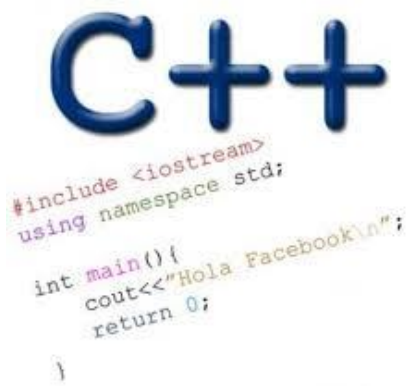
BINARY SEARCH TREES

Problem Solving with Computers-I

https://ucsb-cs24-sp17.github.io/



Imposter panel: Tomorrow Thurs (06/01), 12:30pm to 1:50pm, HFH 1132

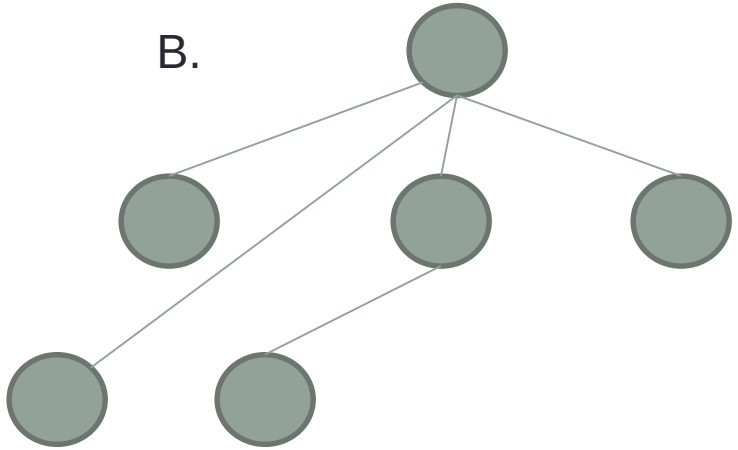
Come hear faculty, grad students and undergrad alumni talk about their careers and how they dealt with feeling like an Imposter!

Please RSVP : https://goo.gl/forms/ttvzHNPWAZ0GCPA92

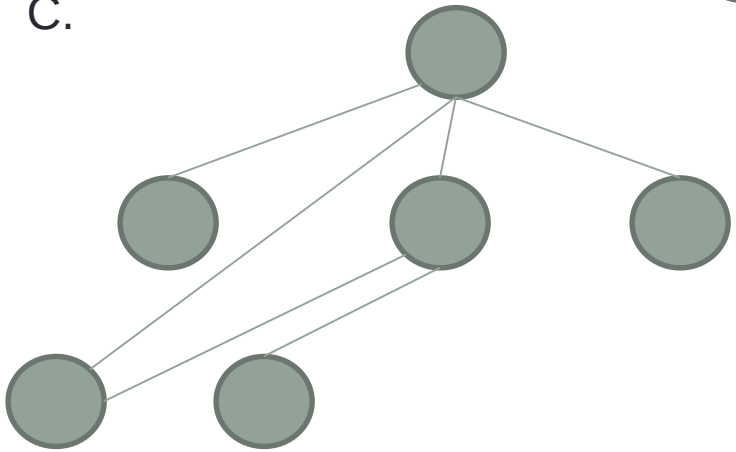
Which of the following is/are a tree?

A.

B.

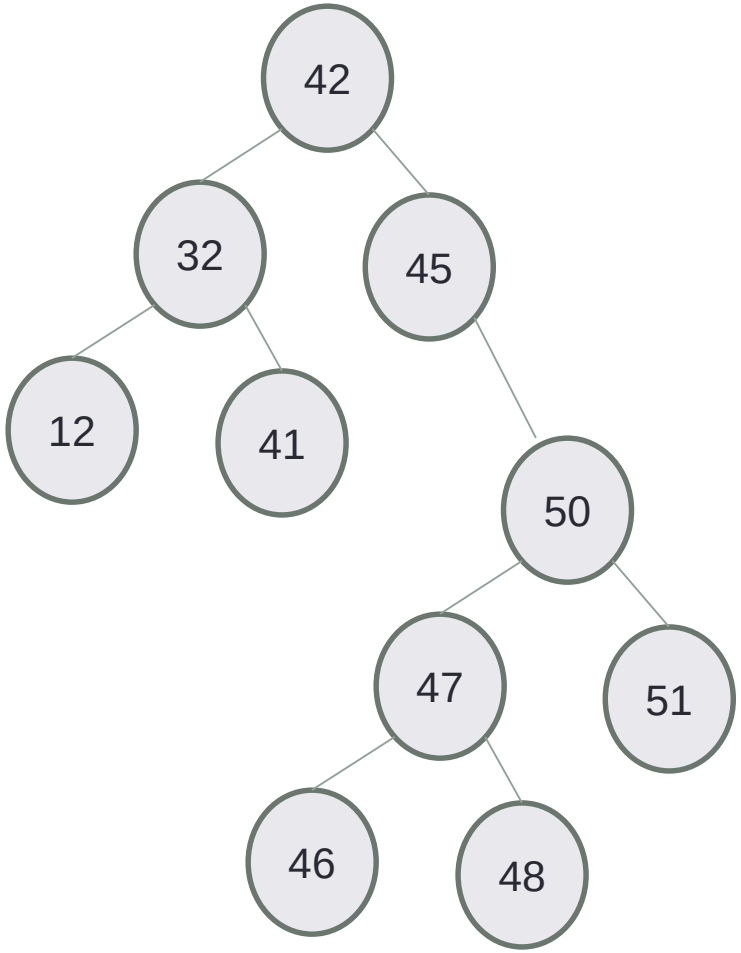


C.



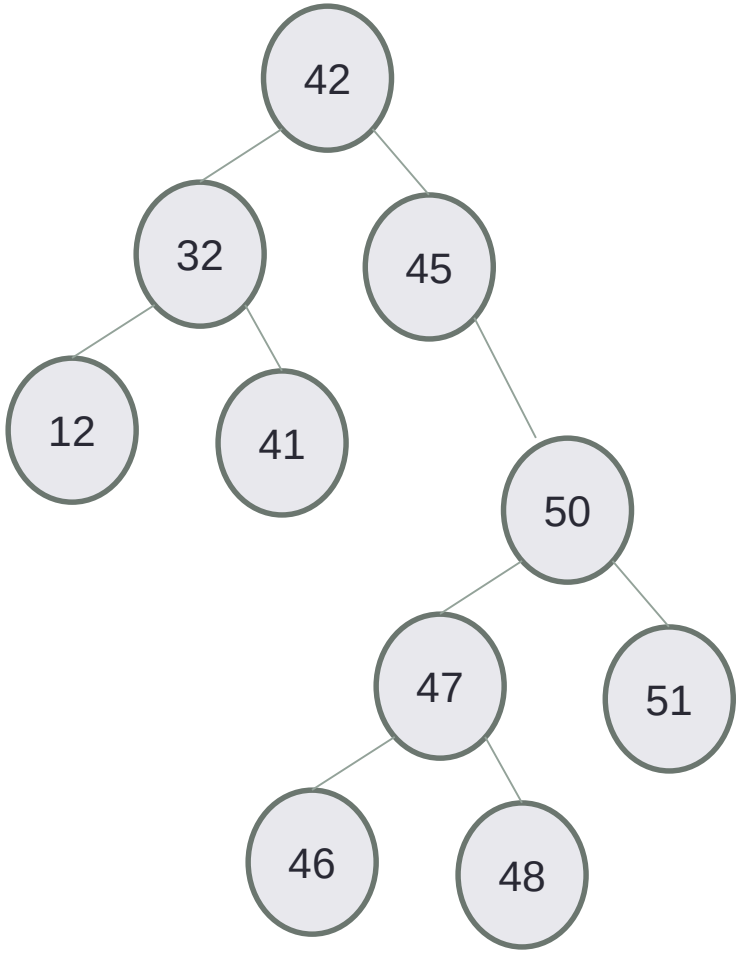
D. A & B
E. All of A-C

Lab08: Binary Search Tree – What is it?



What are the numbers in the nodes?

Binary Search Tree – What is it?

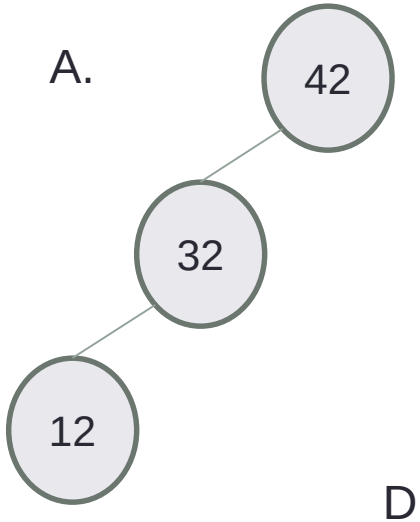


For any node,
Keys in node's left subtree < Node key
Node key < Keys in node's right subtree

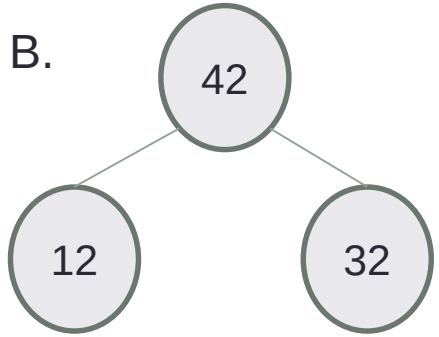
Do the keys have to be integers?

Which of the following is/are a binary search tree?

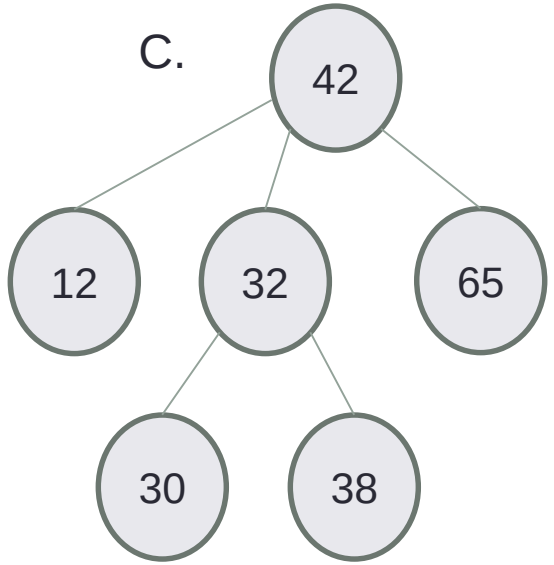
A.



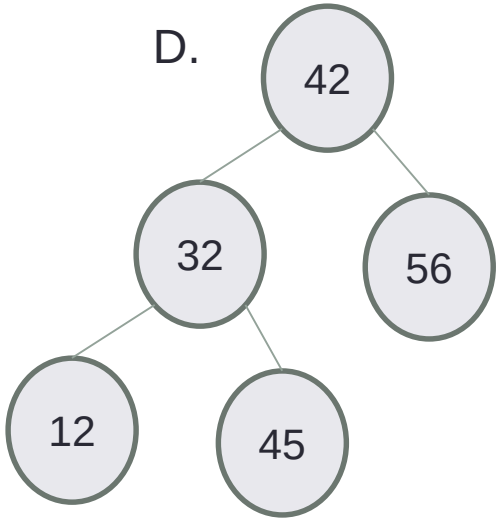
B.



C.



D.



E. More than one of these

Binary Search Trees

- What are the operations supported?
- What are the running times of these operations?
- How do you implement the BST i.e. operations supported by it?

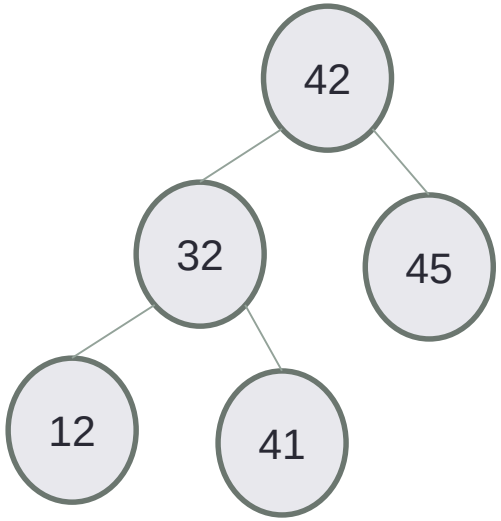
Binary Search Trees

- What is it good for?
  - If it satisfies a special property i.e. Balanced, you can think of it as a dynamic version of the sorted array

Under the hood: Searching an element in the BST

To search for element with key k

1. Start at the root
2. If k=key(root), found key, stop.
3. Else If k< key(root), recursively search the left subtree: T_L
   Else recursively search the right subtree: T_R

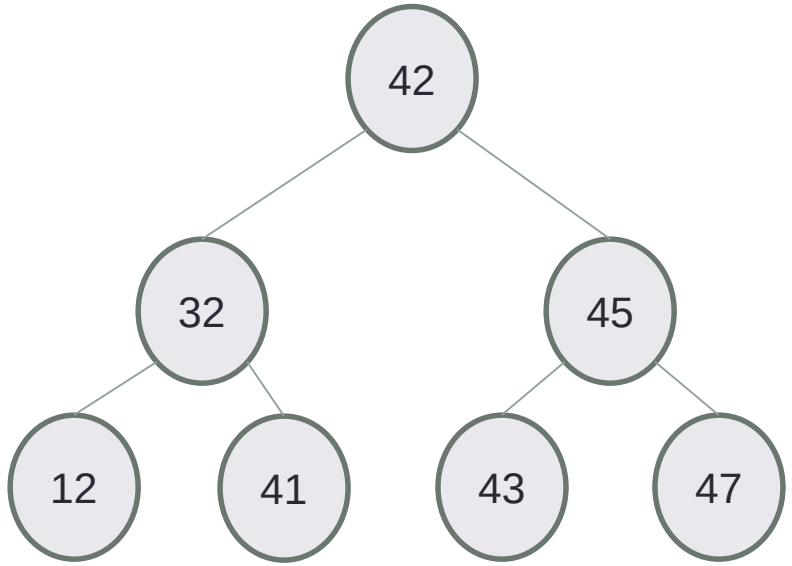


Search for 41.
Now search for 43.

Traversing the BST

Different methods of tree traversal:

- Pre order traversal
- Post order traversal
- In order traversal



BST, with templates:

```
template<typename Data>

class BSTNode {
public:
  BSTNode<Data>* left;
  BSTNode<Data>* right;
  BSTNode<Data>* parent;
  Data const data;

  BSTNode( const Data & d ) :
     data(d) {
    left = right = parent = 0;
  }

};
```

BST, with templates:

```
template<typename Data>

class BSTNode {
public:
  BSTNode<Data>* left;
  BSTNode<Data>* right;
  BSTNode<Data>* parent;
  Data const data;

  BSTNode( const Data & d ) :
     data(d) {
    left = right = parent = 0;
  }

};
```

How would you create **a BSTNode object** on the runtime stack?

A. BSTNode n(10);
B. BSTNode<int> n;
C. BSTNode<int> n(10);
D. BSTNode<int> n = new BSTNode<int>(10);
E. More than one of these will work

{ } syntax OK too

BST, with templates:

```
template<typename Data>

class BSTNode {
public:
  BSTNode<Data>* left;
  BSTNode<Data>* right;
  BSTNode<Data>* parent;
  Data const data;

  BSTNode( const Data & d ) :
     data(d) {
    left = right = parent = 0;
  }

};
```

How would you create a **pointer** to BSTNode with integer data?

A. BSTNode* nodePtr;
B. BSTNode<int> nodePtr;
C. BSTNode<int>* nodePtr;

BST, with templates:

```
template<typename Data>

class BSTNode {
public:
  BSTNode<Data>* left;
  BSTNode<Data>* right;
  BSTNode<Data>* parent;
  Data const data;

  BSTNode( const Data & d ) :
     data(d) {
    left = right = parent = 0;
  }

};
```

Complete the line of code to create a new BSTNode object with int data on the heap and assign nodePtr to point to it.

```
BSTNode* nodePtr
```

Working with a BST

```
template<typename Data>
class BST {

private:

  /** Pointer to the root of this BST, or 0 if the BST is empty */
  BSTNode<Data>* root;

public:

  /** Default constructor. Initialize an empty BST. */
  BST() : root(nullptr){  }

  void insertAsLeftChild(BSTNode<Data>* parent, const Data & item)
  {
     // Your code here
  }
```

Working with a BST: Insert

```
void insertAsLeftChild(BSTNode<Data>* parent, const Data & item)
  {
     // Your code here
  }
```

Which line of code correctly inserts the data item into the BST as the left child of the parent parameter.

A. `parent.left = item;`
B. `parent->left = item;`
C. `parent->left = BSTNode(item);`
D. `parent->left = new BSTNode<Data>(item);`
E. `parent->left = new Data(item);`

Working with a BST: Insert

```
void insertAsLeftChild(BSTNode<Data>* parent, const Data & item)
{
    parent->left = new BSTNode<Data>(item);
}
```

Is this function complete? (i.e. does it to everything it needs to correctly insert the node?)

A. Yes. The function correctly inserts the data
B. No. There is something missing.

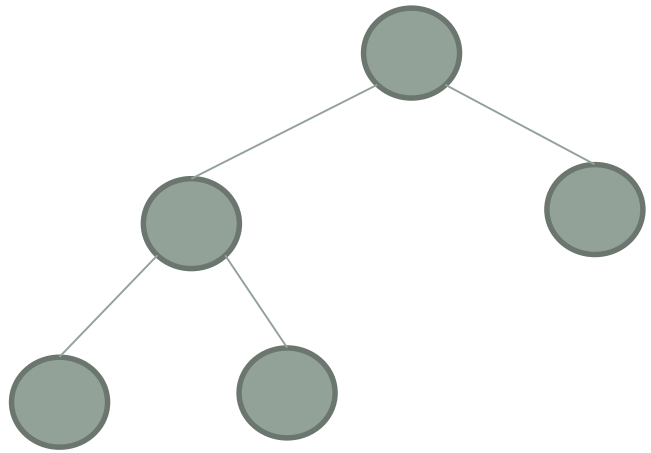
Working with a BST: Insert

```
void insertAsLeftChild(BSTNode<Data>* parent, const Data & item)
{


    parent->left = new BSTNode<Data>(item);



}
```

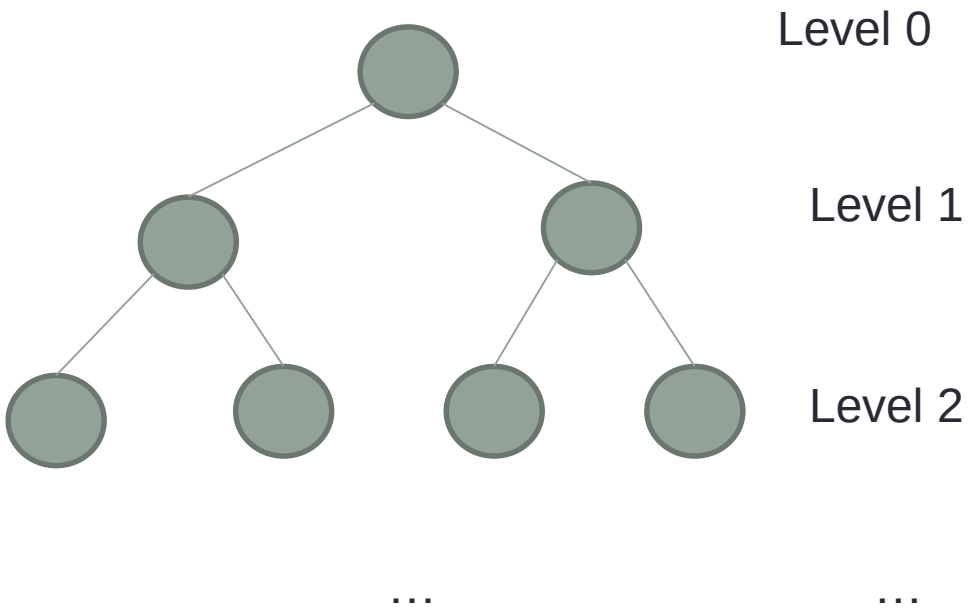
How fast is BST find algorithm?

How long does it take to find an element in the tree in terms of the tree's height, H?

Height of a node: the height of a node is the number of edges on the longest path from the node to a leaf
Height of a tree: the height of the root of the tree



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

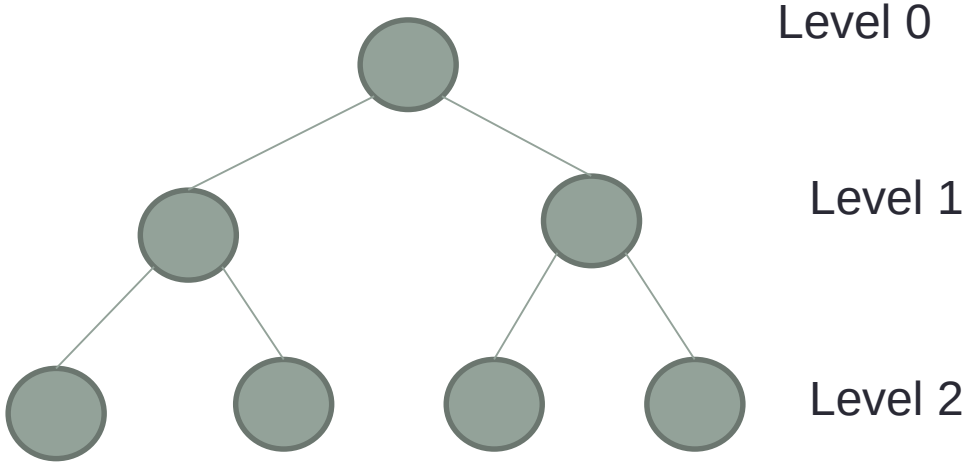


How many nodes are on level L in a completely filled binary search tree?

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

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



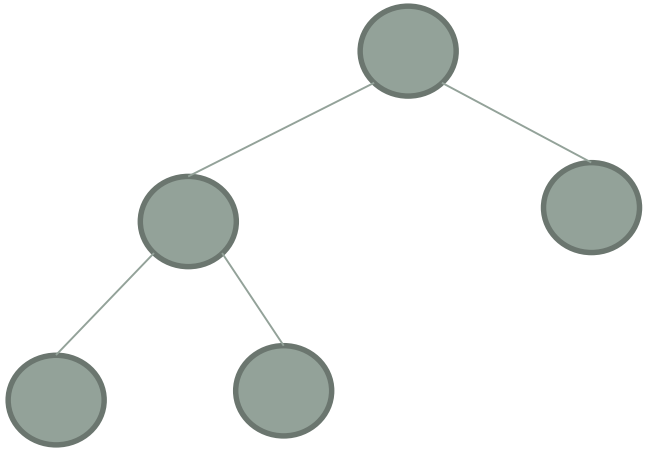
Finally, what is the height (exactly) of the tree in terms of N?

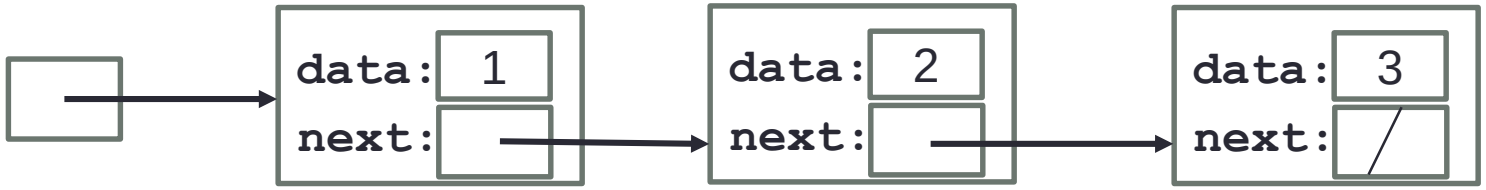
$$H = \log_2(N+1) - 1$$

And since we knew finding a node was O(H), we now know it is O($\log_2$ N)

Worst case analysis

- Are binary search trees *really* faster than linked lists for finding elements?
- A. Yes
- B. No



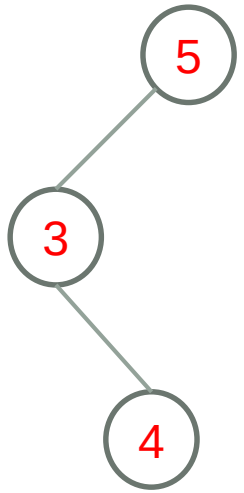


Average case analysis of a "successful" find

Given a BST having N nodes x_1, .. $x_{N,}$ such that key(x_i) = k_i

How many compares to locate a key in the BST?

1. Worst case:
2. Best case:
3. Average case:



Here is the result! Proof is a bit involved but if you are interested in the proof, come to office hours

$D_{avg}(N)$	Average #comparisons to find a single item in any BST with N nodes

$$D_{avg}(N) \approx 1.386 \log_2 N$$

Conclusion: The average time to find an element in a BST with no restrictions on shape is Θ(log N).