

VE281

Data Structures and Algorithms

Binary Search Tree Additional Operations

Learning Objectives:

- Know some additional operations of binary search tree that are efficient
- Know how these operations are implemented and their time complexity

Recap: Average-Case Time Complexity

	Search	Insert	Remove
Linked List	$O(n)$	$O(n)$	$O(n)$
Sorted Array	$O(\log n)$	$O(n)$	$O(n)$
Hash Table	$O(1)$	$O(1)$	$O(1)$
BST	$O(\log n)$	$O(\log n)$	$O(\log n)$

So, why we use BST, not hash table?

Why BST?

- Other Operations Supported by BST

Average-Case
Time Complexity

- Output in Sorted Order

$O(n)$

- Get Min/Max

$O(\log n)$

- Get Predecessor/Successor

$O(\log n)$

- Rank Search

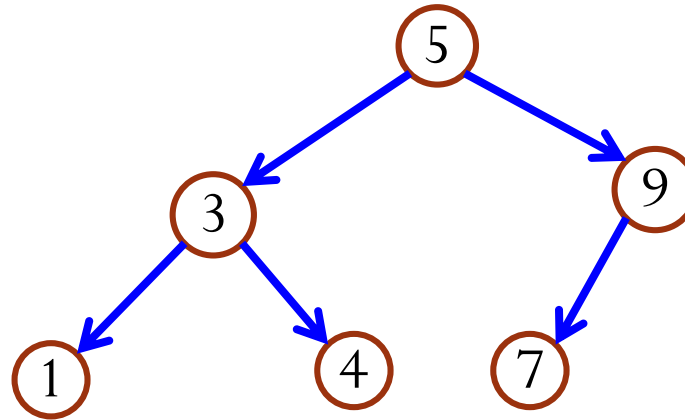
$O(\log n)$

- Range Search

$O(n)$

Note: Hash table does not support efficient implementation of the above methods.

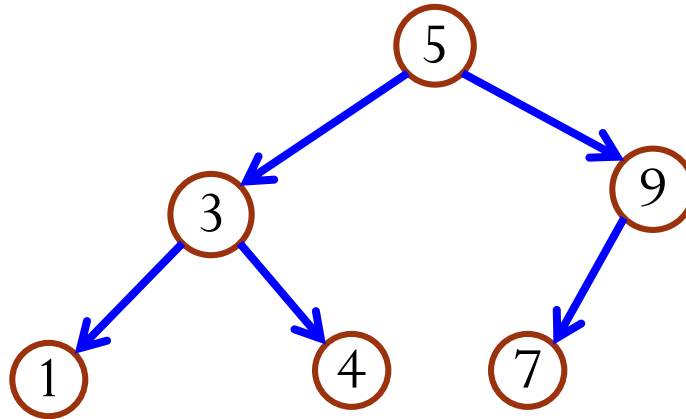
Output in Sorted Order



- Output: 1, 3, 4, 5, 7, 9
- **How?**
 - In-order depth-first traversal.
- Time complexity: $O(n)$.

- Visit the left subtree
- Visit the node
- Visit the right subtree

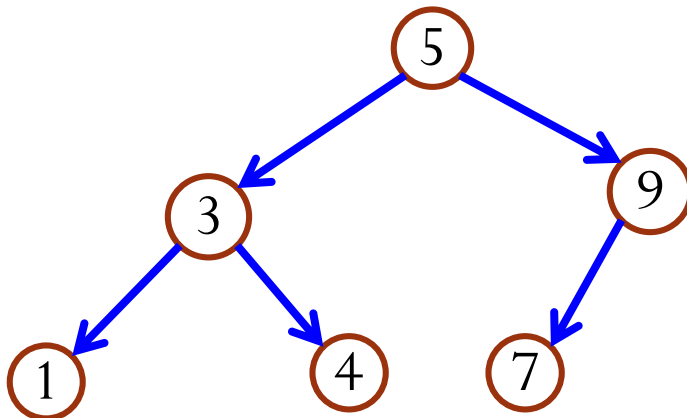
Get Min/Max



- To get **min** (**max**) key of the tree:
 - Start at root.
 - Follow **left** child pointer (**right for max**) until you cannot go anymore.
 - Return the last key found.
- Time complexity? $O(\text{height})$. On average: $O(\log n)$.

Get Predecessor/Successor

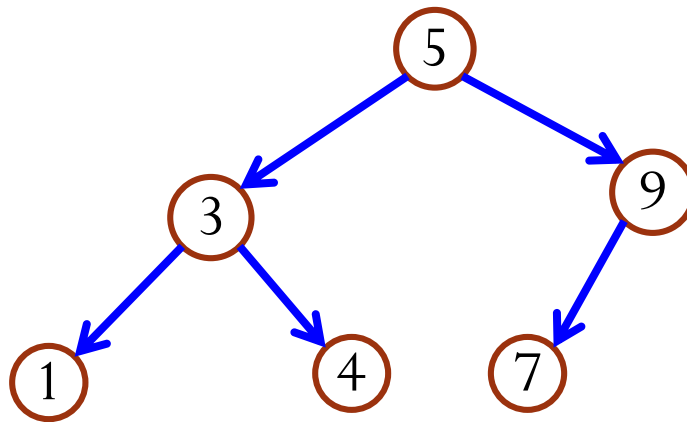
- Given **a node** in the BST, get its predecessor/successor.
 - Predecessor**: the node with the **largest** key that is **smaller** than the current key.
 - Successor**: the node with the **smallest** key that is **larger** than the current key.
 - Predecessor/Successor** is in the sense of in-order depth-first traversal.



What's predecessor of key 5?

What's successor of key 5?

Get Predecessor of a Node



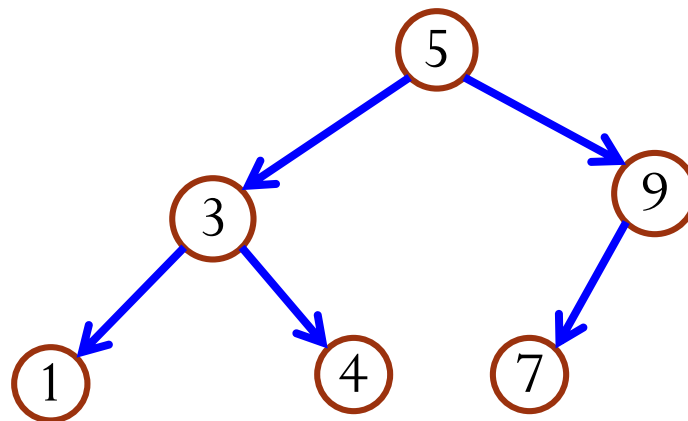
What's predecessor of key 5?

What's predecessor of key 7?

- **Easy case:** left subtree of the node is **nonempty** ...
 - ... return **max** key in left subtree.
- **Otherwise:** left subtree is **empty** ...
 - ... follow **parent pointers** until you get to a key less than the current key.
 - Equivalent: its first **left** ancestor.
- Time complexity? $O(\text{height})$. On average: $O(\log n)$.

Rank Search

- **Rank**: the index of the key in the **ascending order**.
 - We assume that the smallest key has rank 0.
- **Rank search**: get the key with rank k (i.e., the k -th smallest key).
 - Hash table does not support efficient rank search.
 - How to do rank search with a BST?
 - Simple solution: keep counting during an in-order depth-first traversal.



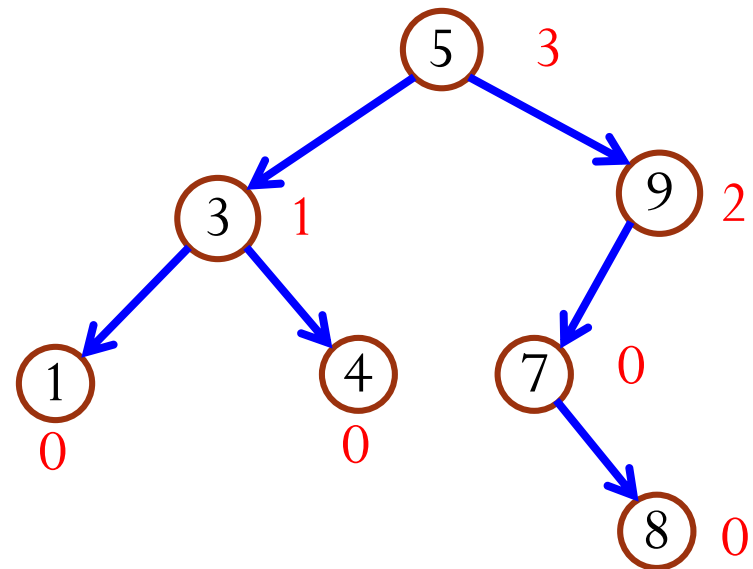
What's the average-case time complexity?

Can we do better?

BST with **leftSize**

- Each node has an additional field **leftSize**, indicating the number of nodes in its left subtree.

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





Which Statements Are Correct?

- Suppose we modify the basic BST to implement a BST with leftsize. Select all the correct statements.
- A.** The search method should be updated.
- B.** The insertion method should be updated, but not for the removal method.
- C.** The removal method should be updated, but not for the insertion method.
- D.** Both the insertion and removal methods should be updated.

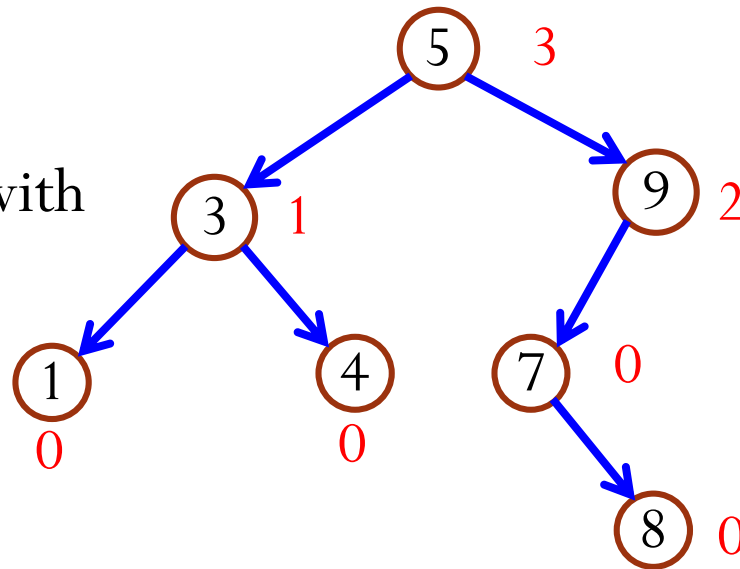


Rank Search

- Can we increase the efficiency of rank search with a BST with **leftSize**?

- What is the node with

- rank = 3?
- rank = 2?
- rank = 5?



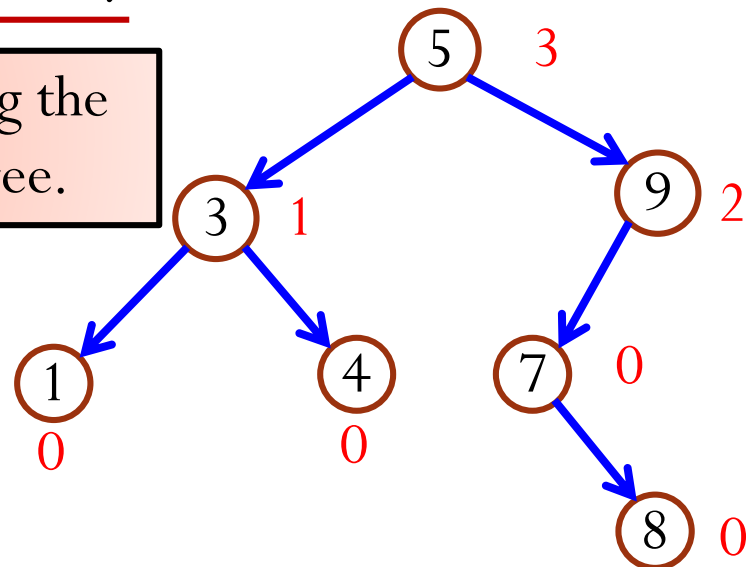
- Observation: **x .leftSize** = the rank of **x** in the **tree rooted at x** .
 - The rank of node 9 is 2 in the tree rooted at node 9.

Rank Search

```
node *rankSearch(node *root, int rank) {  
    if(root == NULL) return NULL;  
    if(rank == root->leftSize) return root;  
    if(rank < root->leftSize)  
        return rankSearch(root->left, rank);  
    else  
        return rankSearch(root->right,  
            rank - 1 - root->leftSize);  
}
```

The number of nodes including the current **root** and its left subtree.

What will **rankSearch(root, 5)** return?

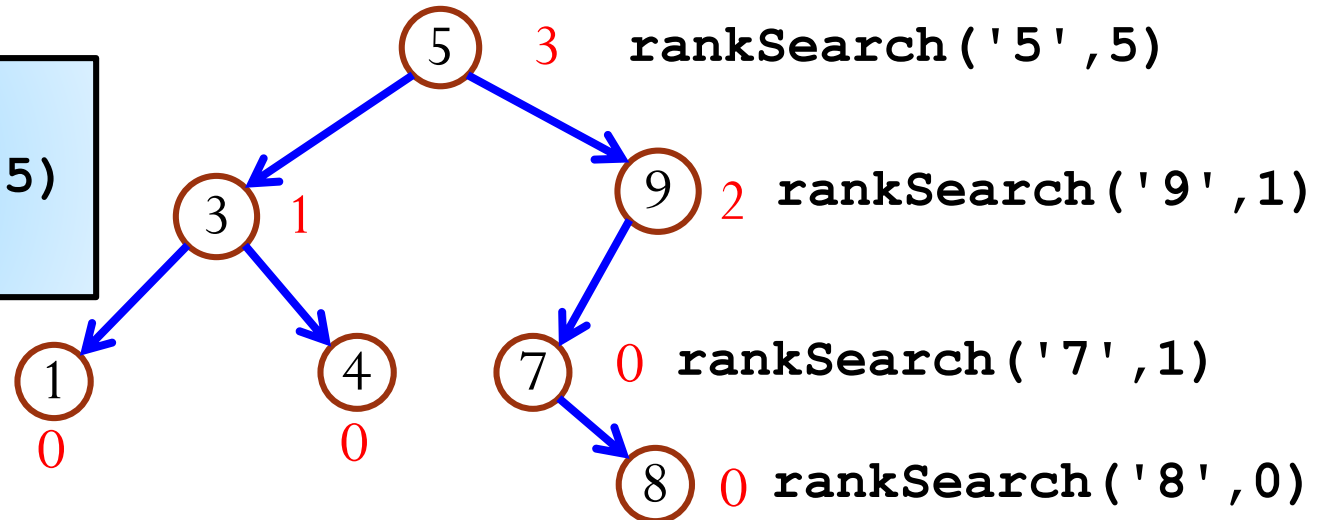


Rank Search

Example

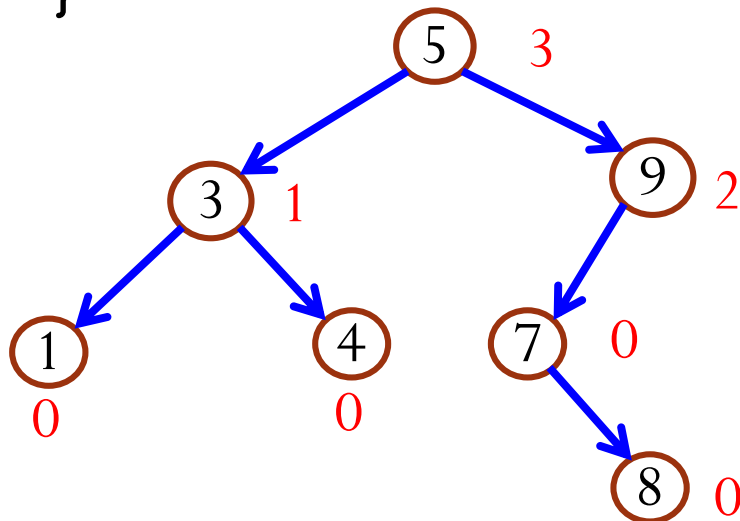
```
node *rankSearch(node *root, int rank) {  
    if(root == NULL) return NULL;  
    if(rank == root->leftSize) return root;  
    if(rank < root->leftSize)  
        return rankSearch(root->left, rank);  
    else  
        return rankSearch(root->right,  
            rank - 1 - root->leftSize);  
}
```

What will
rankSearch(root, 5)
return?



Rank Search

```
node *rankSearch(node *root, int rank) {  
    if(root == NULL) return NULL;  
    if(rank == root->leftSize) return root;  
    if(rank < root->leftSize)  
        return rankSearch(root->left, rank);  
    else  
        return rankSearch(root->right,  
            rank - 1 - root->leftSize);  
}
```

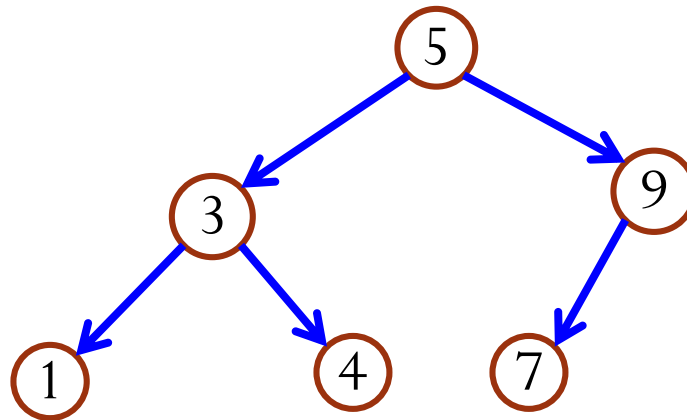


Time complexity?

$O(\text{height})$. On average: $O(\log n)$.

Range Search

- Instead of finding an exact match, find all items whose keys fall **between a range of values, inclusive**, in **sorted order**
 - E.g., between 4 and 8, inclusive.



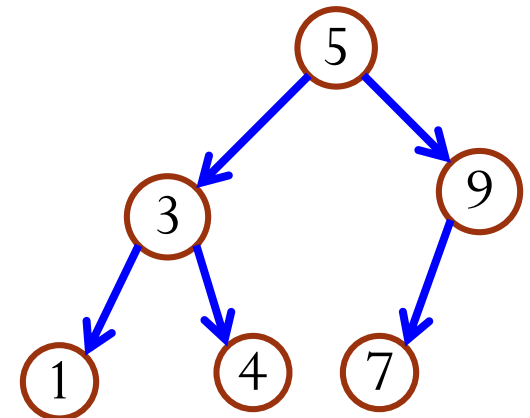
How could you implement range search?

- Example applications:
 - Buy ticket for travel between certain dates.

Range Search

Algorithm

1. Compute range of left subtree.
 - If search range covers all or part of left subtree, search left. (**recursive call**)
2. If root is in search range, add root to results.
3. Compute range of right subtree.
 - If search range covers all or part of right subtree, search right. (**recursive call**)
4. Return results.



```
void rangeSearch(node *root, Key searchRange[],  
    Key treeRange[], List results)
```


Range Search

Example

`rangeSearch('5', [4,8], $(-\infty, +\infty)$, results)`

searchRange **treeRange**

Does $(-\infty, 5)$ overlap $[4, 8]$? **Yes**

Does $(-\infty, 3)$ overlap $[4, 8]$? **No**

Is 3 in $[4, 8]$? **No**

Does $(3, 5)$ overlap $[4, 8]$? **Yes**

Is 4 in $[4, 8]$? **results \leftarrow 4**

Is 5 in $[4, 8]$? **results \leftarrow 5**

Does $(5, +\infty)$ overlap $[4, 8]$? **Yes**

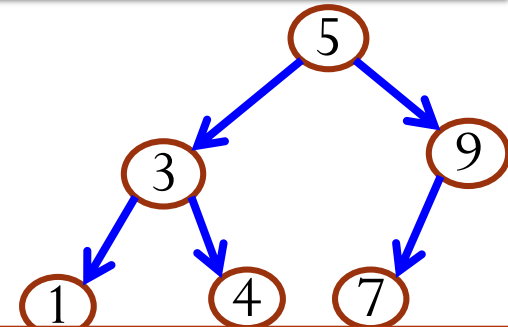
Does $(5, 9)$ overlap $[4, 8]$? **Yes**

Is 7 in $[4, 8]$? **results \leftarrow 7**

Is 9 in $[4, 8]$? **No**

Does $(9, +\infty)$ overlap $[4, 8]$? **No**

Call `rangeSearch('3', [4,8], $(-\infty, 5)$, results)`



Call `rangeSearch('9', [4,8], $(5, +\infty)$, results)`

results:
4, 5, 7

Note: results
are in order

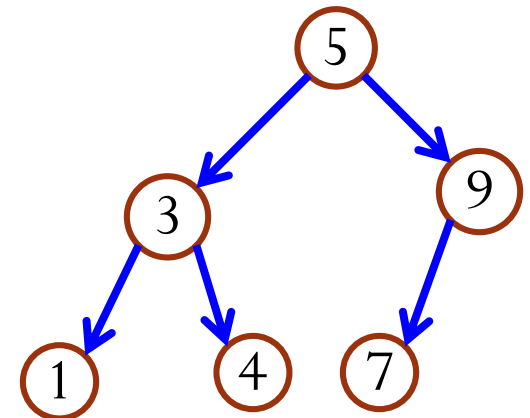
Range Search

Supporting Functions

- If node is in the search range, add node to the **results** list.
- Compute subtree's range:
 - Replace upper bound of left subtree by node's key
 - If possible, node's key "minus one".
 - Replace lower bound of right subtree by node's key
 - If possible, node's key "plus one".
- If search range covers all or part of subtree, search subtree.
 - Recursive calls

Range Search

1. Compute range of left subtree.
 - If search range covers all or part of left subtree, search left. (**recursive call**)
2. If root is in search range, add root to results.
3. Compute range of right subtree.
 - If search range covers all or part of right subtree, search right. (**recursive call**)
4. Return results.



Time complexity?

$O(n)$