x.right);
   t = x.parent;
   while (t != null && x == t.right) {
      x = t;
      t = t.parent;
   }
   return t;
}
```

* You have to program predecessor
* Nodes are only attached as leaves

*Will have to memorize how to code all of these funcs

```
        ↙key value
insert (k) {
                                          node
                                          ┌──────────────┐ * C++ while(temp)*
O(h)   n = new node(k); //node constructor left, right, parent set to null
    p = null; t = root;   // p = prev, t = temp
    // traversal (no look-ahead)
    while (t != null) {
        p = t;
        if(k < t.key)
            t = t.left;
        else
            t = t.right;
    } // t always null here
    n.parent = p;
    if (p == null) {
        root = n;
        return;
    }
    if (k < p.key) {
        p.left = n;
    else
        p.right = n;
    }
} // insert
```