BEST & WORST CASE ANALYSIS RUNNING TIME OF BST OPERATIONS

Problem Solving with Computers-II

Definition of Big-O

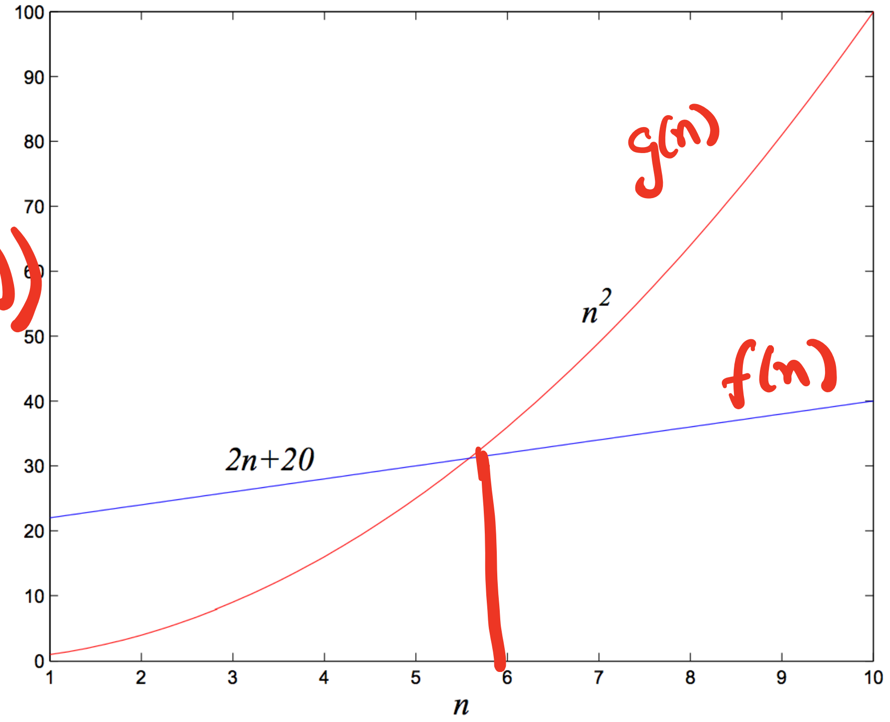
f(n) and g(n) map positive integer inputs to positive reals.

We say $f = O(g)$ if there is a constant $c > 0$ and $k>0$ such that $f(n) \leq c \cdot g(n)$ for all $n >= k$.

$f = O(g)$

means that “f grows no faster than g”

$f = O(g)$ $\quad$ $f(n) = O(g(n))$



Big-Omega

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

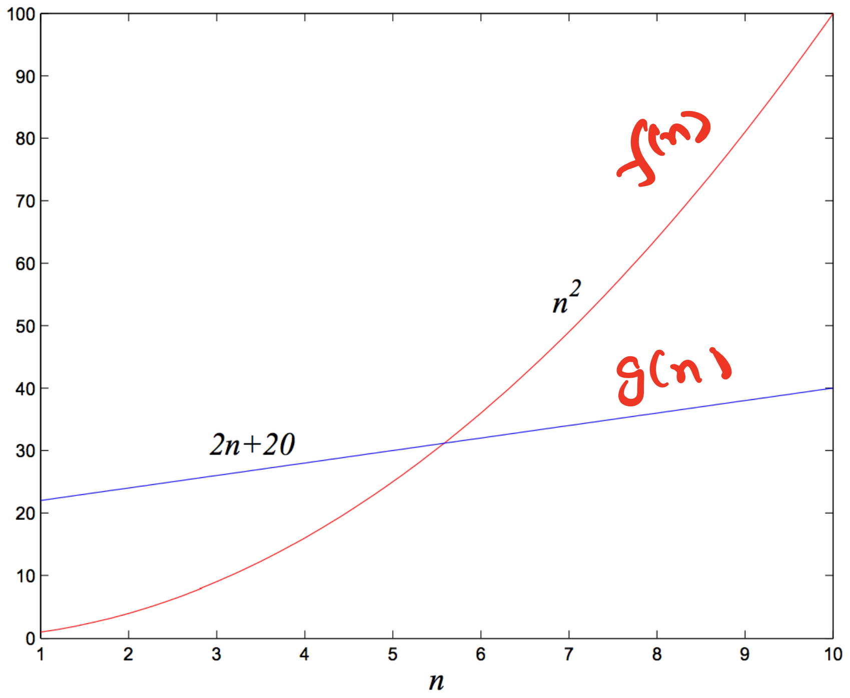
We say $f = \Omega(g)$ if there are constants $c > 0$, $k>0$ such that

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

$f = \Omega(g)$

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

$f(n) = \Omega(g(n))$



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

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

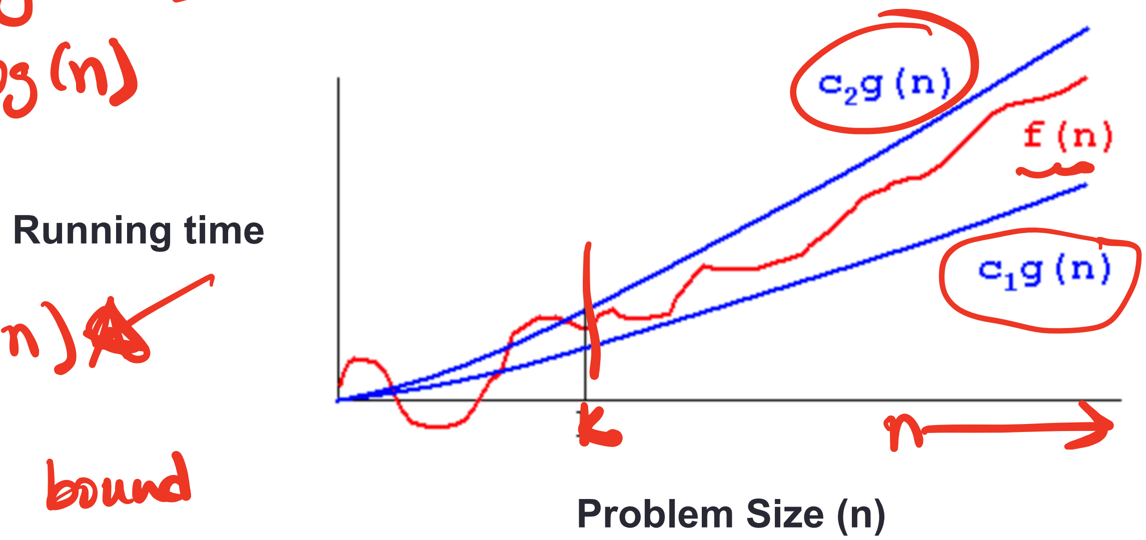
$f(n) = \Theta(g(n))$

$f(n) = 5n\log(n)$

$= O(n^2)$

$= O(n \log n)$

"tightest" upper bound



Best case and worst case analysis

What is the Big-O running time of search in a sorted array of size n?

n is very large

…using linear search?

Best case: looking for the min key value : O(1)

Worst case: looking for the max key value : O(n)

…using binary search?

Best case: looking for the mid value : O(1)

Worst case: looking for a key that doesn't exist: O(log n)

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Worst case analysis of binary search

```
bool binarySearch(int arr[], int element, int n){
//Precondition: input array arr is sorted in ascending order
  int begin = 0;
  int end = n-1;
  int mid;
  while (begin <=  end){
    mid = (end + begin)/2;
    if(arr[mid]==element){
      return true;
    }else if (arr[mid]< element){
      begin = mid + 1;
    }else{
      end = mid - 1;

    }
  }
  return false;
}
```

C_1

C_1

C_2

Iteration no.	end - begin
1	$n-1$
2	$\left(\frac{n-1}{2}\right) - 1$
3	$\frac{1}{2}\left(\frac{n-1}{2} - 1\right) - 1 = \frac{n-1}{2^2} - \frac{1}{2} - 1$
4	$\frac{n-1}{2^3} - \frac{1}{2^2} - \frac{1}{2} - 1$
⋮	

Iteration No k | end - begin

$$\frac{n-1}{2^{k-1}} - \left(1 + \frac{1}{2} + \frac{1}{2^2} + \cdots \frac{1}{2^{k-2}}\right)$$

(sum the geometric series)

$$= \frac{n-1}{2^{k-1}} - \frac{\left(1 - \frac{1}{2^{k-2+1}}\right)}{\left(1 - \frac{1}{2}\right)}$$

$$= \frac{n-1}{2^{k-1}} - 2 \cdot \left(1 - \frac{1}{2^{k-1}}\right)$$

$$= \frac{(n-1+2)}{2^{k-1}} - 2 \quad \text{— (1)}$$

At iteration k,

$$(\text{end} - \text{begin}) = \frac{(n+1)}{2^{k-1}} - 2$$

We stop when $(\text{end} - \text{begin}) < 0$

(Substitute for end-begin from (1))

$$\frac{n+1}{2^{k-1}} < 2$$

$$2 \cdot 2^{k-1} > n+1$$

$$2^k > n+1$$

$$k > \log_2(n+1)$$

The max number of iterations of the while loop is $\log(n+1)$

Running time of binary search on array of size n

$$T(n) = c_1 + \log(n+1) \cdot c_2,$$ for some constants c_1 & c_2

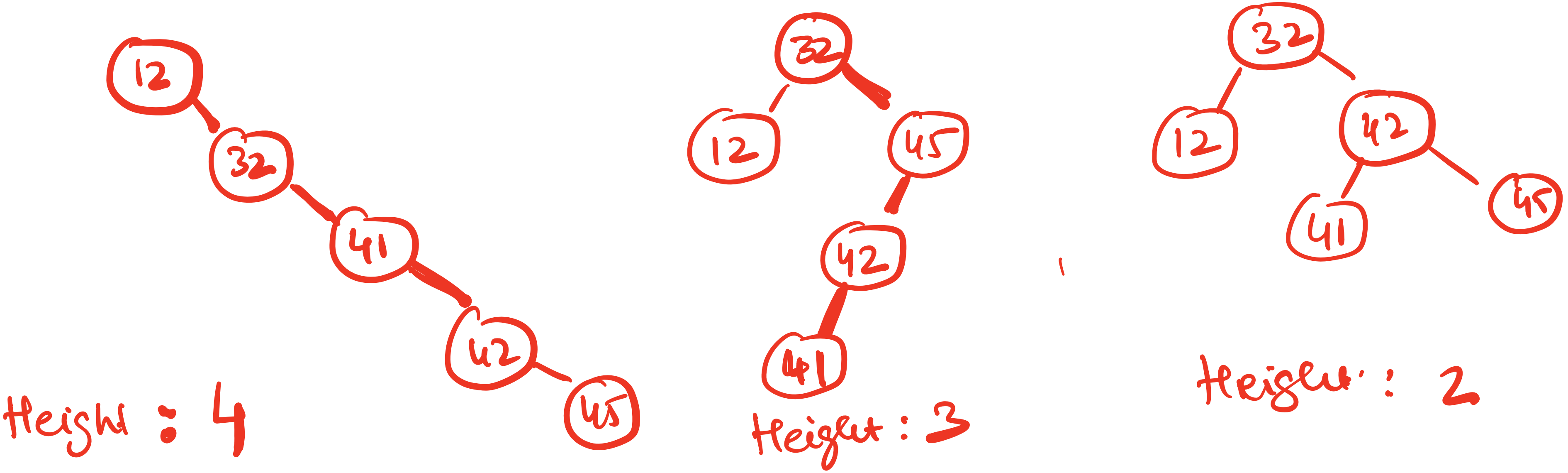
$$= \underline{O}(\log n)$$ (By def. of Big O)

Best case and worst case : sorted array

	Best case	Worst case
Search (Binary search)	$O(1)$	$O(\log n)$
Min/Max	$O(1)$	$O(1)$
Median	$O(1)$	$O(1)$
Successor/Predecessor	$O(1)$	$O(1)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

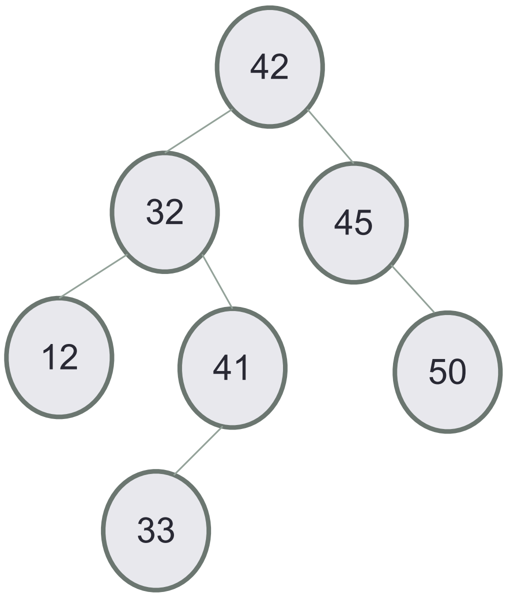
- Path – a sequence of (zero or more) connected nodes.
- Length of a path - number of edges traversed on the path
- Height of node – Length of the longest path from the node to a leaf node.
- **Height of the tree** - Length of the longest path from the **root** to a leaf node.



BSTs of different heights are possible with the same set of keys
Examples for keys: 12, 32, 41, 42, 45

BST search - best case

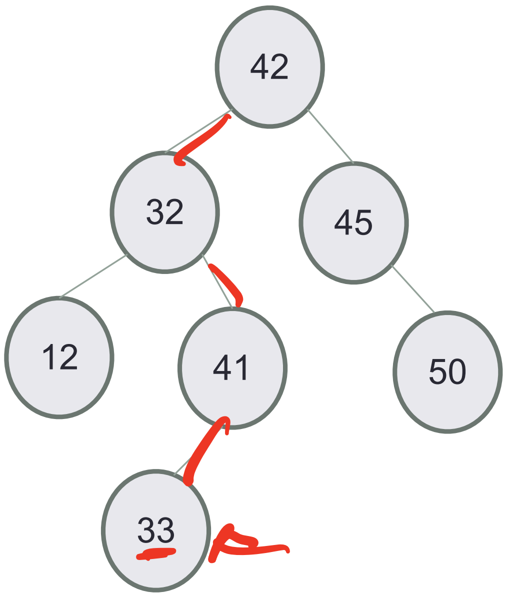
O(1)



Given a BST with N nodes, in the best case, which key would be searching for?

A. root node (e.g. 42)
B. any leaf node (e.g. 12 or 33 or 50)
C. leaf node that is on the longest path from the root (e.g. 33)
D. any key, there is no best or worst case

BST search - worst case



Given a BST with N nodes, in the worst case, which key would be searching for?

A. root node (e.g. 42)

B. leaf node (e.g. 12 or 41 or 50)

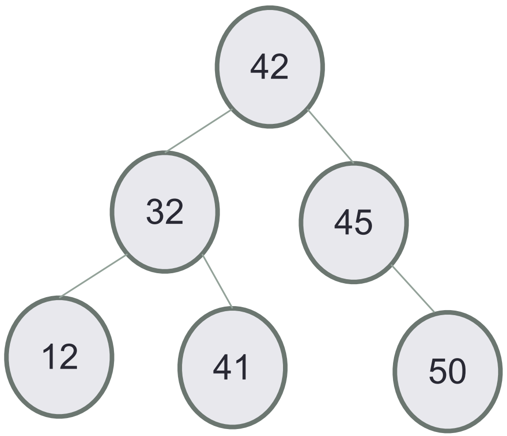
C. leaf node that is on the longest path from the root (e.g. 33)

D. a key that doesn't exist in the tree

$T(n) = O(H)$

H: Height of the tree

Worst case Big-O of search, insert, min, max



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

A. O(1)

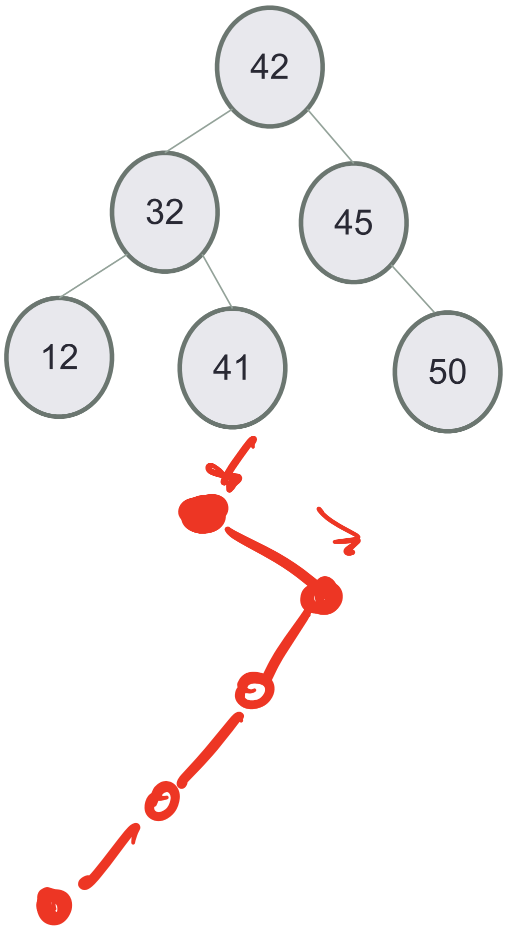
B. O(log H)

C. O(H)

D. O(H*log H)

E. O(N)

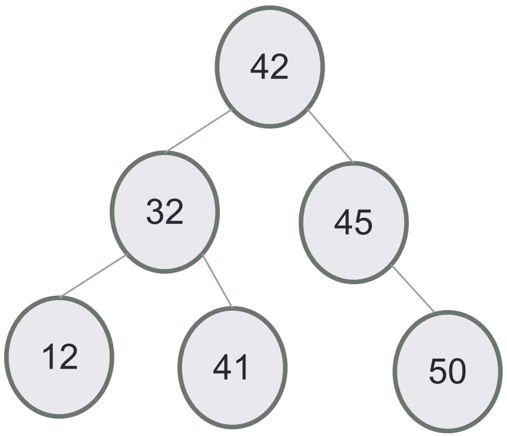
BST operations (worst case)



Given a BST of height H and N nodes, which of the following operations has a complexity of O(H)?

A. min or max
B. insert
C. predecessor or successor
D. delete
E. All of the above

Big O of traversals

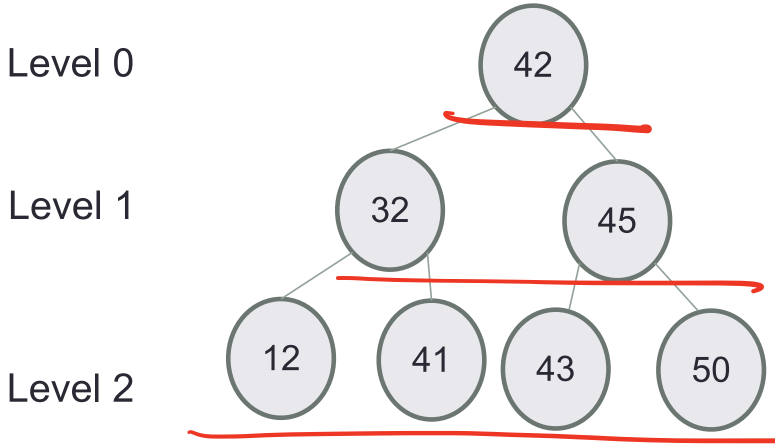


In Order: O(N)

Pre Order: O(N)

Post Order: O(N)

Types of BSTs



Example of a full binary tree

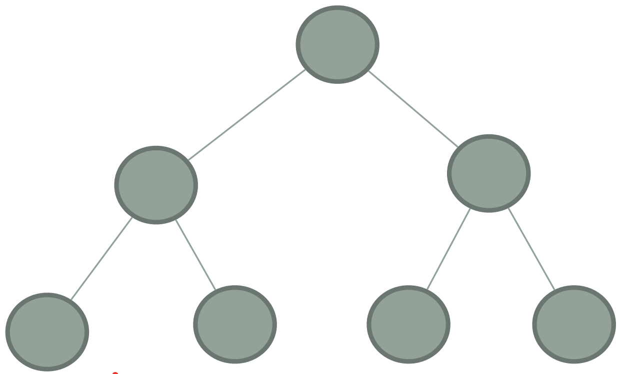
Balanced BST: Any BST where the height is $O(\log n)$

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

Full Binary Tree: A complete binary tree whose last level is completely filled

Relating H (height) and n (#nodes) for a full binary tree

We derived this result in the previous lecture.



Height of full binary tree : H

Number of nodes : N

Sum of the number of nodes from level 0 to level H must be equal to N

Sum of the number of nodes from level 0 to H…

$= 2^0 + 2^1 + 2^2 + \dots 2^H$

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

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

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

$\Rightarrow H = \log(N+1) - 1$

$= O(\log N)$

A full binary tree is a balanced tree because its height is O(log 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