BST RUNNING TIME ANALYSIS

Problem Solving with Computers-II

Big-Omega

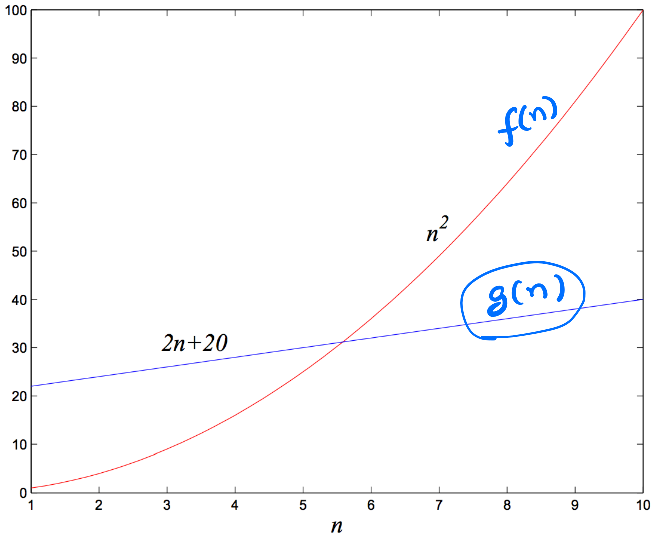
- f(n) and g(n) map positive integer inputs to positive reals.

We say $\boxed{f = \Omega(g)}$ if there are constants $c > 0$, $k>0$ such that

$c \cdot g(n) \leq f(n)$ for $n >= k$

$f = \Omega(g)$

means that "f grows at least as fast as g"

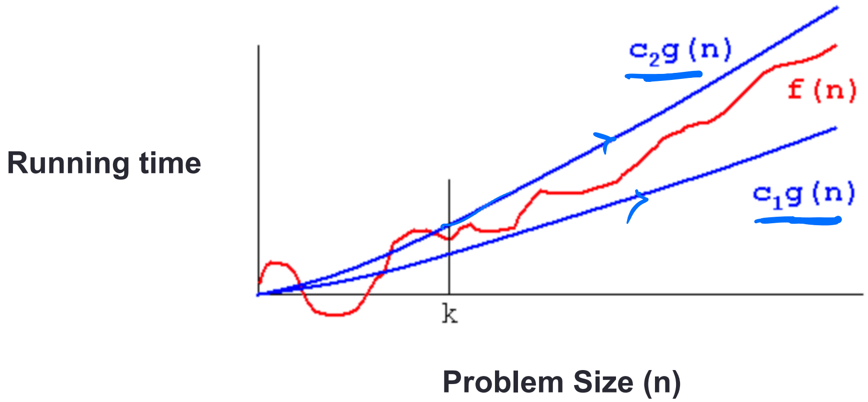


Big-Theta

- f(n) and g(n) map positive integer inputs to positive reals.

We say $f = \Theta(g)$ if there are constants c_1, c_2, k such that

$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, for $n >= k$

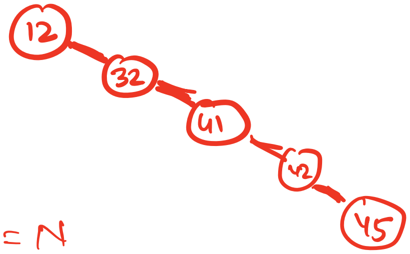


Binary Search Trees

- WHAT are the operations supported?
- HOW do we implement them?
- WHAT are the (worst case) running times of each operation?

- Path – a sequence of nodes and edges connecting a node with another node.
- A path starts from a node and ends at another node or a leaf
- Height of node – The height of a node is the number of edges on the longest downward path between that node and a leaf.

1.

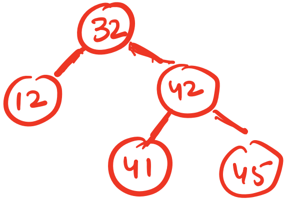


H = N

Path: 12 → 32 → 41 → 42

Height : 4

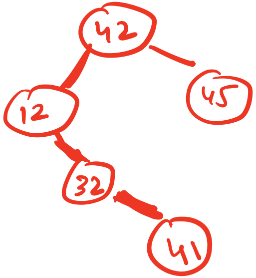
2.



32 → 42 → 41

Height: 2

3.

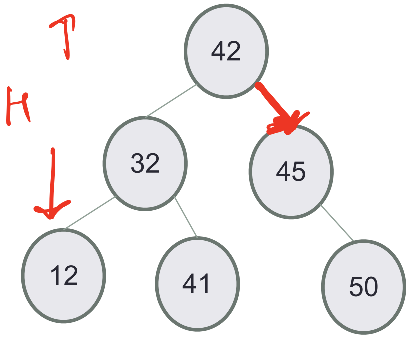


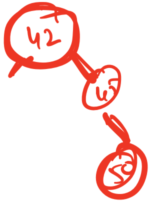
Height : 3

BSTs of different heights are possible with the same set of keys

Examples for keys: 12, 32, 41, 42, 45

Worst case Big-O of search, insert, min, max : O(H)





Given a BST of height H with N nodes, what is the worst case complexity of searching for a key?

A. O(1)

B. O(log H)

C. O(H) Worst case.

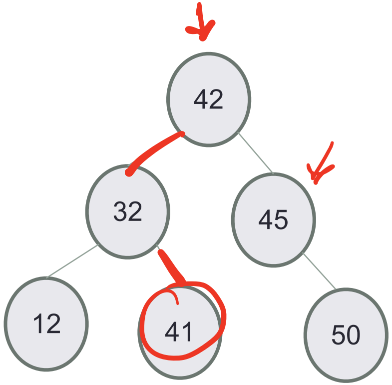
D. O(H*log H)

E. O(N)

Best case : Searching for the key @ root
O(1)

Worst case: Search for a leaf that is H edges away.

Worst case Big-O of predecessor / successor



Given a BST of height H and N nodes, what is the worst case complexity of finding the predecessor or successor key?

A. O(1)

B. O(log H)

C. O(H)

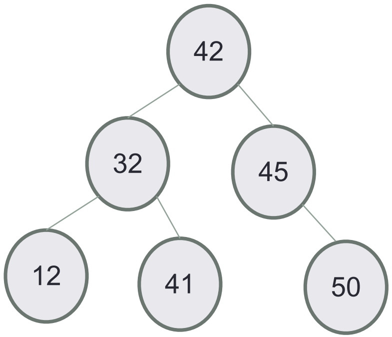
D. O(H*log H)

E. O(N)

Best case: O(1)

Worst case: O(H)

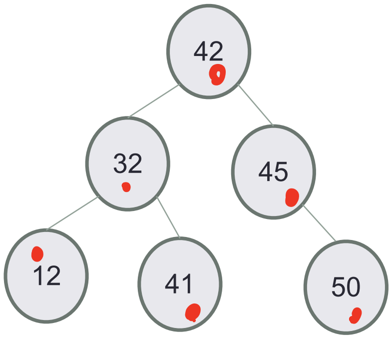
Worst case Big-O of delete



Given a BST of height H and N nodes, what is the worst case complexity of deleting a node?

A. O(1)
B. O(log H)
C. O(H)
D. O(H*log H)
E. O(N)

Big O of traversals



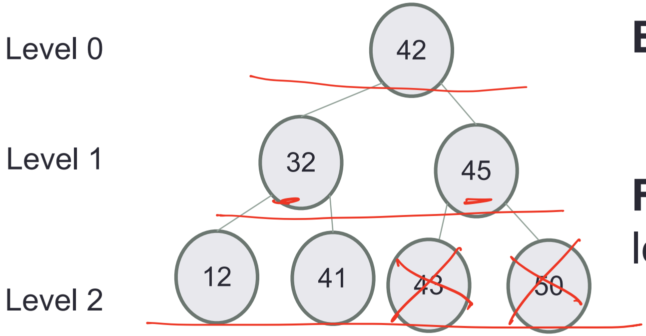
In Order: O(n)

Pre Order: O(n)

Post Order: O(n)

All operations except traversals are O(H)

Types of BSTs



Balanced BST: $H = O(\log_2 N)$

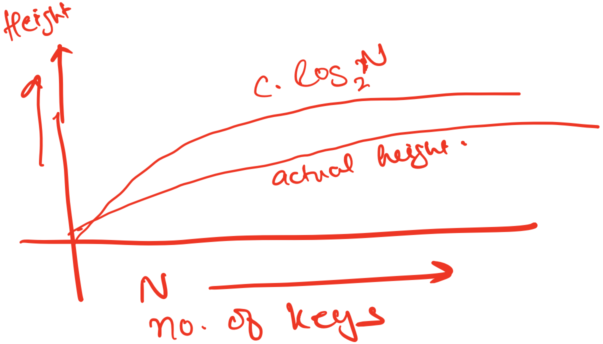
Height of the tree

e.g AVL, Red-Black Trees

Full Binary Tree: Every node other than the leaves has two children.

Show that a full BST is a balanced BST

Complete Binary Tree: Every level, except possibly the last, is completely filled, and all nodes are as far left as possible



$T(n) = c_1 n^2 + c_2 n$

$H(N) = ?$

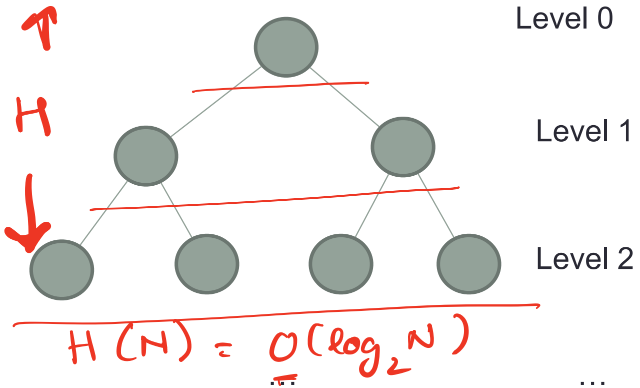
Relating H (height) and N (#nodes)

$$\sum_{i=0}^{H} 2^i = 2^{H+1} - 1$$

$$2^{H+1} - 1 = N$$

$$2^{H+1} = N+1$$

$$H = \log_2(N+1) - 1$$



$H(N) = O(\log_2 N)$

What is the height (exactly) of a full binary tree in terms of N?

$$H(N) = \lceil \log_2(N+1) \rceil - 1 \quad \text{(Complete tree)}$$

$$H(N) \le \log_2(N+1) - 1 + 1$$

$$H(N) = O(\log_2 N)$$

Balanced trees

- Balanced trees by definition have a height of O(log N)
- A completely filled tree is one example of a balanced tree
- Other Balanced BSTs include AVL trees, red black trees and so on
- Visualize operations on an AVL tree: https://visualgo.net/bn/bst

Big-O analysis of iterative Fibonacci

```
function F(n){
 Create an array fib[1..n]
 fib[1] = 1
 fib[2] = 1
 for i = 3 to n:
    fib[i] = fib[i-1] + fib[i-2]
 return fib[n]
}
```

O(1)

O(n)

O(1)

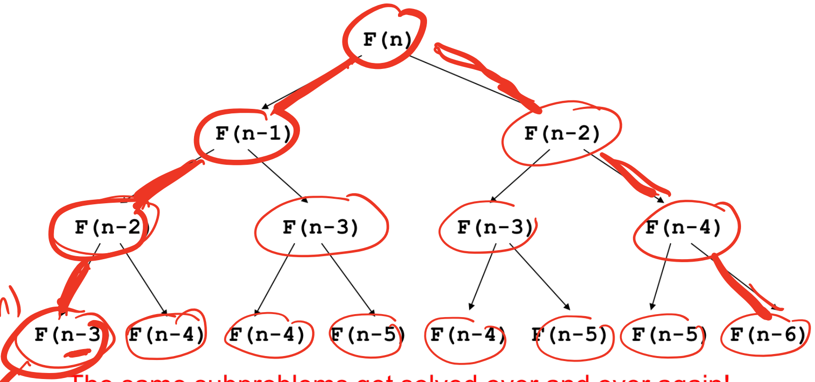
$T(n) = O(1) + O(n) \cdot O(1)$

$= O(n)$

Big-O analysis of recursive Fibonacci

```
function F(n){
   if(n == 1) return 1
   if(n == 2) return 1
return F(n-1) + F(n-2)
}
```

What takes so long? Let's unravel the recursion…



The same subproblems get solved over and over again!

T(n) : # of primitive steps to calculate F(n)

$T(1) = 1$ $T(2) = 2$

$T(n) = T(n-1) + T(n-2) + 5$

Recurrence relation

$T(n) = c \cdot$ # of function calls

We can obtain upper and lower bounds on the number of function calls by thinking about the minimum and maximum nodes in the call tree

$$T(n) \leq c(2^{n-1+1} - 1)$$

$$T(n) \geq c(2^{n/2+1} - 1)$$

$$T(n) = O(2^{n})$$

$$T(n) = \Omega(2^{n/2})$$