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

Midterm 2

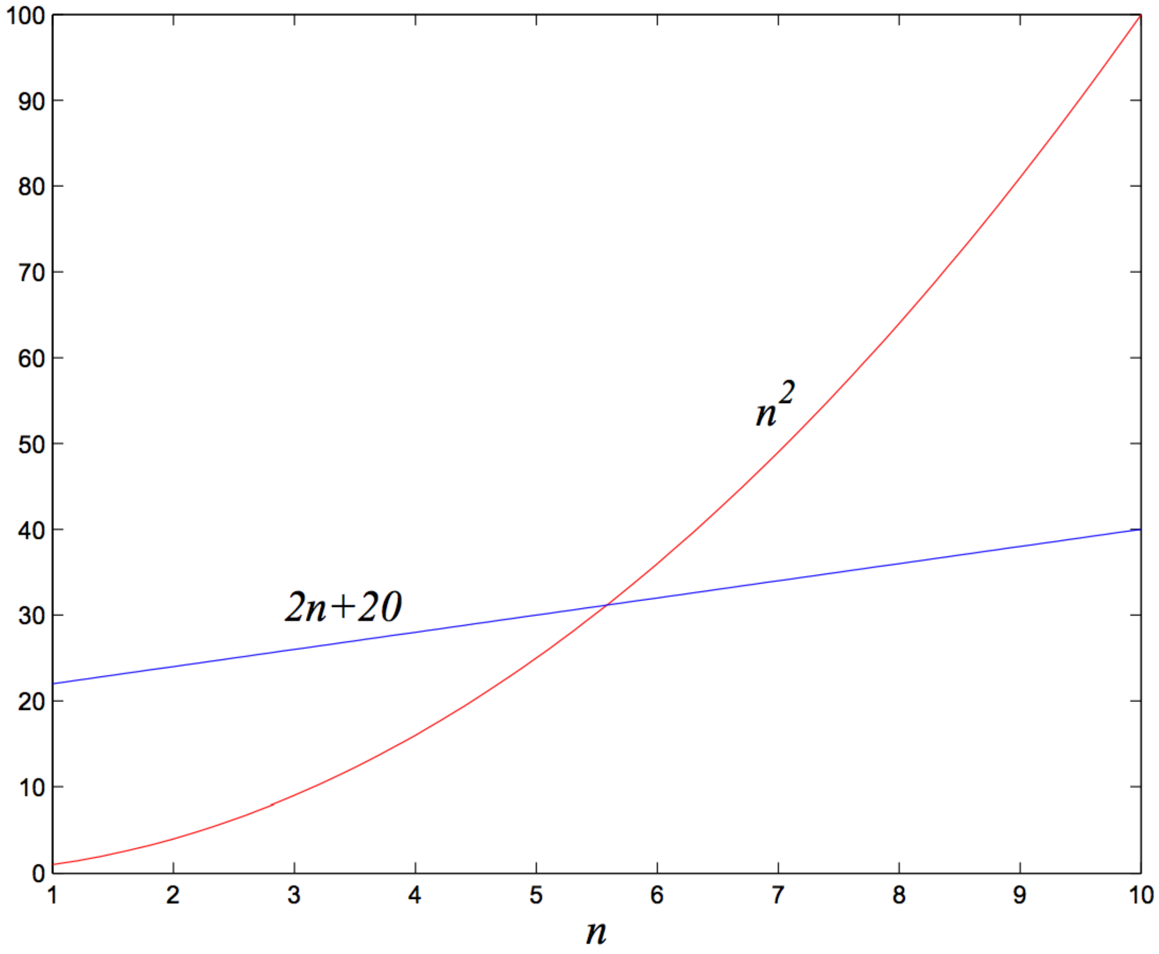
- Cumulative but the focus will be on
  - BST
  - Running time analysis

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"



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)

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

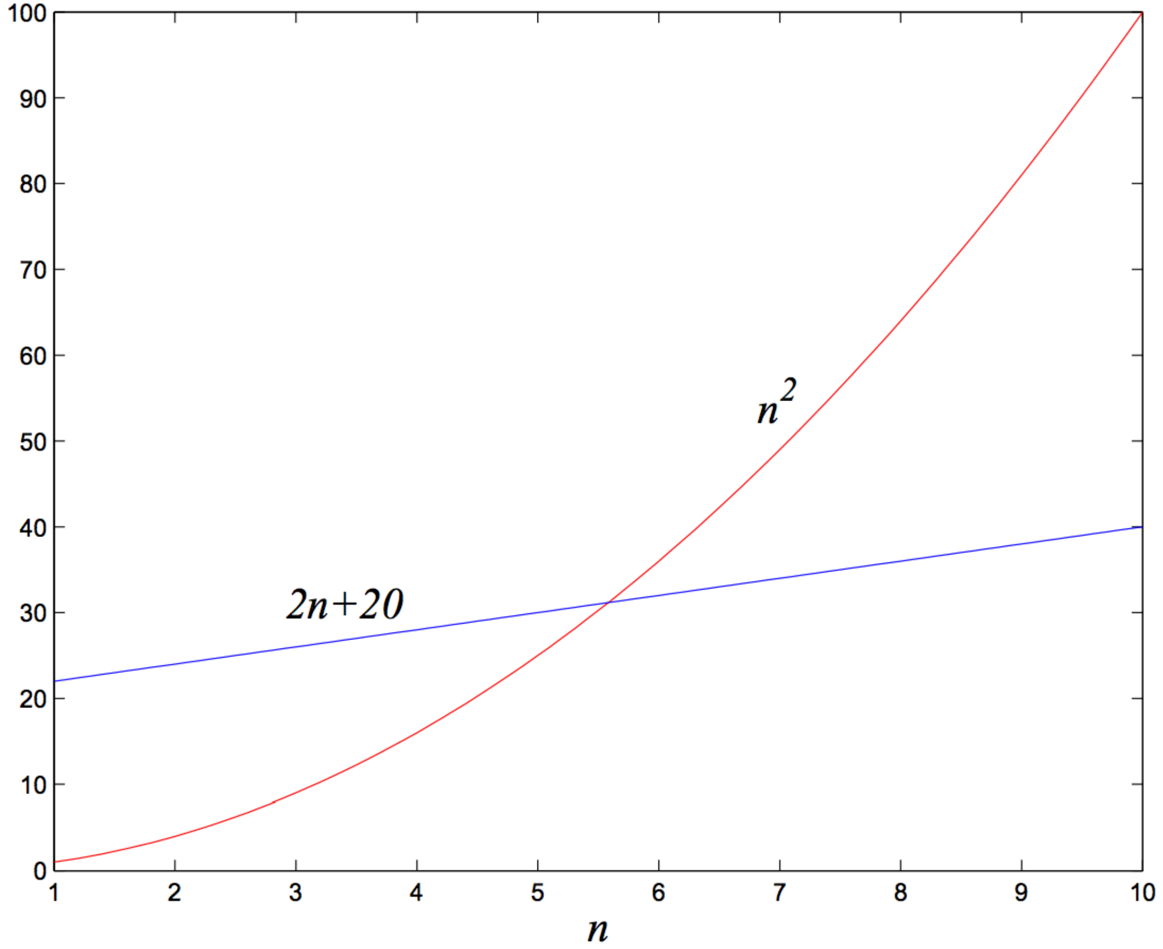
Big-Omega

- 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"

g is a lower bound



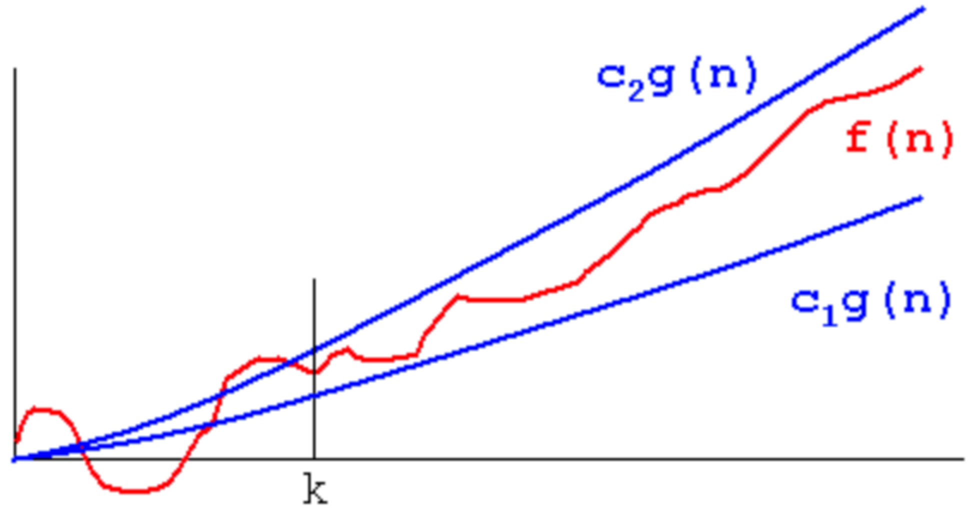
Big-Theta

- 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 $n >= k$

f and g grow at the same rate

Running time



Problem Size (n)

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;
}
```

Binary Search Trees

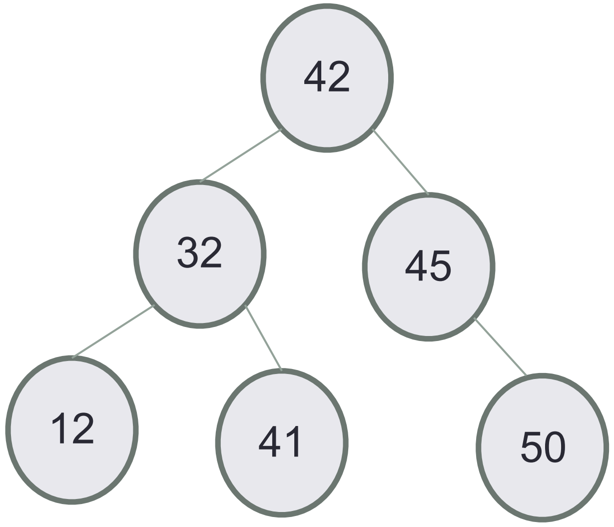
- WHAT are the operations supported?
- HOW do we implement them?
- WHAT are the (worst case) running times of each operation?

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.

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

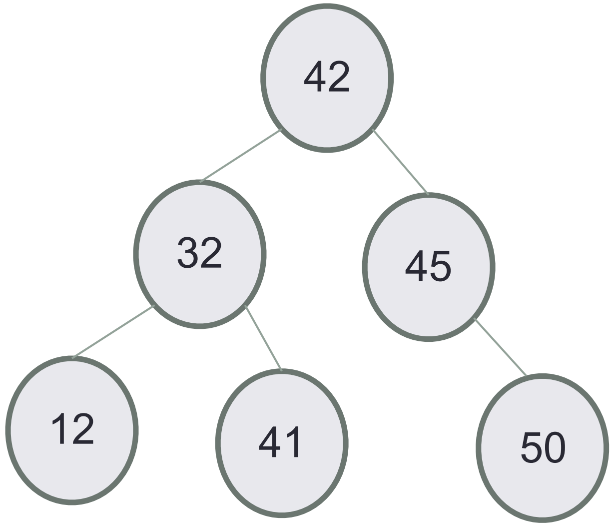
Worst case Big-O of search



- 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)

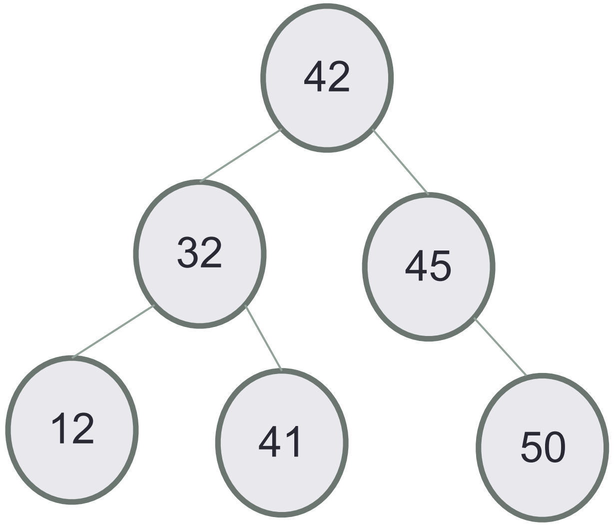
Worst case Big-O of insert



- 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)

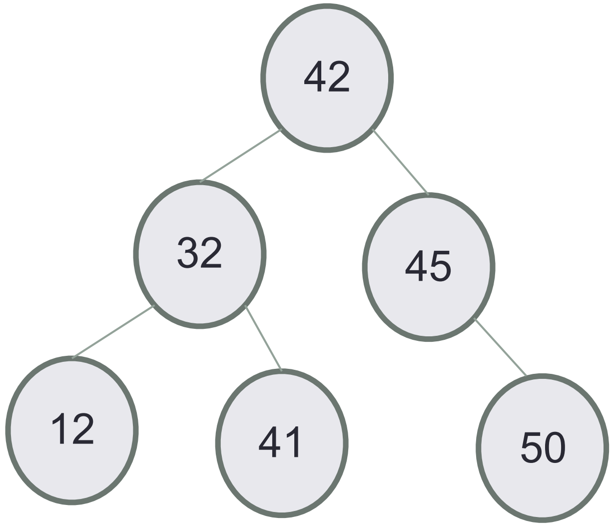
Worst case Big-O of min/max



- 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)

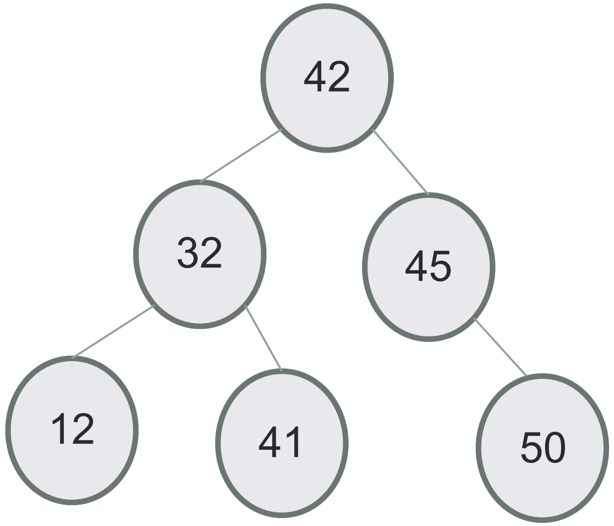
Worst case Big-O of predecessor/successor



- 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



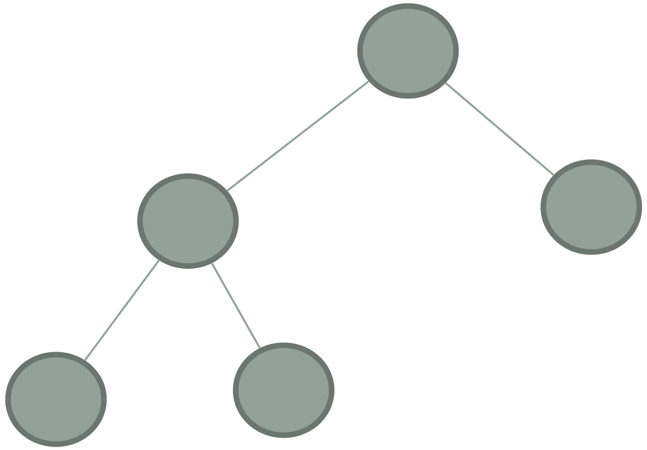
- Given a BST of height H and N nodes, what is the worst case complexity of deleting the key (assume no duplicates)?

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

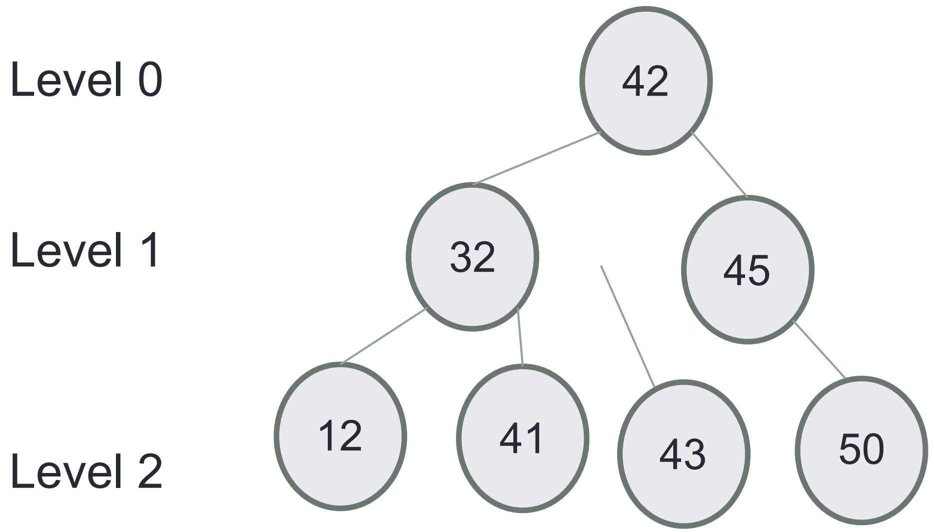
Worst case analysis

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

- A. Yes
- B. No

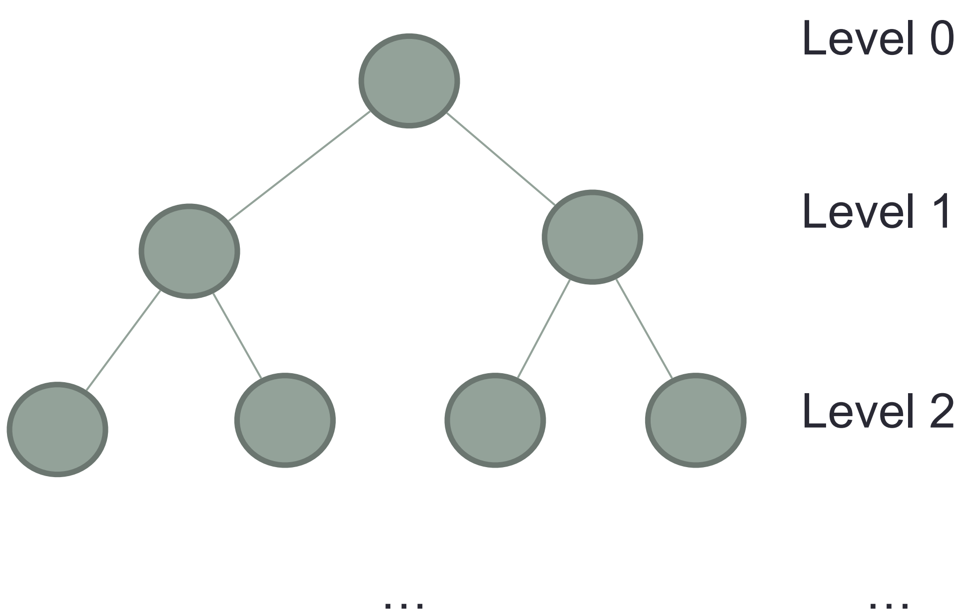


Completely filled binary tree



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

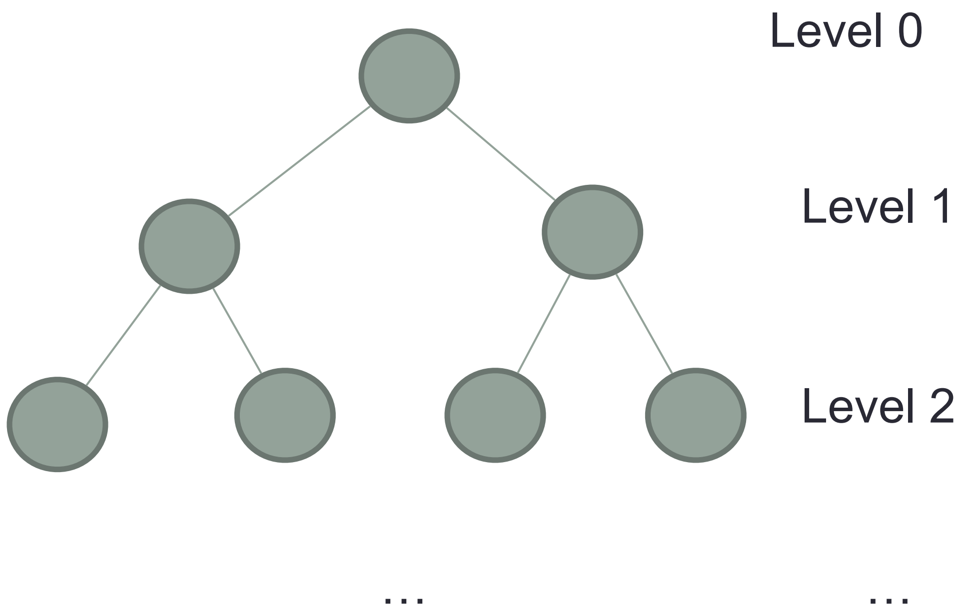
Relating H (height) and N (#nodes)
find is O(H), we want to find a f(N) = H



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



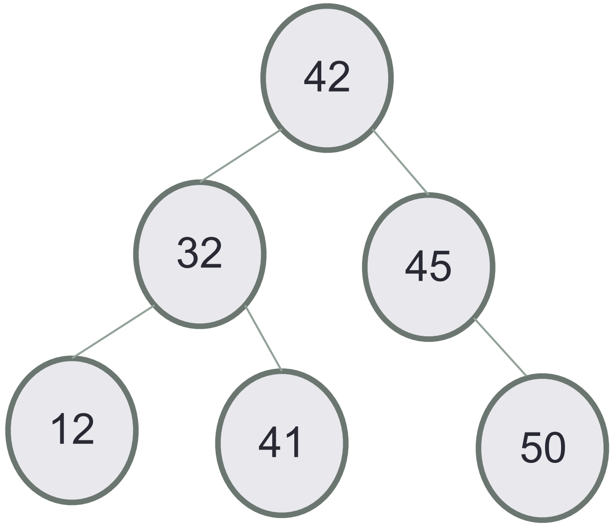
Finally, what is the height (exactly) of the tree in terms of N?

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 of traversals



In Order: O(N)

Pre Order: O(N)

Post Order: O(N)

Summary of operations

Operation	Sorted Array	Balanced Binary Search Tree	Linked List
Min	O(1)	O(log N)	O(N)
Max	O(1)	O(log N)	O(N)
Median	O(1)	?, maybe O(N)	?
Successor	O(1)	O(log N)	?
Predecessor	O(1)	O(log N)	?
Search	O(log N)	O(log N)	O(N)
Insert	O(N)	O(log N)	O(1) if it's at the front, O(N) otherwise
Delete	O(N)	O(log N)	O(N) to search, O(1) to delete and rearrange pointers