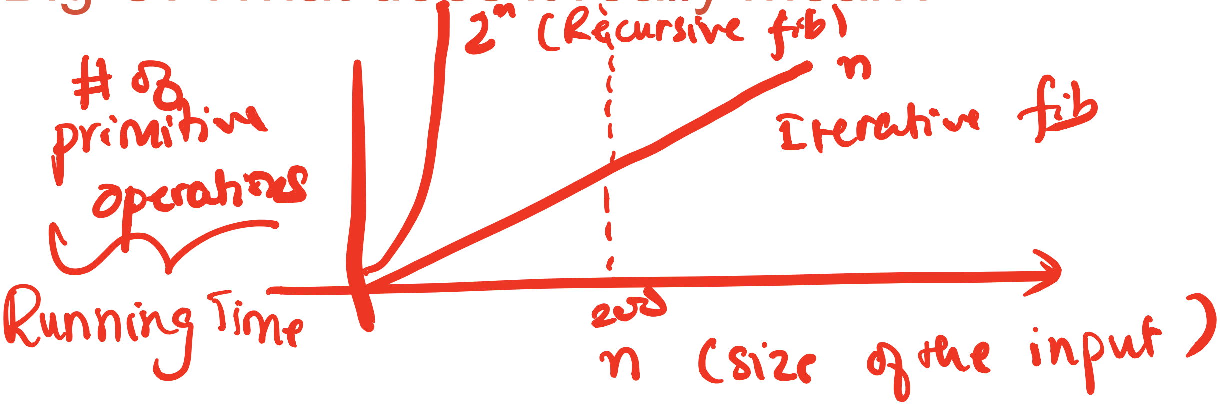
RUNNING TIME ANALYSIS - PART 2
BINARY SEARCH TREES

Problem Solving with Computers-II

How is PA01 going?

A. Done!
B. On track to finish
C. On track to finish but my code is a mess
D. Stuck and struggling
E. Haven’t started

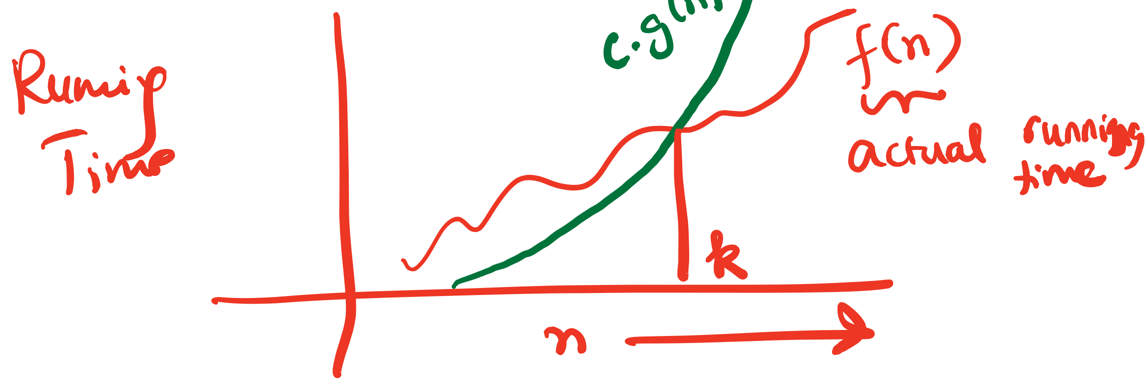
Big O: What does it really mean?



Big O is an upper limit on the running time of an algorithm as n gets large

$$f(n) = 5\underbrace{N^2 \log N} + \cancel{N} + \cancel{1000}$$

$$= O(\underbrace{N^2 \log N}) \rightarrow g(n)$$



$$f(n) = 5N^2 \log N + N + 1000$$

$$< 5N^2 \log N + N^2 \log N + N^2 \log N$$

$$= \underset{c}{7}\,\underbrace{N^2 \log N}_{g(n)}$$

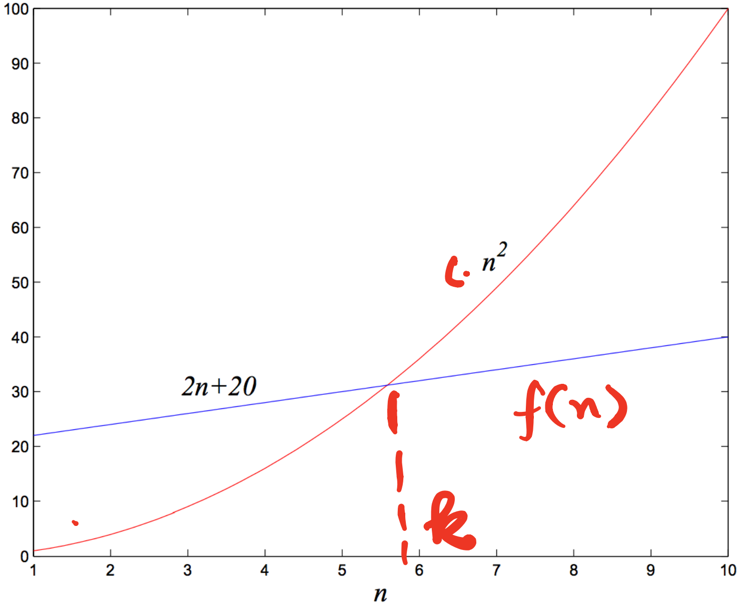
$$f(n) = O(N^2 \log N)$$

A more precise definition of Big-O

- f(n) and g(n): running times of two algorithms on inputs of size n.
- 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"



According to the definition on the previous slide if $f = O(g)$, $f = O(h)$ for any $h > g$

So if $f = O(N^2 \log N)$, then technically

$f = O(N^3)$

$f = O(N^4)$ .... and so on

But in practice when doing Big-O analysis we look for the lowest order function that satisfies the definition of Big-Oh (the tightest upper bound to $f(n)$)

What is the Big-O running time of algoX?

- Assume **dataA** is some data structure that supports the following operations with the given running times, where n is the number of keys stored in the data structure:
  - insert: O(log n)
  - min: O(1)
  - delete: O(log n)

n is the no. of keys in ds

input size

```
void algoX(int arr[], int N)
{
        dataA ds;//ds contains no keys
        for(int i=0; i < N;   i++)
                ds.insert(arr[i]);
        for(int i=0; i < N;   i++){
                arr[i] = ds.min();
                ds.delete(arr[i]);
        }
}
```

A. $O(N^2)$
B. O(N logN)
C. O(N)
D. O(log N)
E. Not enough information to compute

```
for(int i=0; i < N;    i++)
        ds.insert(arr[i]);
```

Running time of this loop is less than

$c_1 N \log N$

Reason:

Each insert takes a different amount of time because the running time depends on the number of keys already in ds.

The first insert takes the least time, the last one takes the most.

Although we don't know the exact number of operations for each insert, we can find an upper limit.

Specifically, the running time of each insert is less than $c_1 \log N$

```
for(int i=0; i < N;   i++){
        arr[i] = ds.min();
        ds.delete(arr[i]);
}
```

Running time of this loop is less than

$N(c_2 + c_3 \log N)$

Overall running time is

$c_1 N \log N + c_2 N + c_3 N \log N$

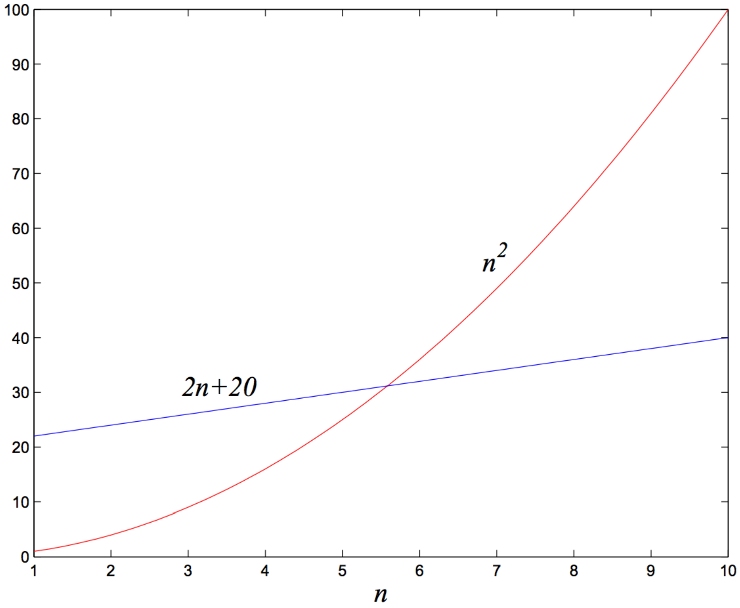
$= O(N \log N)$

Big-Omega

- f(n) and g(n): running times of two algorithms on inputs of size n.
- 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 $c \cdot g(n) \leq f(n)$ for $n >= k$

$f = \Omega(g)$
means that "f grows at least as fast as g"

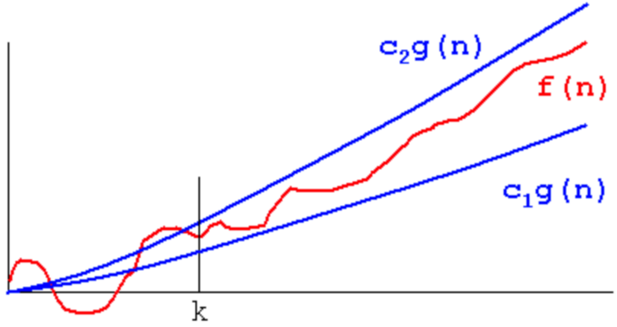


Big-Theta

- f(n) and g(n): running times of two algorithms on inputs of size n.
- 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 $n >= k$

Running time



Problem Size (n)

Best case, worst case, average case running times

Operations on sorted arrays

- Min : $O(1)$
- Max: $O(1)$
- Median: $O(1)$
- Successor: $O(1)$
- Predecessor: $O(1)$
- Search: depends on the search algo
- Insert : → worst case $O(N)$
- Delete: → worst case $O(N)$

Binary Search:

$$\frac{N}{2^{k-1}} <= 1$$

$$N <= 2^{k-1}$$

$$\log N <= k-1$$

$$k >= (\log N + 1)$$

Iteration #	Size of my array
1	N
2	$N/2$
3	$N/4$
4	$N/8$
k	$N/2^{k-1}$

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

lo ↑ (index 0) hi ↑ (index 14)

42

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;
 0 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;
}
```

f

Worst case running time:

→ See previous slide for getting an upper limit on the number of iterations

→ constant time

Overall $= (\log N+1) * c$

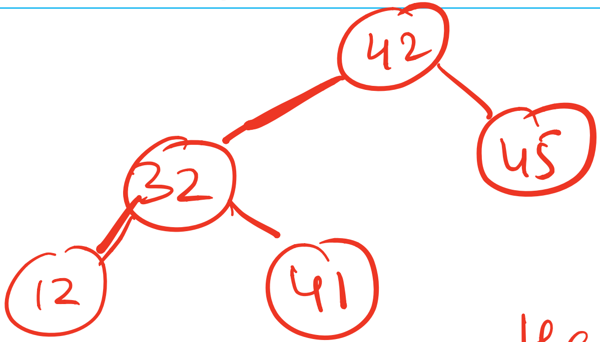
$= O(\log N)$

Binary Search Trees

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

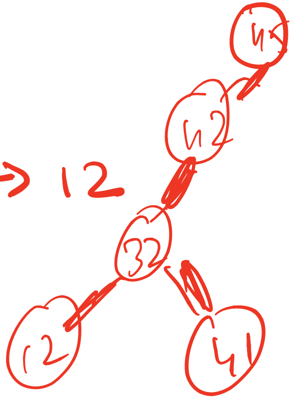
Height of the tree

- Path – a sequence of nodes and edges connecting a node with a descendant.
- 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.



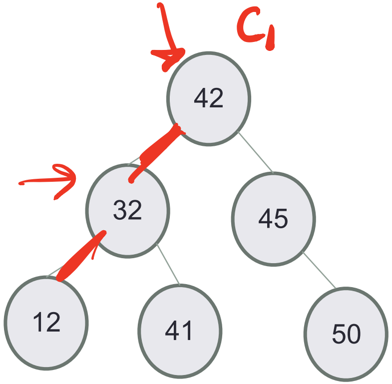
Path: 42 → 32
32 → 41
42 → 32 → 12

Height : 2



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



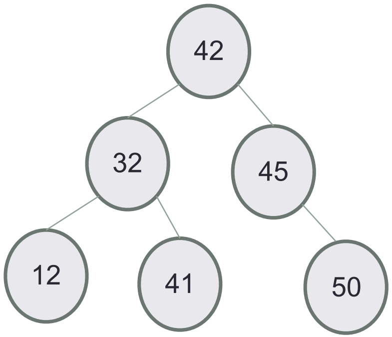
2

$c_1 * H$

- 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)
D. O(H*log H)
E. O(N)

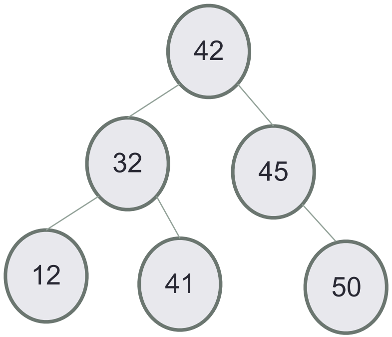
Worst case Big-O of insert



- Given a BST of height H and N nodes, what is the worst case complexity of inserting a key?

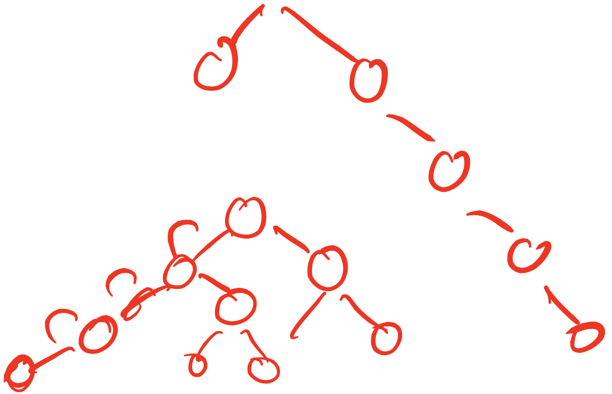
A. O(1)
B. O(log H)
C. O(H)
D. O(H*log H)
E. O(N)

Worst case Big-O of min/max

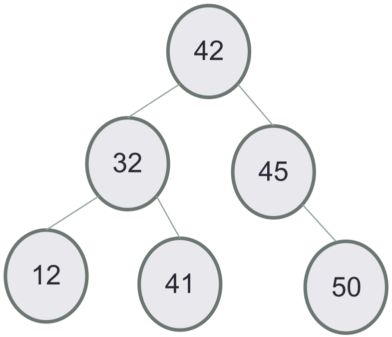


- Given a BST of height H and N nodes, what is the worst case complexity of finding the minimum or maximum key?

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



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)

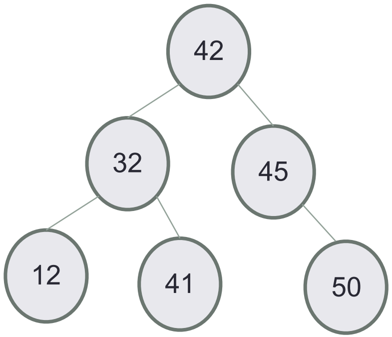
B. O(log H)

C. O(H)

D. O(H*log H)

E. O(N)

Worst case Big-O of delete



- Given a BST of height H and N nodes, what is the worst case complexity of deleting the key (assume no duplicates)?

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

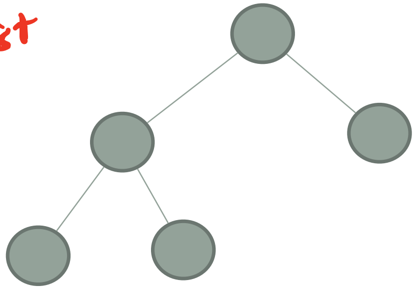
Worst case analysis

Are binary search trees *really* faster than linked lists for finding elements?

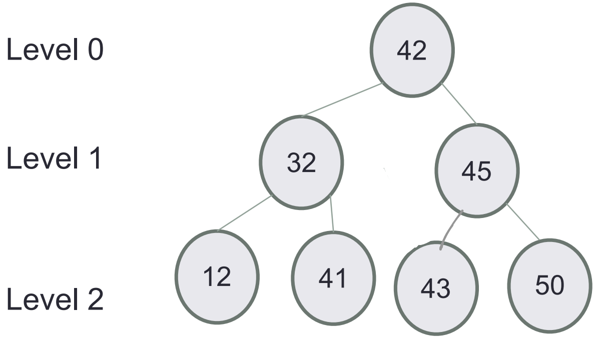
- A. Yes
- B. No

In the worst case, BST looks like a linked list and $H = N-1$

In this case all the operations will be $O(N)$ same as linked list



Completely filled binary tree



Nodes at each level have exactly two children, except the nodes at the last level

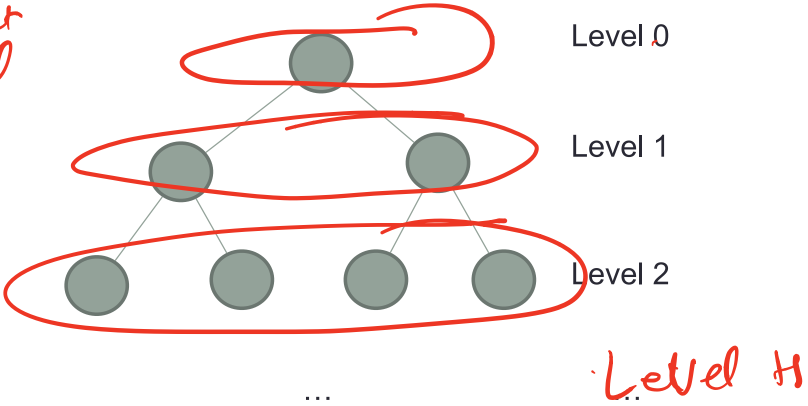
A balanced BST, by definition is one where $H = O(\log N)$

We will show that a completely filled BST is balanced. To do this we have to show that its height is $O(\log N)$

Relating H (height) and N (#nodes)

find is O(H), we want to find a f(N) = H

Level	no. of nodes at that level
0	1
1	2
2	4
k	2^k
H	2^H



How many nodes are on level L in a completely filled binary search tree?

A. 2

B. L

C. 2*L

D. 2^L

Relating H (height) and N (#nodes)

find is O(H), we want to find a f(N) = H

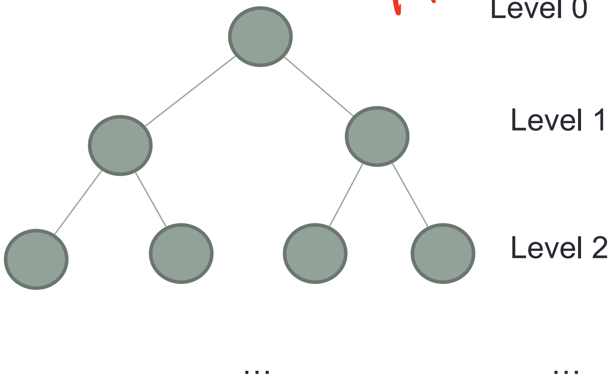
$$2^{H+1} = N+1$$

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

$$\underbrace{1+2+4+\dots 2^H}_{\downarrow} = N$$

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

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



Finally, what is the height (exactly) of the tree in terms of N?

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

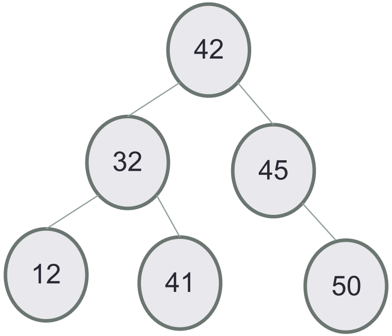
Fun fact that we used in this proof: the largest non-negative no. that can be stored in 8 bits is

$$1+2+4+\dots 2^7 = 2^8 - 1 = 255$$

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 of traversals



In Order: O(n)

Pre Order: O(n)

Post Order: O(n)

Summary of operations : Worst case Big-O

Operation	Sorted Array	Balanced Binary Search Tree	Linked List
Min	O(1)	O(logN)	O(N)
Max	O(1)	O(logN)	O(N)
Median	O(1)	—	—
Successor	O(1)	O(logN)	—
Predecessor	O(1)	O(logN)	—
Search *	O(logN)	O(logN)	O(N)
Insert	O(N)	O(logN)	O(1) Insert to head
Delete	O(N)	O(logN)	O(N) to search, O(1) to delete