VE281

Data Structures and Algorithms

Binary Search Trees

Course Evaluation

- For instructor to improve the teaching, students' feedbacks are very important.

- JI will use an online evaluation system called "IDEA," starting this semester for every course.
  - Evaluation period: from Nov. 19th to Dec. 16th.
  - IDEA will send an email to your **SJTU email account**, which gives you instructions. Please check your SJTU email.
  - All responses are **anonymous** and **voluntary**.
  - I value your feedback. Please provide your feedback objectively. I welcome your written comments.

Review

- Binary Tree Traversal
- Depth-first traversal
  - Pre-order
  - Post-order
  - In-order
  - Implemented with recursion
  - Implemented with a stack
- Level order traversal
  - Implemented with a queue

Outline

- Binary Search Trees
  - search, insertion, removal
- Average Case Time Complexity

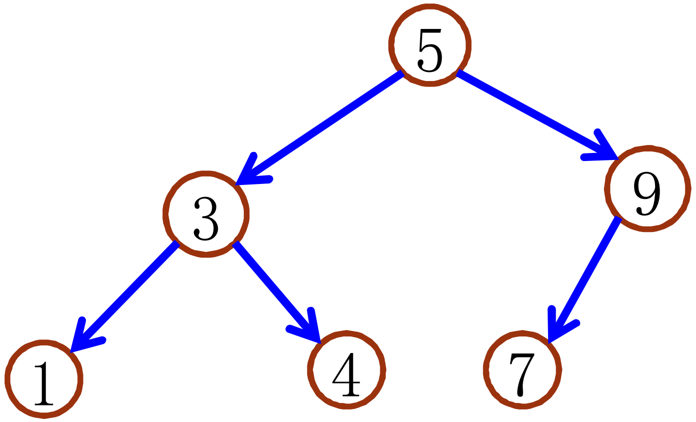
Binary Search Tree

- A binary search tree (BST) is a binary tree with the following properties:
  - Each node is associated with a **key**. A key is a value than can be compared.
  - The key of any node is greater than the keys of all nodes in its left subtree and smaller than the keys of all nodes in its right tree.

- A BST allows search, insertion, and removal by key.
  - The **average case** time complexities for these operations are $O(\log n)$.

Binary Search Tree

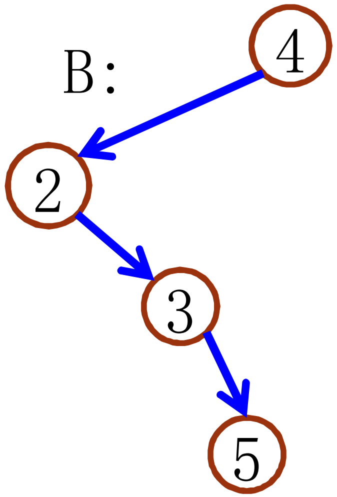
Example



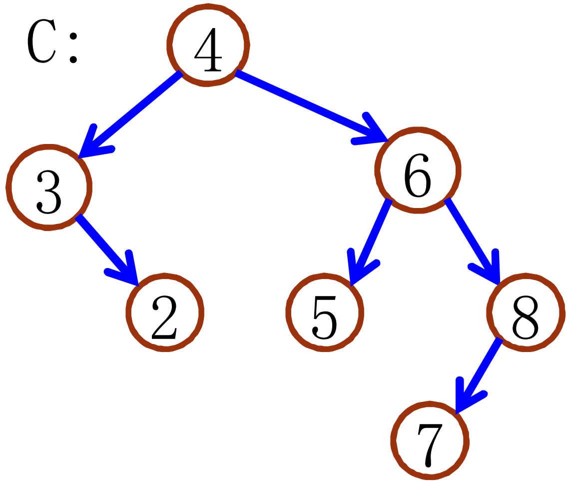
Exercise: which of the following trees are BST?

A:

B:



C:



Binary Search Tree

Search

```
node *search(node *root, Key k)
// EFFECTS: return the node whose key is k.
// If no matching node, return NULL.
```

- Procedure: Compare the search key with the key of the root
  - If they are equal, return the root.
  - If search key < root key, search the left subtree.
  - If search key > root key, search the right subtree.
  - Recursively applying the above procedure.

Binary Search Tree

Search

```
struct node {
  Item item;
  node *left;
  node *right;
};
```

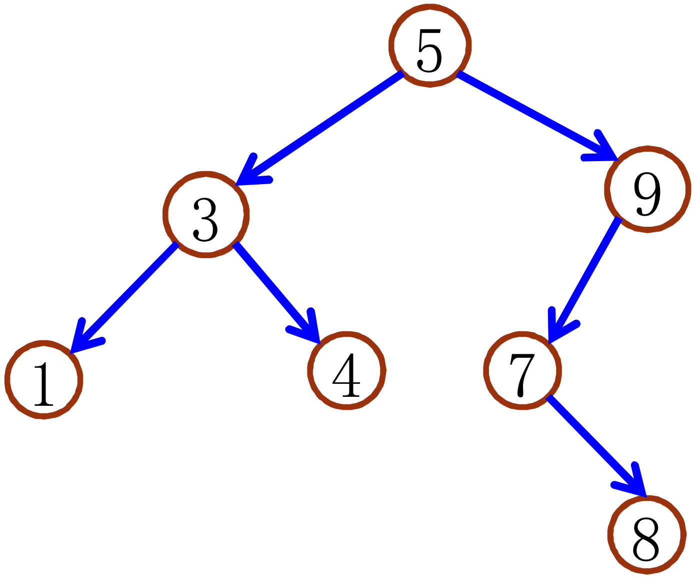
```
struct Item {
  Key key;
  Val val;
};
```

```
node *search(node *root, Key k) {
  if(root == NULL) return NULL;
  if(k == root->item.key) return root;
  if(k < root->item.key)
    return search(root->left, k);
  else return search(root->right, k);
}
```

Binary Search Tree

Insertion

- Insertion inserts the item **as a leaf** of the BST.
- It inserts at a proper location in the BST, maintaining the BST properties.



Insert a node with key =

Binary Search Tree

Insertion

```
void insert(node *&root, Item item)
// EFFECTS: insert the item as a leaf,
// maintaining the BST property.
{
  if(root == NULL) {
    root = new node(item);
    return;
  }
  if(item.key < root->item.key)
    insert(root->left, item);
  else if(item.key > root->item.key)
    insert(root->right, item);
}
```

Question: why define root as the

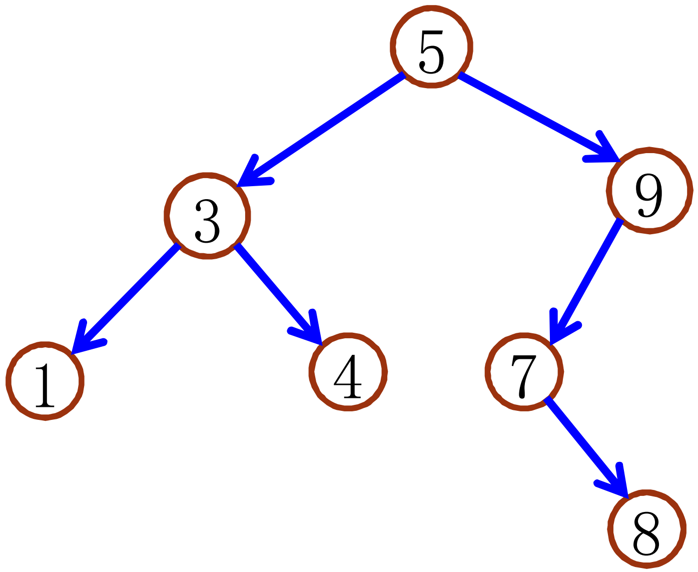
Question: what happens if the key is already in the BST?

Binary Search Tree

Removal

```
void remove(node *&root, Key k) {
  if(root == NULL) return;
  if(k < root->item.key) remove(root->left, k);
  else if(k > root->item.key)
    remove(root->right, k);
  else { // root->item.key == k
    // What to do when root->item.key == k?
  }
}
```

- How will you remove 8?
- How will you remove 9?
- How will you remove 5?



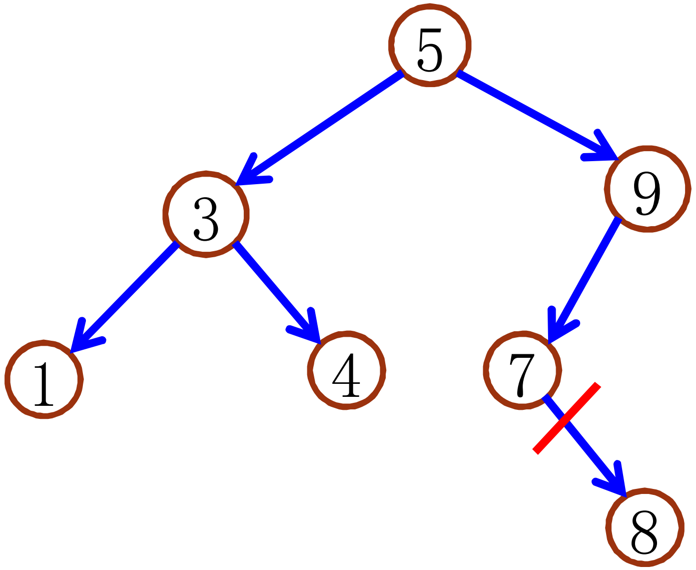
Binary Search Tree

Removal

- We distinguish three cases:
  - Node to be removed is a leaf.
  - Node to be removed is a degree-one node.
  - Node to be removed is a degree-two node.

Remove A Leaf

- Remove node 8

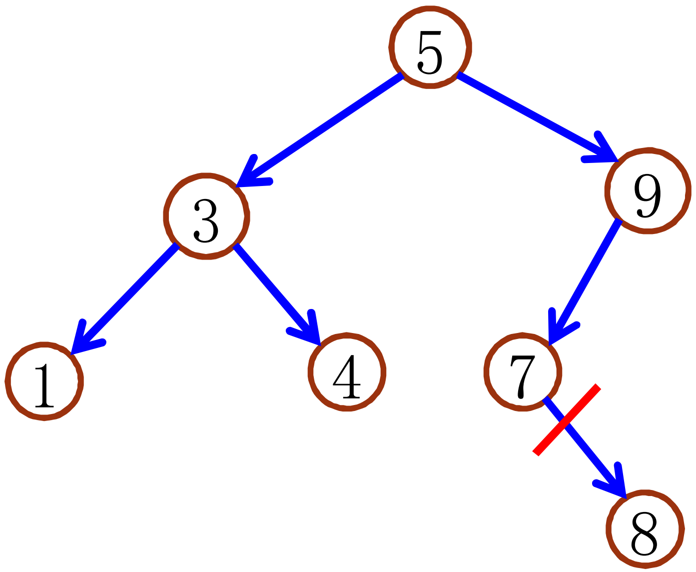


Remove A Leaf

Code

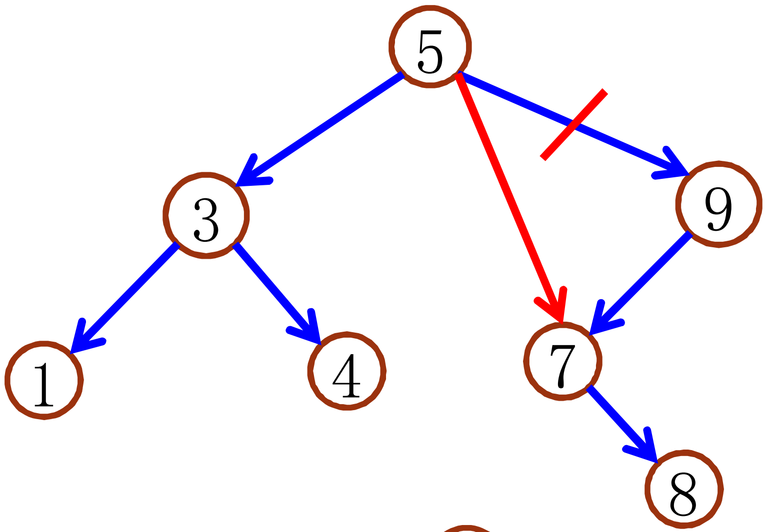
```
else { // root->item.key == k
  if(isLeaf(root)) {
    delete root;
    root = NULL;
  }
  else { // remove degree-one or two node
    ...
  }
}
```

Note: **root** is a reference to a pointer, which could be its parent’s **left** pointer or **right** pointer. Our code effectively changes that pointer to NULL

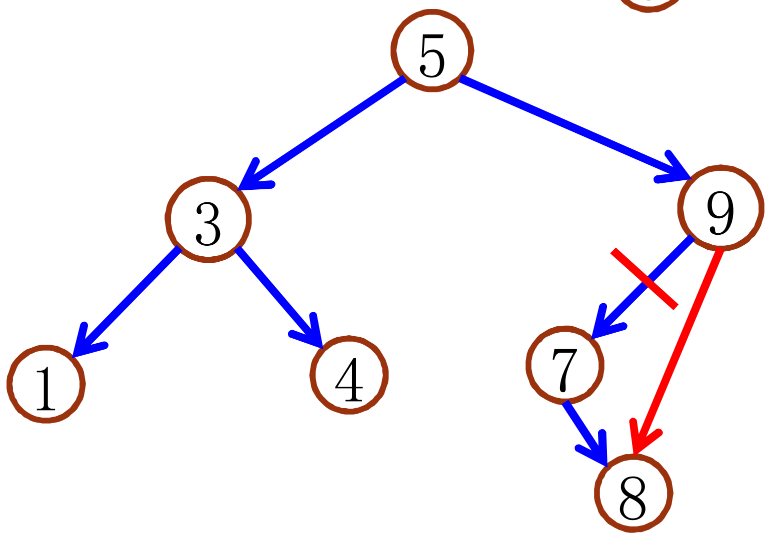


Remove A Degree-One Node

- Remove node 9



- Remove node 7



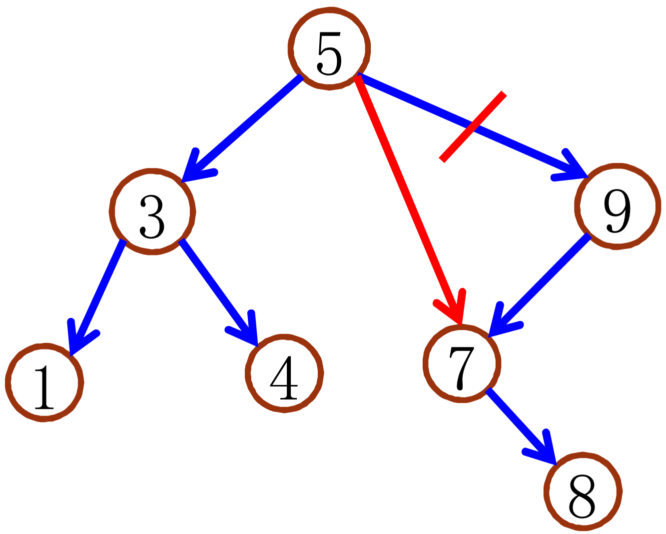
Remove A Degree-One Node

Code

```
else { // remove degree-one or two node
  if(root->right == NULL) { // no right child
    node *tmp = root;
    root = root->left;
    delete tmp;
  }

}
```

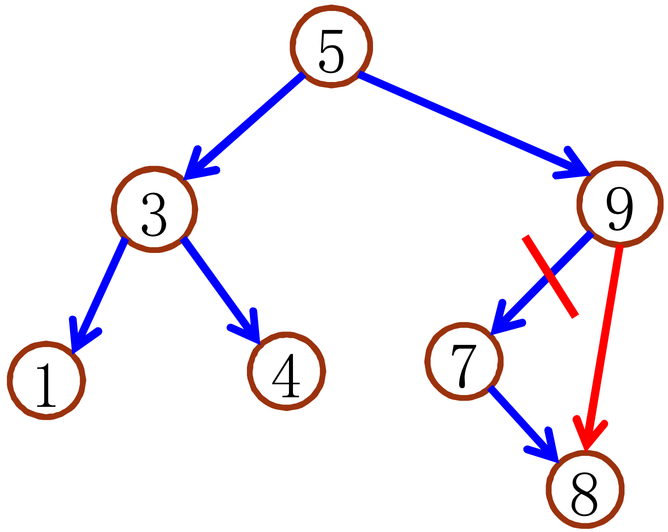
Note the order!



Remove A Degree-One Node

Code

```
else { // remove degree-one or two node
  if(root->right == NULL) { // no right child
    node *tmp = root;
    root = root->left;
    delete tmp;
  }
  else if(root->left == NULL) { // no left child
    node *tmp = root;
    root = root->right;
    delete tmp;
  }
  else {
  // remove degree-two node
  }
}
```

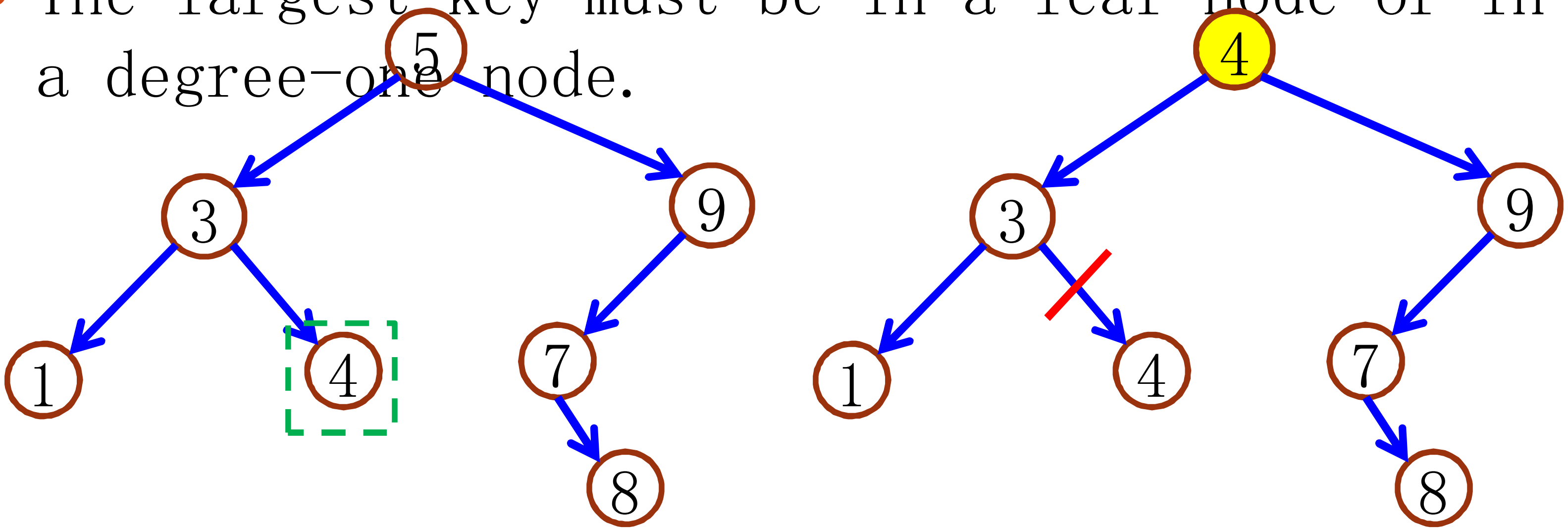


Remove A Degree-Two Node

- Remove node 5
- Idea: Replace with the largest key in the left subtree.
  - or replace with the smallest key in the right sub

Why?

- The largest key must be in a leaf node or in a degree-one node.

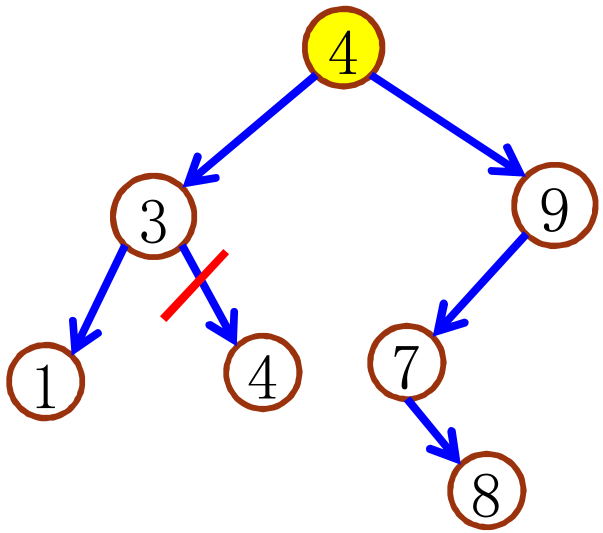


Remove A Degree-Two Node

Code

```
else { // remove degree-two node
  node *&replace = findMax(root->left);
  root->item = replace->item;
  node *tmp = replace;
  replace = replace->left;
  // both leaf and degree-one node are OK
  delete tmp;
}
```

```
node *&findMax(node *&root)
// EFFECTS: return the reference
// to the pointer to the node
// that has the largest key in
// the tree rooted at root
```



Remove A Degree-Two Node

Code

- How do you implement the function **findMax()**?

```
node *&findMax(node *&root) {
  if(root == NULL) return root;
  if(root->right == NULL) return root;
  return findMax(root->right);
}
```

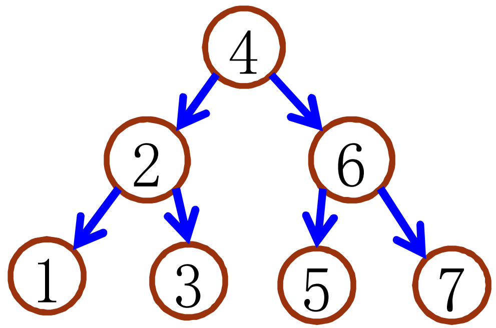
Removal of Binary Search Tree

Summary

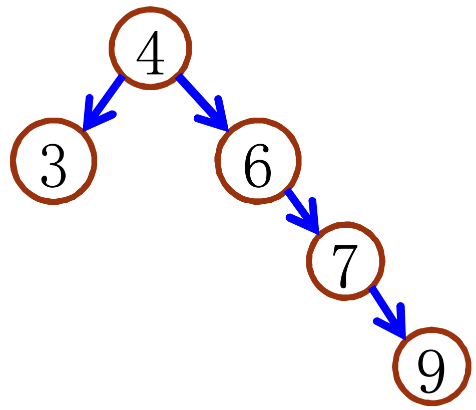
- Node to be removed is a leaf.
  - Delete the node.
- Node to be removed is a degree-one node.
  - “Bypass” the node from its parent to its child.
- Node to be removed is a degree-two node.
  - Replace the node key with the largest key in the left subtree.

Exercise

- Insert 4, 2, 6, 3, 7, 1, 5



- Delete 2, insert 9, delete 5, delete 1



Outline

- Binary Search Trees
  - search, insertion, removal
- Average Case Time Complexity

Complexity Analysis

- If the **depth** of the tree is h, what is the time complexity for a **successful** search in the
  - worst case? $O(h)$
  - average case? $O(h)$

- If the **number of nodes** is n, what is the time complexity for a **successful** search in the
  - worst case? $O(n)$
  - average case?

Average Case Analysis

- If the successful search reaches a node at depth d, the number of nodes visited is $d + 1$.
  - The complexity is $\Theta(d + 1)$.

- Assume that it is equally likely for the object of the search to appear in any node of the search tree. The average complexity is
  - $\Theta(\bar{d} + 1)$
  - $\bar{d}$ is the average depth of the nodes in a given tree

$$\bar{d} = \frac{1}{n}\sum_{i=1}^{n} d_i$$

Internal Path Length

- $\sum_{i=1}^{n} d_i$ is called **internal path length**.
- To get the average case complexity, we need to get the **average** of $\sum_{i=1}^{n} d_i$ for all trees of n nodes.
- Define the **average internal path length** of a tree containing n nodes as $I(n)$.
  - $I(1) = 0$.
- For a tree of n nodes, suppose it has l nodes in its left subtree.
  - The number of nodes in its right subtree is $n - 1 - l$.
  - The average internal path length for such a tree is

$$T(n; l) = I(l) + I(n - 1 - l) + n - 1$$

- $I(n)$ is average of $T(n; l)$ over $l = 0,1, \ldots, n - 1$.

Internal Path Length

- Assume all insertion sequences of n keys $k_1 < \cdots < k_n$ are equally likely.
  - The first key inserted being any k_l are equally likely.
- If the first key inserted is k_{l+1}, the left subtree has l nodes.
- All left subtree sizes are equally likely!

$$
\begin{aligned}
I(n) &= \frac{1}{n}\sum_{l=0}^{n-1} T(n;l) \\
&= \frac{1}{n}\sum_{l=0}^{n-1} \left(I(l) + I(n-1-l) + n - 1\right) \\
&= \frac{2}{n}\sum_{l=0}^{n-1} I(l) + (n-1)
\end{aligned}
$$

Average Case Analysis

- After solving the previous recurrence relation, we can obtain

$$I(n) = \Theta(n \log n)$$

- Thus, the average complexity for a successful search is

$$\Theta\left(\frac{1}{n} I(n)\right) = \Theta(\log n)$$