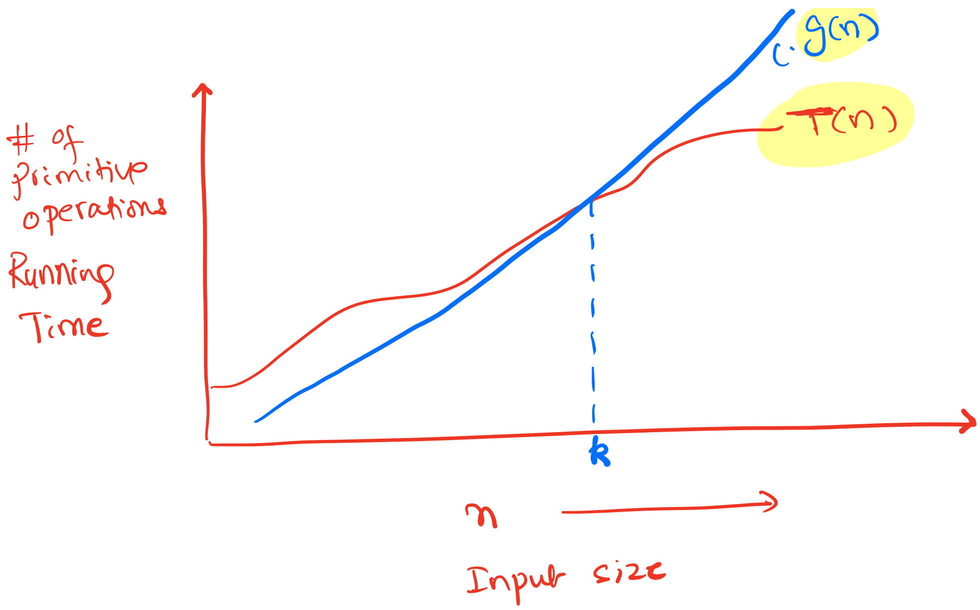
RUNNING TIME ANALYSIS - PART 2

Problem Solving with Computers-II

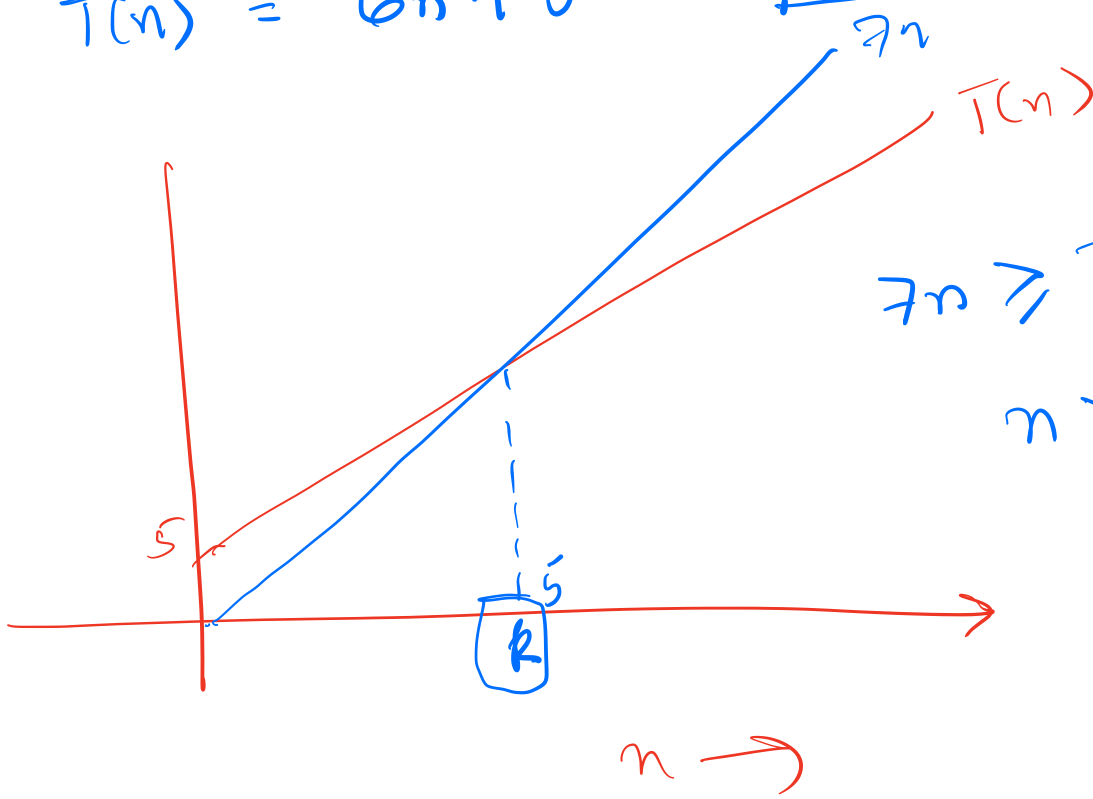


$$T(n) = O(g(n))$$

$c \cdot g(n) \geq T(n)$ for al $n \geq k$

$$\boxed{T(n) = O(n)}$$

$$T(n) = 6n + 5$$



$7n \geq T(n)$

$n \geq k$

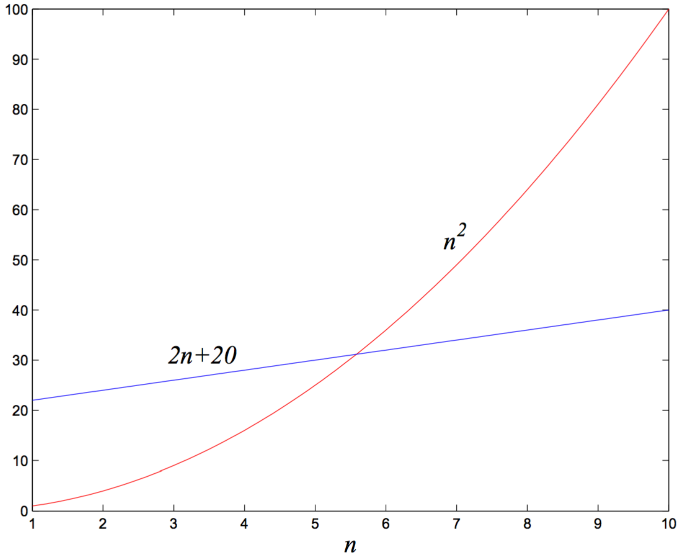
Definition of Big-O

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"

$T(n) = O(g(n))$



What is the Big O running time of sumArray2

A. $O(n^2)$
B. $O(n)$
C. $O(n/2)$
D. $O(\log n)$
E. None of the array

```
/* n is the length of the array*/
int sumArray2(int arr[], int n)
{
        int result = 0;
        for(int i=0; i < n; i=i+2)
            result+=arr[i];
        return result;
}
```

1 2 3

$T(n) = 1 + 1 + \#$ of time the loop runs $* 6 + 1$

$= 3 + \frac{n}{2} * 6 = O(n)$

What is the Big O of sumArray3

A. $O(n^2)$
B. O(n)
C. O(n/2)
D. O(log n)
E. None of the array

```
/* N is the length of the array*/
int sumArray3(int arr[], int n)
{
        int result = 0;
        for(int i= 1; i < n; i=i*2)
            result+=arr[i];
        return result;
}
```

T(n) = O(1) + # time the loop runs · O(1)

log n · O(1)

= O(1) + O(log n)

Iteration #	i
1	1
2	2
3	4
$\vdots$	$\vdots$
$\boxed{k}$	2^{k-1}

Loop exits:

$$2^{k-1} \geq n$$

$$k-1 \geq \log_2 n$$

$$k \geq \log_2 n + 1$$

$$T(n) = c_1 + (\log_2 n + 1)\, c_2$$

$$= c_1 + c_2 + c_2 \cdot \boxed{\log_2 n}$$

$$= O(\log n)$$

Given the step counts for different algorithms, express the running time complexity using Big-O

1. 10,000,000 — $O(1)$
2. 3*n — $O(n)$
3. 6*n-2 — $O(n)$
4. 15*n + 44 — $O(n)$
5. 50*n*log(n) — $O(n \log n)$
6. n^2 — $O(n^2)$
7. n^2-6n+9 — $O(n^2)$
8. $3n^2+4*\log(n)+1000$ = $O(n^2)$

For polynomials, use only leading term, ignore coefficients: linear, quadratic

Common sense rules of Big-O

1. Multiplicative constants can be omitted: $14n^2$ becomes n^2 .
2. n^a dominates n^b if $a > b$: for instance, n^2 dominates n.
3. Any exponential dominates any polynomial: 3^n dominates n^5 (it even dominates 2^n).

$$3^n > n^5$$

$$T(n) = 3^n + n^5$$

$$= O(3^n)$$

Best case and worst case running times

Operations on sorted arrays of size n

	Best case	Worst case
• Min :	$O(1)$	$O(1)$
• Max:	$O(1)$	$O(1)$
• Median:	$O(1)$	$O(1)$
• Successor:	$O(1)$	$O(1)$
• Predecessor:	$O(1)$	$O(1)$
• Search (key) — naive search :	$O(1)$	$O(n)$
binary search :	$O(1)$	$O(\log n)$
• 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

begin mid begin end

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;
}
```

$O(1)$

$O(1)$

Iteration	end - begin
1	$n-1$
2	$\frac{n-1}{2}$
3	$\frac{n-1}{4}$
...	
k	$\frac{n-1}{2^{k-1}}$

Stop $\quad (end - begin) < 1$

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

$$n-1 < 2^{k-1}$$

$$\log_2 (n-1) < k-1$$

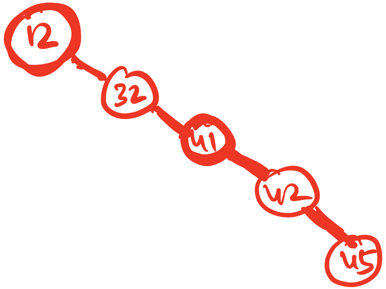
$$k > \log_2 (n-1) + 1$$

$$T(n) = O(1) + (\log(n-1) + 1) * O(1)$$

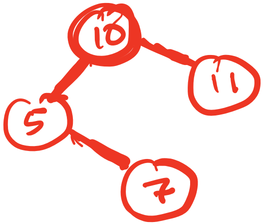
$$= O(\log_2 n)$$

- 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.

Case 1.



Height = 0



Height = 2

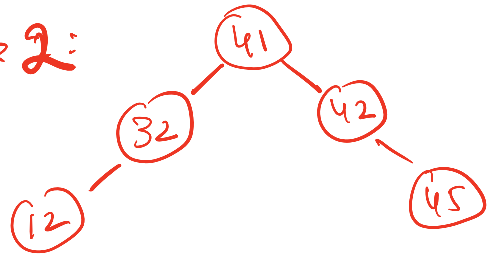
Height = 4

Height of the root = Height of tree

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

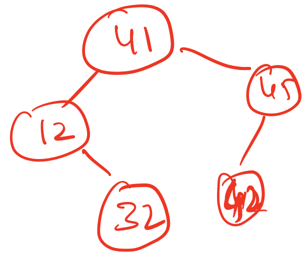
Examples for keys: 12, 32, 41, 42, 45

Case 2:



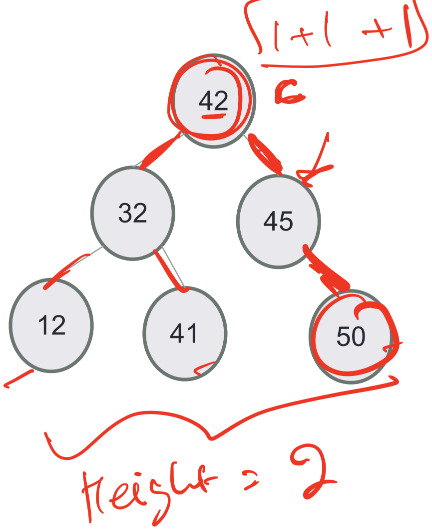
Height = 2

Case 3:



Height = 2

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



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)

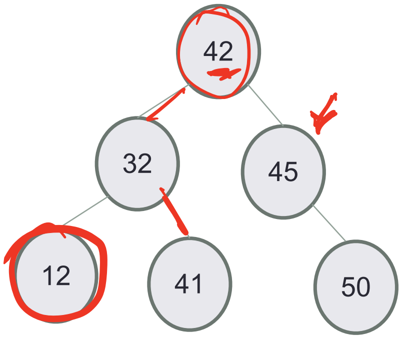
D. O(H*log H)

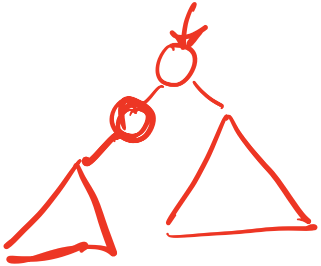
E. O(N)

Best case: O(1)

Worst case: O(H)

Worst case Big-O of predecessor / successor





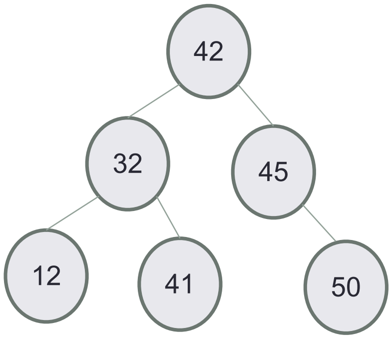
Given a BST of height H and N nodes, what is the worst case complexity of finding the predecessor or successor key?

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

Best case: O(1)
predecessor(45)

worst case: O(H)
predecessor(42)
predecessor(12)

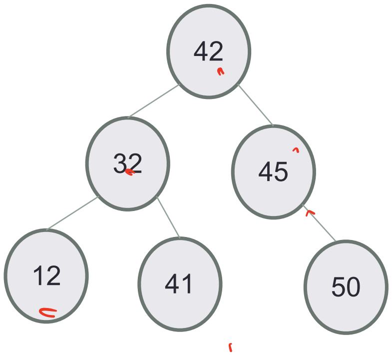
Worst case Big-O of delete



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

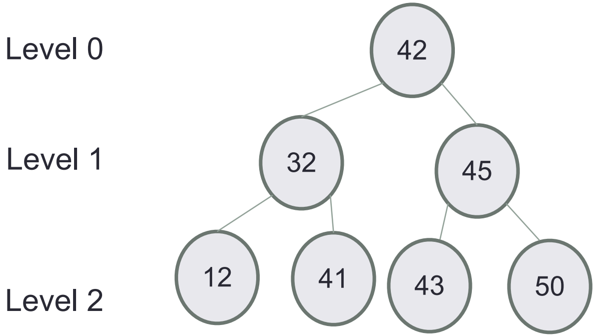


In Order: $O(n)$

Pre Order: $O(n)$

Post Order: $O(n)$

Types of BSTs

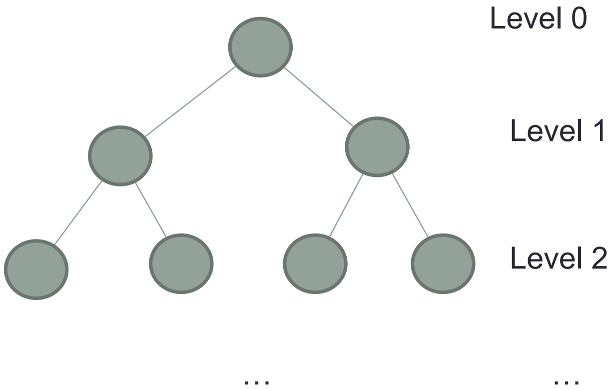


Balanced BST:

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

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

Relating H (height) and N (#nodes)



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