# VE281

## Data Structures and Algorithms

Average-Case Time Complexity of BST;

BST Operations

# Announcement

- Written assignment 4 is released
  - On binary search tree
  - Due time: 3:40 pm, Nov. 3 (make-up lecture)
    - If you have difficulty in attending the lecture, please submit your homework to TA Yichen in advance

# Outline

- Analysis of BST Average-Case Time Complexity

- Efficient Binary Search Tree Operations

# 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 level $d$, the number of nodes visited is $d + 1$.
  - The complexity is $\Theta(d)$.

- 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 $\Theta(\bar{d})$
  - $\bar{d}$ 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

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

- $\boxed{I(n) \text{ is average of } T(n; l) \text{ over } l = 0,1,\dots,n-1.}$

# 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.
- Note: If first key inserted is $k_{l+1}$, the left subtree has $l$ nodes.
- Claim: All left subtree sizes are equally likely.
- Therefore, we have

$$I(n) = \frac{1}{n}\sum_{l=0}^{n-1} T(n;l)$$

$$= \frac{1}{n}\sum_{l=0}^{n-1} [I(l) + I(n-1-l) + n - 1]$$

$$= \frac{2}{n}\sum_{l=0}^{n-1} I(l) + (n-1)$$

# Solving the Recursion

$$I(n) = \frac{2}{n}\sum_{l=0}^{n-1} I(l) + (n-1)$$

replace $n$ with $n-1$

$$I(n-1) = \frac{2}{n-1}\sum_{l=0}^{n-2} I(l) + (n-2)$$

$$\sum_{l=0}^{n-2} I(l) = \frac{(n-1)[I(n-1)-(n-2)]}{2}$$

# Solving the Recursion

$$I(n) = \frac{2}{n}\sum_{l=0}^{n-1} I(l) + (n-1) \qquad \sum_{l=0}^{n-2} I(l) = \frac{(n-1)[I(n-1)-(n-2)]}{2}$$

$$I(n) = \frac{n+1}{n} I(n-1) + \frac{2(n-1)}{n}$$

$$\frac{I(n)}{n+1} = \frac{I(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \quad \leq \frac{I(n-1)}{n} + \frac{2}{n}$$

# Solving the Recursion

$$\frac{I(n)}{n+1} \leq \frac{I(n-1)}{n} + \frac{2}{n}$$

$$\frac{I(n)}{n+1} \leq \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \cdots + \frac{2}{2} + \frac{I(1)}{2}$$

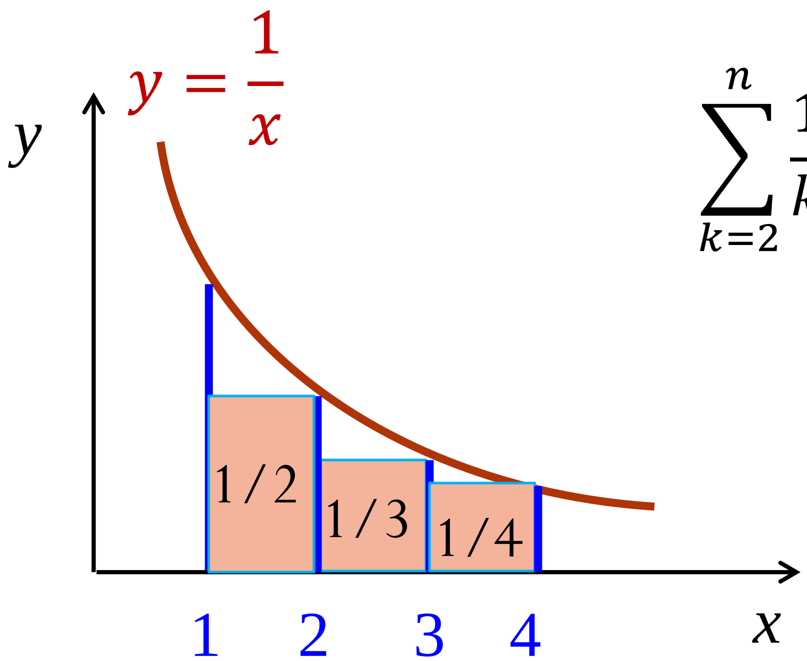
$I(1) = 0$

$$\frac{I(n)}{n+1} \leq 2\sum_{k=2}^{n} \frac{1}{k}$$

Note: $\displaystyle\sum_{k=2}^{n} \frac{1}{k} < \ln n$

# Proof of the Claim

- Claim: $\displaystyle\sum_{k=2}^{n}\frac{1}{k} < \ln n$




$$\sum_{k=2}^{n}\frac{1}{k} < \int_{1}^{n}\frac{1}{x}\,dx = \ln n$$

# Average Case Analysis Conclusion

- What we get so far:

$$\frac{I(n)}{n+1} \leq 2\sum_{k=2}^{n}\frac{1}{k} < 2\ln n$$

- Thus, we have

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

- Thus, the average complexity for a successful search is

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

# Average Case Time Complexity

- It can also be shown that given $n$ nodes, the average-case time complexity for an **unsuccessful search** is $O(\log n)$.
- Given $n$ nodes, the average-case time complexities for search, insertion, and removal are all $O(\log n)$.
  - Insertion and removal include "search".

| | 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?

| | Average-Case Time Complexity |
|---|---|
| **Other Operations Supported by BST** | |
| 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 | $O(n)$ |

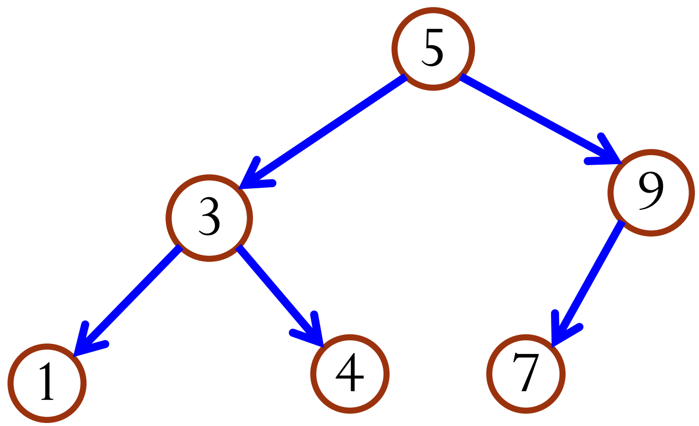
<u>Note</u>: Hash table does not support efficient implementation of the above methods.

# Outline

- Analysis of BST Average-Case Time Complexity

- Efficient Binary Search Tree Operations

# Output in Sorted Order

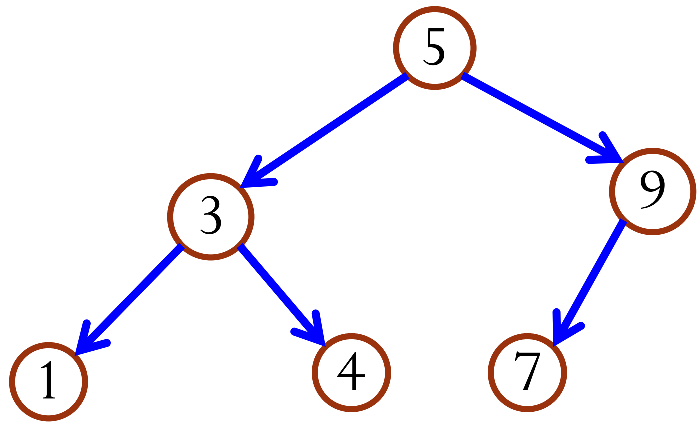



- Output: 1, 3, 4, 5, 7, 9
- **How?**
  - In-order depth-first traversal.

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

- Time complexity: $O(n)$.

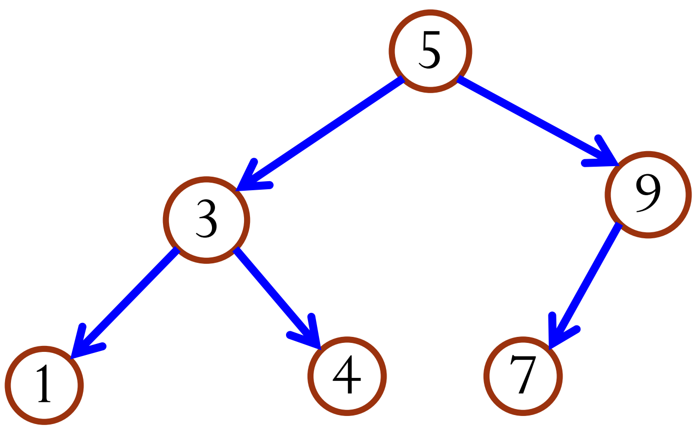
# Get Min/Max




- 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

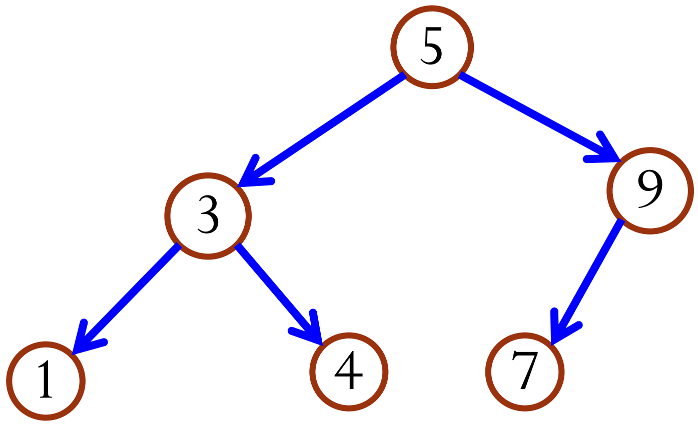
- Given **a node** in the BST, get its predecessor/successor.
  - **Predecessor**: the node with the **largest** key that is **smaller** than the current key.
  - **Successor**: the node with the **smallest** key that is **larger** than the current key.
  - **Predecessor**/**Successor** is in the sense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




What's predecessor of key 5?

What's successor of key 5?

# Get Predecessor of a Node



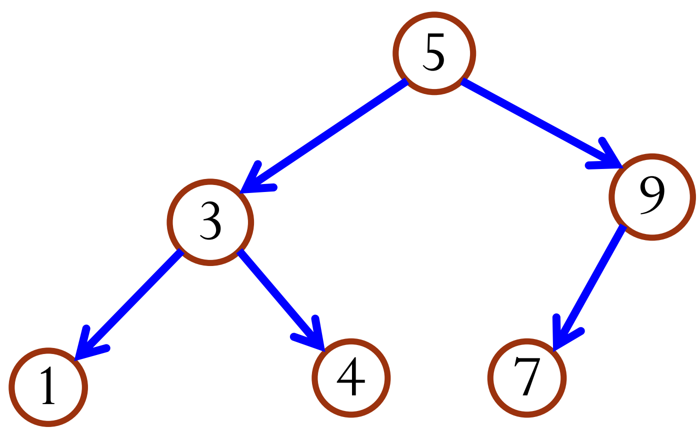

What's predecessor of key 5?

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 **emtpy** …
  - … follow **parent pointers** until you get to a key less than the current key.
  - Equivalent: its first <u>**left**</u> 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?

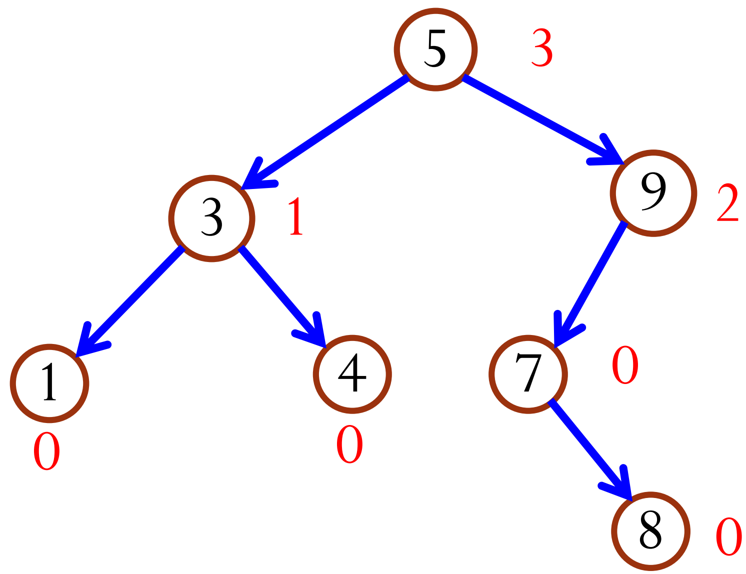
Can we do better?

# BST with `leftSize`

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

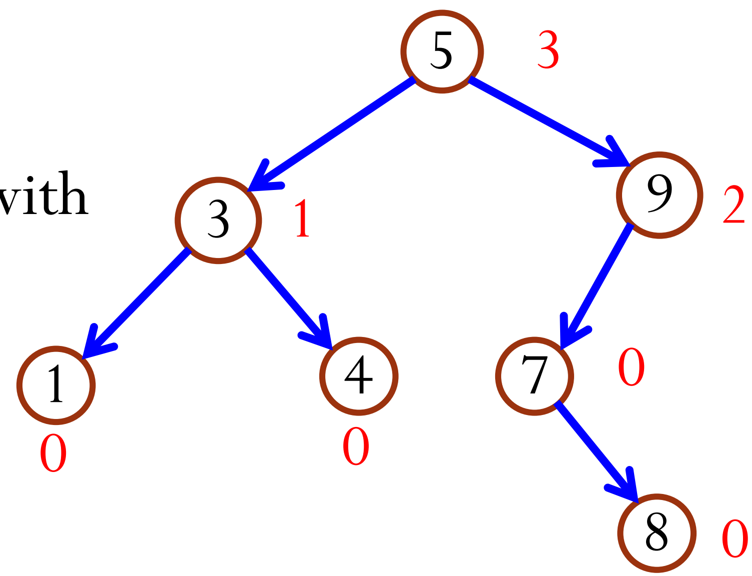
```
struct node {
   Item item;
   int leftSize;
   node *left;
   node *right;
};
```

Should change insertion and removal methods.



# Rank Search

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



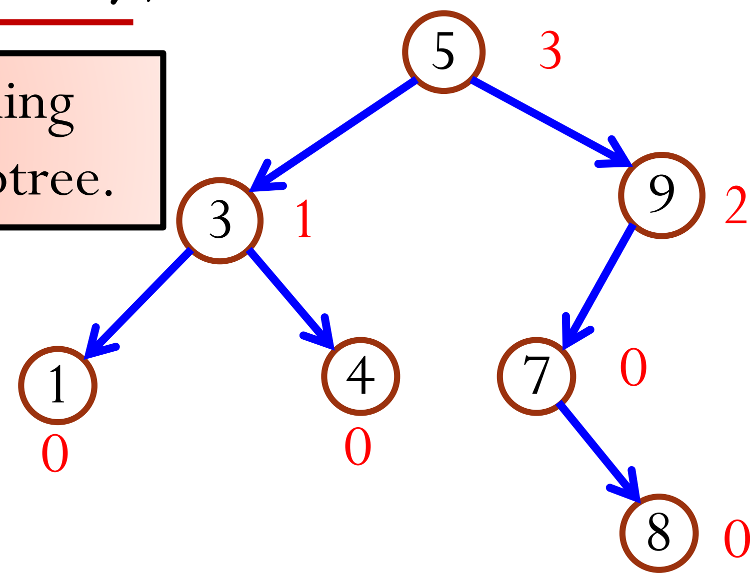

- What is the node with
  - rank = 3?
  - rank = 2?
  - rank = 5?
- Observation: **`x.leftSize`** = the rank of **`x`** in the **tree rooted** at **`x`**.
  - The rank of node 9 is 2 in the tree rooted at node 9.

# 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 subtree.

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

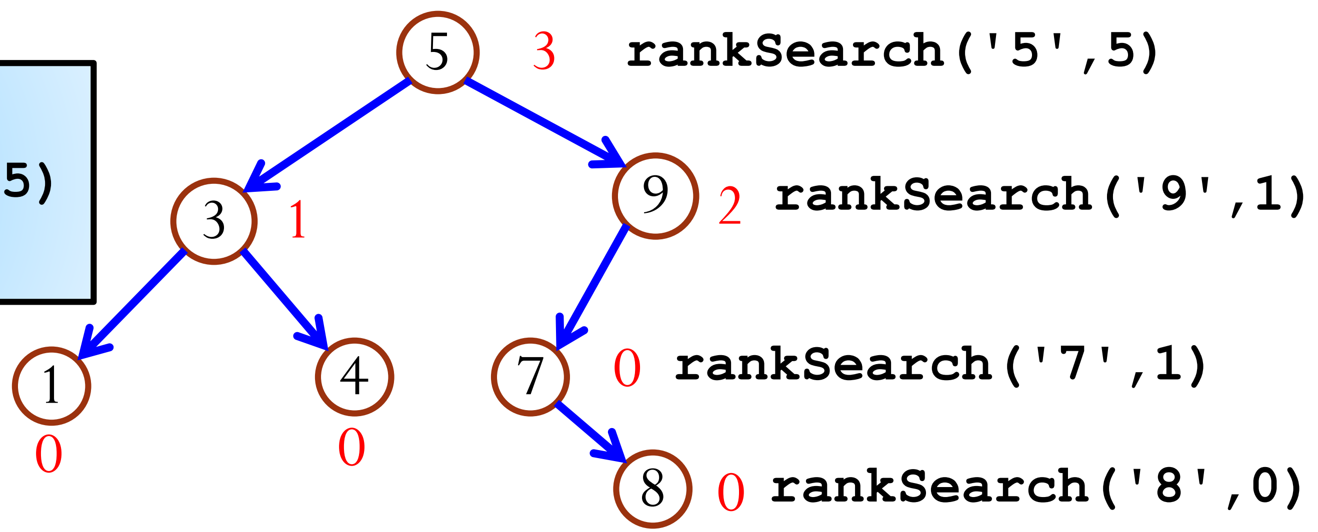


# Rank Search

## Example

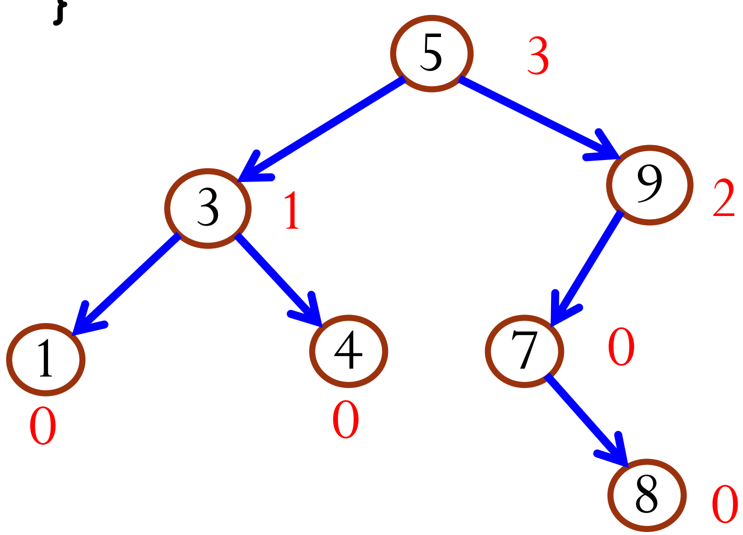
```
node *rankSearch(node *root, int rank) {
  if(root == NULL) return NULL;
  if(rank == root->leftSize) return root;
  if(rank < root->leftSize)
    return rankSearch(root->left, rank);
  else
    return rankSearch(root->right,
      rank - 1 - root->leftSize);
}
```

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);
}
```

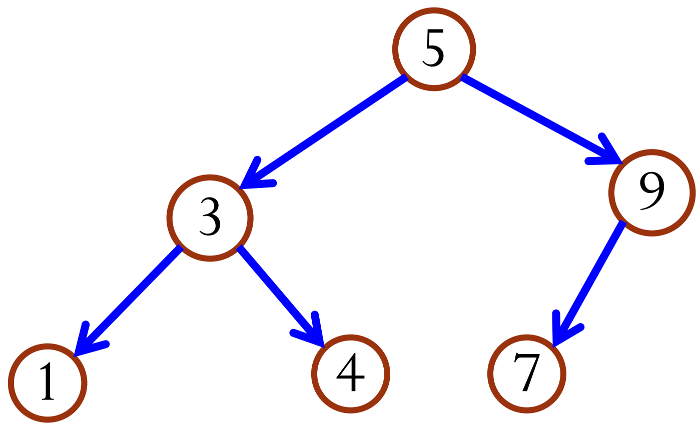



**Time complexity?**

$O$(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.

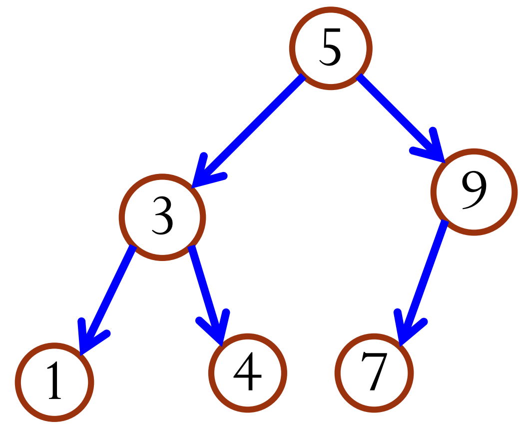



How could you implement range search?

- 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 node is in search range add node 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], (-∞,+∞), results)
searchRange treeRange

Does (-∞,5) overlap [4,8]? **Yes** — Call rangeSearch('3', [4,8],(-∞,5),results)

Does (-∞,3) overlap [4,8]? **No**

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

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

Is 4 in [4,8]? results ← 4

Is 5 in [4,8]? results ← 5

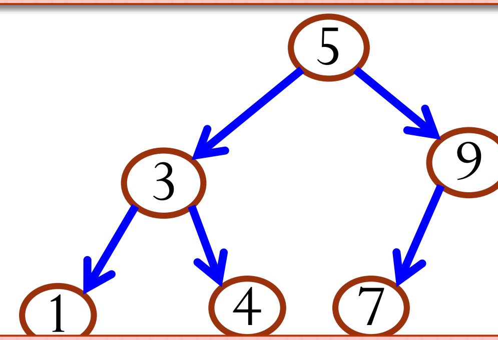
Does (5,+∞) overlap [4,8]? **Yes** — Call rangeSearch('9', [4,8],(5,+∞),results)

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

Is 7 in [4,8]? results ← 7

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

Does (9,+∞) overlap [4,8]? **No**




results:
4,5,7

Note: results are in order

# Range Search
## Supporting Functions

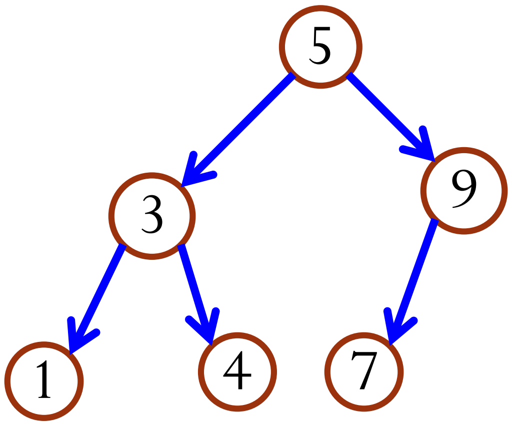
- If node is in the search range, add node to the **`results`** list.

- 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 node is in search range add node 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?**

$O(n)$