

ACOL106: Data Structures and Algorithms

Ragesh Jaiswal, IIT Delhi

Data Structures: Binary Search Trees

Data Structures

Binary Search Trees

- Suppose we want to store n data entries where each data entry consists of a key (this can be thought of as a unique **integer** ID) and a value.
- We would like to perform the following operations:
 - get(k): Search an entry with key k and return the value.
 - put(k, v): Associate value v with key k , replacing and returning any existing value in case an entry with key k exists or inserting a new entry if no entry with key k exists.
 - remove(k): Delete an entry with key k .

Data Structures

Binary Search Trees

- Suppose we want to store n data entries where each data entry consists of a key (this can be thought of as a unique **integer** ID) and a value.
- We would like to perform the following operations:
 - get(k): Search an entry with key k and return the value.
 - put(k, v): Associate value v with key k , replacing and returning any existing value in case an entry with key k exists or inserting a new entry if no entry with key k exists.
 - remove(k): Delete an entry with key k .
- Consider an array-based implementation where the elements are NOT sorted based on the keys. What is the running time of:
 - get(k):
 - put(k, v):
 - remove(k):

Data Structures

Binary Search Trees

- Suppose we want to store n data entries where each data entry consists of a key (this can be thought of as a unique **integer** ID) and a value.
- We would like to perform the following operations:
 - get(k): Search an entry with key k and return the value.
 - put(k, v): Associate value v with key k , replacing and returning any existing value in case an entry with key k exists or inserting a new entry if no entry with key k exists.
 - remove(k): Delete an entry with key k .
- Consider an array-based implementation where the elements are NOT sorted based on the keys. What is the running time of:
 - get(k): $O(n)$
 - put(k, v): $O(n)$
 - remove(k): $O(n)$

Data Structures

Binary Search Trees

- Suppose we want to store n data entries where each data entry consists of a key (this can be thought of as a unique **integer** ID) and a value.
- We would like to perform the following operations:
 - get(k): Search an entry with key k and return the value.
 - put(k, v): Associate value v with key k , replacing and returning any existing value in case an entry with key k exists or inserting a new entry if no entry with key k exists.
 - remove(k): Delete an entry with key k .
- Consider an array-based implementation where the elements are sorted based on the keys. What is the running time of:
 - get(k):
 - put(k, v):
 - remove(k):

Data Structures

Binary Search Trees

- Suppose we want to store n data entries where each data entry consists of a key (this can be thought of as a unique **integer** ID) and a value.
- We would like to perform the following operations:
 - get(k): Search an entry with key k and return the value.
 - put(k, v): Associate value v with key k , replacing and returning any existing value in case an entry with key k exists or inserting a new entry if no entry with key k exists.
 - remove(k): Delete an entry with key k .
- Consider an array-based implementation where the elements are sorted based on the keys. What is the running time of:
 - get(k): $O(\log n)$
 - put(k, v): $O(n)$
 - remove(k): $O(n)$

Data Structures

Binary Search Trees

- Suppose we want to store n data entries where each data entry consists of a key (this can be thought of as a unique **integer** ID) and a value.
- We would like to perform the following operations:
 - get(k): Search an entry with key k and return the value.
 - put(k, v): Associate value v with key k , replacing and returning any existing value in case an entry with key k exists or inserting a new entry if no entry with key k exists.
 - remove(k): Delete an entry with key k .
- Our next goal is to build a data structure where the running time of the operations are:
 - get(k): $O(\log n)$
 - put(k, v): $O(\log n)$
 - remove(k): $O(\log n)$

Data Structures

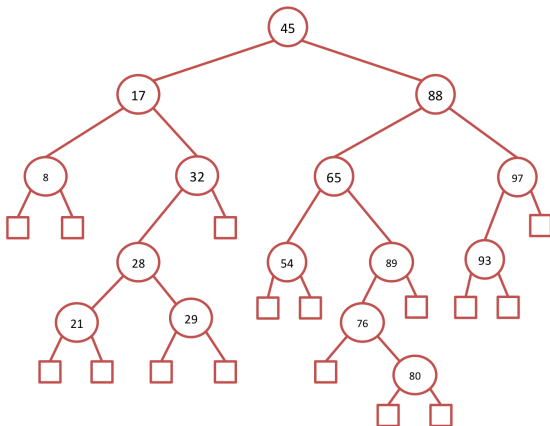
Binary Search Trees

- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - 1 Keys stored in the left sub-tree of p are less than k
 - 2 Keys stored in the right sub-tree of p are greater than k

Data Structures

Binary Search Trees

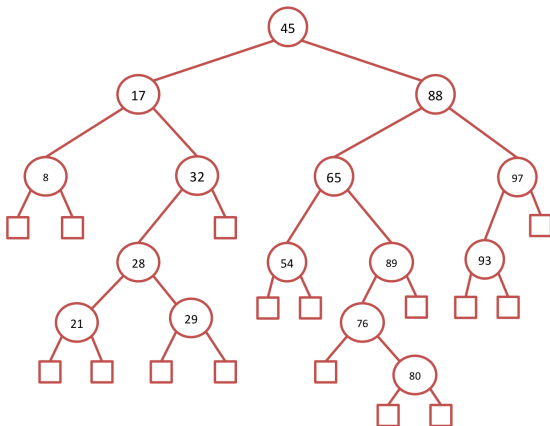
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Is the tree below a binary search tree?



Data Structures

Binary Search Trees

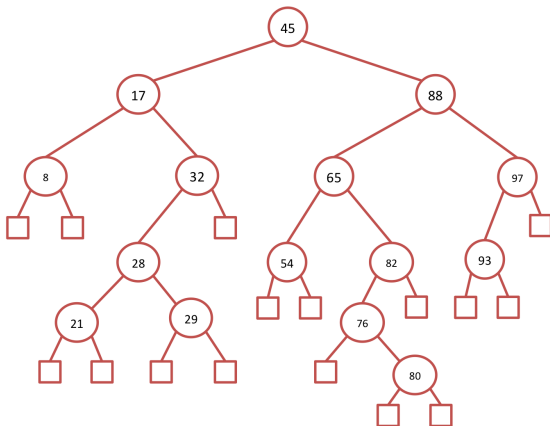
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Is the tree below a binary search tree? **No**



Data Structures

Binary Search Trees

- **Binary Search Tree:** Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - 1 Keys stored in the left sub-tree of p are less than k
 - 2 Keys stored in the right sub-tree of p are greater than k
- Is the tree below a binary search tree?



Data Structures

Binary Search Trees

- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - 1 Keys stored in the left sub-tree of p are less than k
 - 2 Keys stored in the right sub-tree of p are greater than k
- Is the tree below a binary search tree? **Yes**

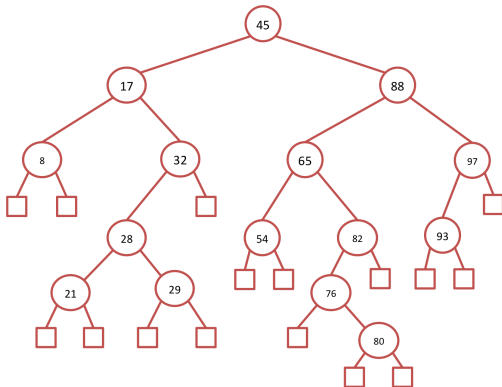
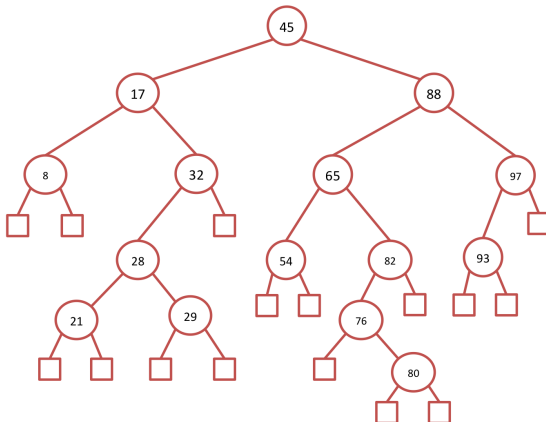


Figure: The “value” of entries is not shown.

Data Structures

Binary Search Trees

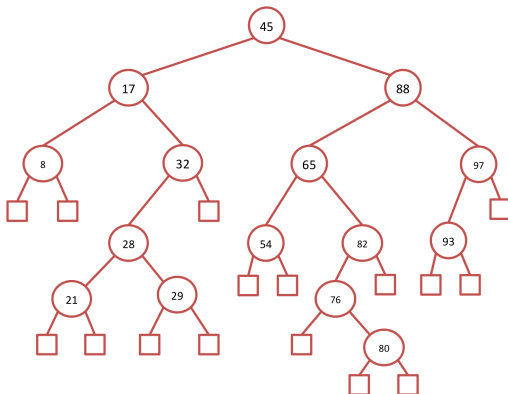
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - 1 Keys stored in the left sub-tree of p are less than k
 - 2 Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{get}(k)$?



Data Structures

Binary Search Trees

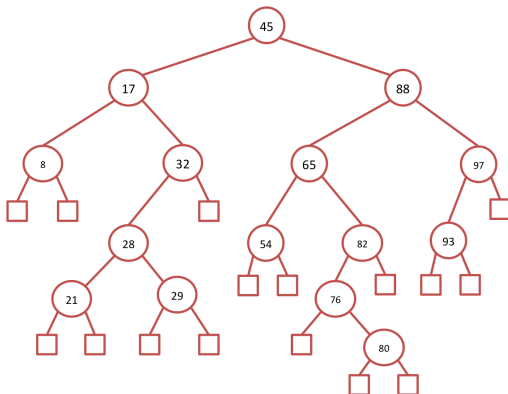
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{get}(k)$?
 - How do we search for the key 68 in the binary search tree below?



Data Structures

Binary Search Trees

- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{get}(k)$?
 - Now try searching 76 in the tree.



Data Structures

Binary Search Trees

- Consider the following implementation:

Code

```
class BinarySearchTree:
    """ The Binary Search Tree class"""
    #-----Nested class _Node -----#
    class _Node:
        def __init__(self, key, value, parent, left_child, right_child):
            self._key = key
            self._value = value
            self._parent = parent
            self._left_child = left_child
            self._right_child = right_child

    #-----#
    def __init__(self):
        self._root = None
        self._size = 0

    def __len__(self):
        return self._size

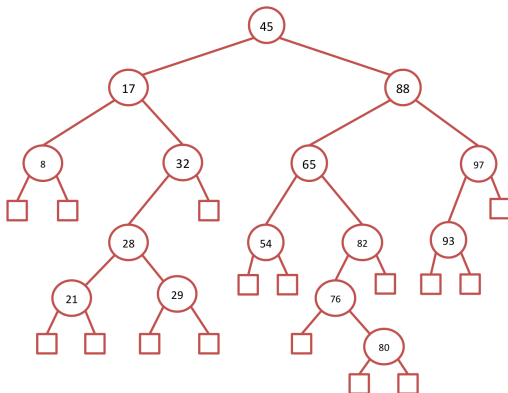
    def isLeaf(self, node):

    def get(self, key):
```

Data Structures

Binary Search Trees

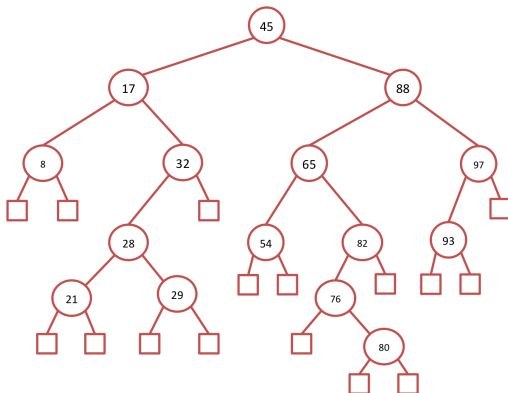
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{put}(k, v)$?
 - Suppose we want to perform $\text{put}(68, \text{"A"})$ in the BST below.



Data Structures

Binary Search Trees

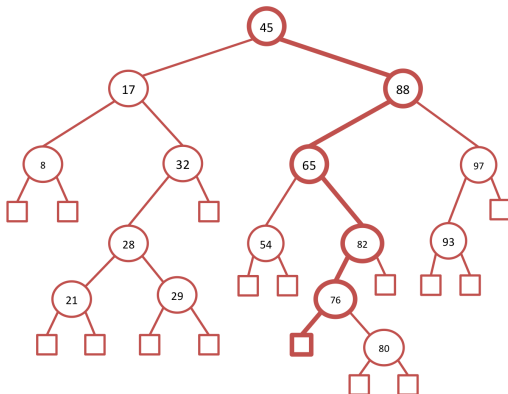
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{put}(k, v)$?
 - Suppose we want to perform $\text{put}(68, \text{"A"})$ in the BST below.



Data Structures

Binary Search Trees

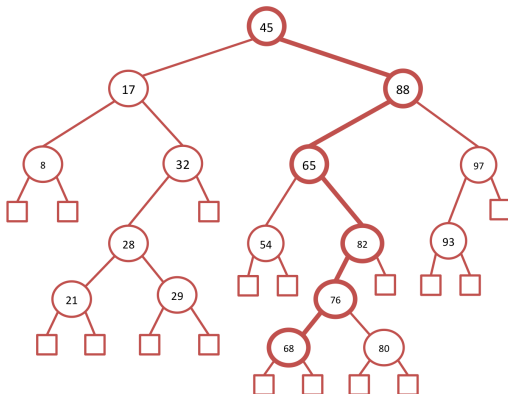
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{put}(k, v)$?
 - Suppose we want to perform $\text{put}(68, \text{"A"})$ in the BST below.



Data Structures

Binary Search Trees

- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{put}(k, v)$?
 - Suppose we want to perform $\text{put}(68, \text{"A"})$ in the BST below.



Data Structures

Binary Search Trees

- Consider the following implementation:

Code

```
class BinarySearchTree:
    """ The Binary Search Tree class"""
    #-----Nested class _Node -----#
    class _Node:
        def __init__(self, key, value, parent, left_child, right_child):
            self._key = key
            self._value = value
            self._parent = parent
            self._left_child = left_child
            self._right_child = right_child

    #-----#
    def __init__(self):
        self._root = None
        self._size = 0

    def __len__(self):
        return self._size

    def isLeaf(self, node):

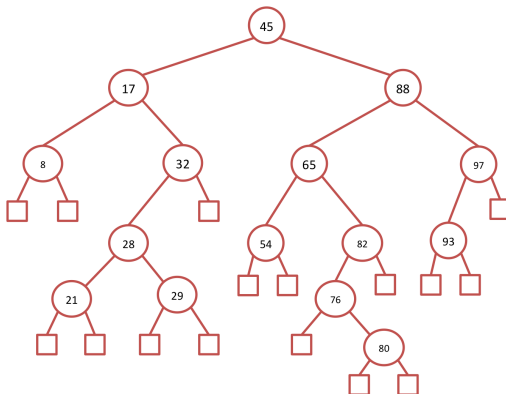
    def get(self, key):

    def put(self, key, value):
```

Data Structures

Binary Search Trees

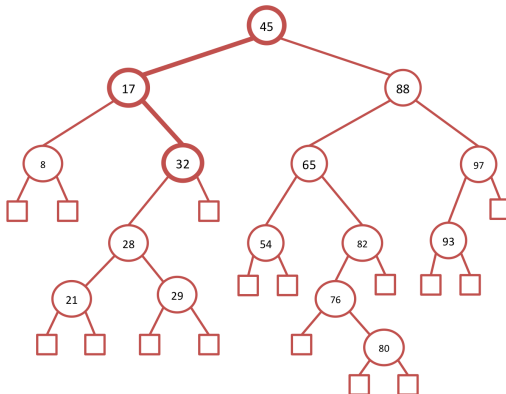
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{remove}(k)$?
 - Suppose we want to perform $\text{remove}(32)$ in the BST below.



Data Structures

Binary Search Trees

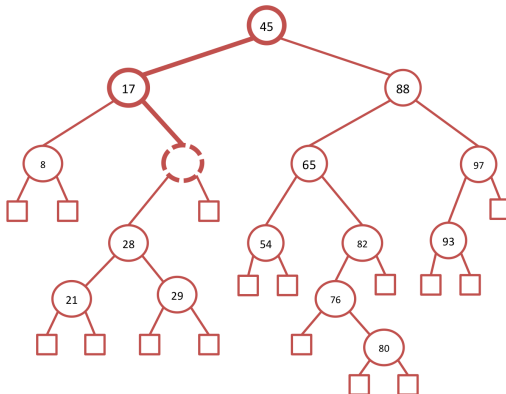
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{remove}(k)$?
 - Suppose we want to perform $\text{remove}(32)$ in the BST below.



Data Structures

Binary Search Trees

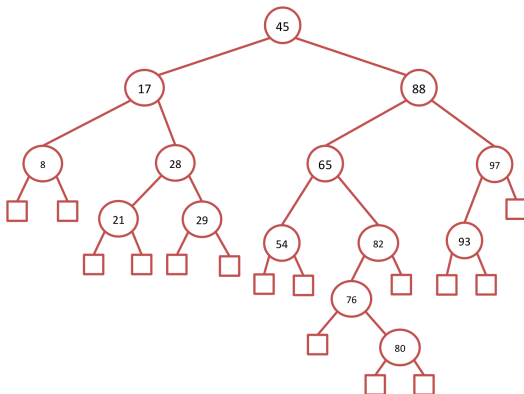
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{remove}(k)$?
 - Suppose we want to perform $\text{remove}(32)$ in the BST below.



Data Structures

Binary Search Trees

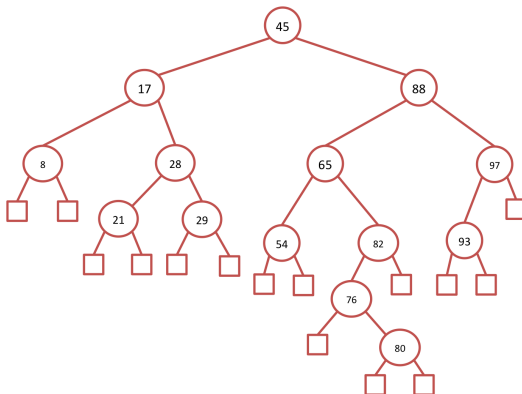
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{remove}(k)$?
 - Suppose we want to perform $\text{remove}(32)$ in the BST below.



Data Structures

Binary Search Trees

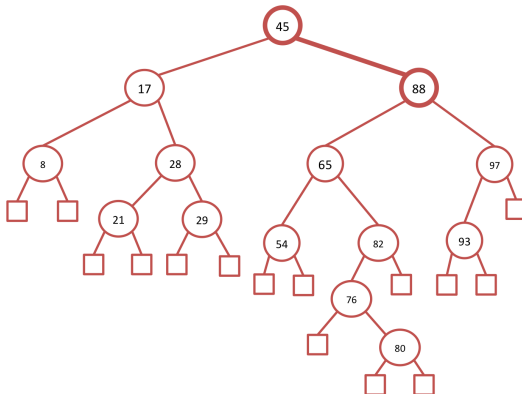
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{remove}(k)$?
 - Suppose we want to perform $\text{remove}(88)$ in the BST below.



Data Structures

Binary Search Trees

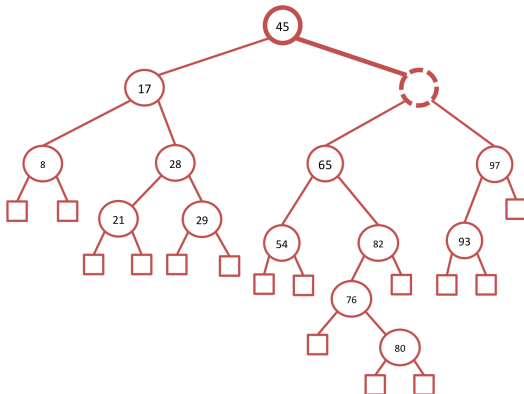
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{remove}(k)$?
 - Suppose we want to perform $\text{remove}(88)$ in the BST below.



Data Structures

Binary Search Trees

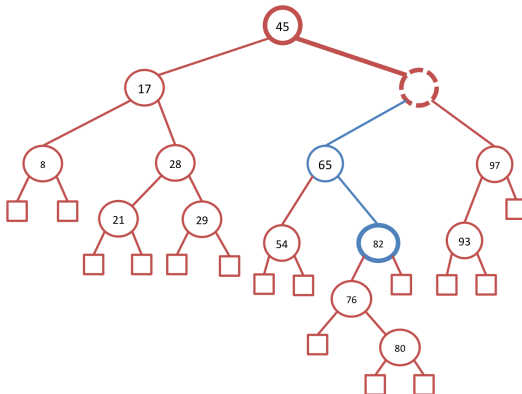
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{remove}(k)$?
 - Suppose we want to perform $\text{remove}(88)$ in the BST below.



Data Structures

Binary Search Trees

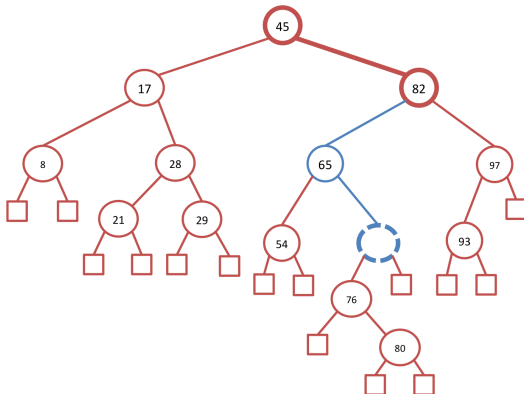
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{remove}(k)$?
 - Suppose we want to perform $\text{remove}(88)$ in the BST below.



Data Structures

Binary Search Trees

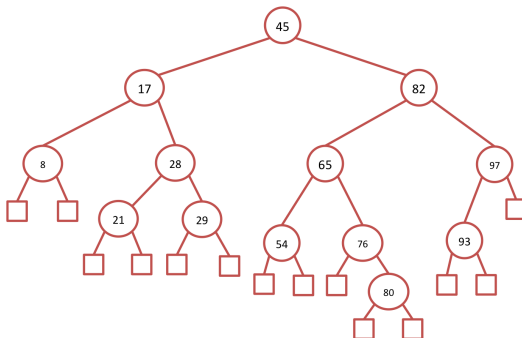
- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{remove}(k)$?
 - Suppose we want to perform $\text{remove}(88)$ in the BST below.



Data Structures

Binary Search Trees

- Binary Search Tree: Binary Search Trees are **proper binary trees** such that each **internal** node p stores a key-value pair such that:
 - ① Keys stored in the left sub-tree of p are less than k
 - ② Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do we perform $\text{remove}(k)$?
 - Suppose we want to perform $\text{remove}(88)$ in the BST below.



Data Structures

Binary Search Trees

- Consider the following implementation:

Code

```
class BinarySearchTree:
    """ The Binary Search Tree class"""
    #-----Nested class _Node -----#
    class _Node:
        def __init__(self, key, value, parent, left_child, right_child):
            self._key = key
            self._value = value
            self._parent = parent
            self._left_child = left_child
            self._right_child = right_child
    #-----#
    def __init__(self):
        self._root = None
        self._size = 0

    def __len__(self):
        return self._size

    def isLeaf(self, node):

    def get(self, key):

    def put(self, key, value):

    def remove(self, key):
```

Data Structures

Binary Search Trees

- What is the worst-case running time of each of the following operations?
 - `get(k)`:
 - `put(k, v)`:
 - `remove(k)`:

Data Structures

Binary Search Trees

- What is the worst-case running time of each of the following operations?
 - `get(k)`: $O(n)$
 - `put(k, v)`: $O(n)$
 - `remove(k)`: $O(n)$

Data Structures

Binary Search Trees

- What is the worst-case running time of each of the following operations when the BST is **balanced**?
 - `get(k)`:
 - `put(k, v)`:
 - `remove(k)`:
- A BST is perfectly balanced if for every internal node, there is an equal number of nodes in its left and right sub-trees.

Data Structures

Binary Search Trees

- What is the worst-case running time of each of the following operations when the BST is **balanced**?
 - `get(k)`: $O(\log n)$
 - `put(k, v)`: $O(\log n)$
 - `remove(k)`: $O(\log n)$
- So, our next goal shall be to build **balanced** binary search trees.

Data Structures: **Balanced** Binary Search Trees

Data Structures

Binary Search Trees

- Consider the following implementation:

Code

```
class BinarySearchTree:
    """ The Binary Search Tree class"""
    #-----Nested class _Node -----#
    class _Node:
        def __init__(self, key, value, parent, left_child, right_child):
            self._key = key
            self._value = value
            self._parent = parent
            self._left_child = left_child
            self._right_child = right_child
    #-----#
    def __init__(self):
        self._root = None
        self._size = 0

    def __len__(self):
        return self._size

    def isLeaf(self, node):

    def get(self, key):

    def put(self, key, value):

    def remove(self, key):
```

Data Structures

Binary Search Trees

- What is the worst-case running time of each of the following operations?
 - `get(k)`:
 - `put(k, v)`:
 - `remove(k)`:

Data Structures

Binary Search Trees

- What is the worst-case running time of each of the following operations?
 - `get(k)`: $O(n)$
 - `put(k, v)`: $O(n)$
 - `remove(k)`: $O(n)$

Data Structures

Binary Search Trees

- What is the worst-case running time of each of the following operations when the BST is **balanced**?
 - `get(k)`:
 - `put(k, v)`:
 - `remove(k)`:
- A BST is perfectly balanced if for every internal node, there is an equal number of nodes in its left and right sub-trees.

Data Structures

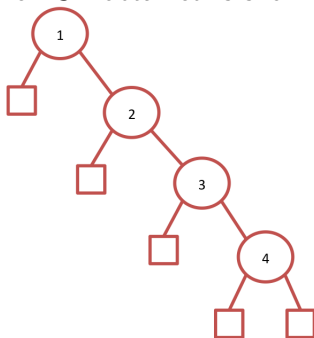
Binary Search Trees

- What is the worst-case running time of each of the following operations when the BST is **balanced**?
 - `get(k)`: $O(\log n)$
 - `put(k, v)`: $O(\log n)$
 - `remove(k)`: $O(\log n)$
- So, our next goal shall be to build **balanced** binary search trees.

Data Structures

Balanced Binary Search Trees

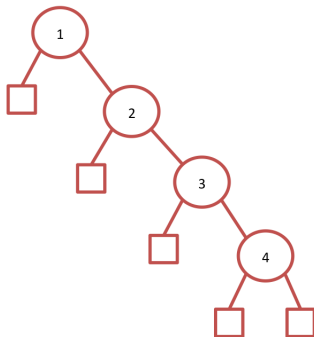
- Suppose we start with an empty BST and insert the keys 1, 2, 3, 4, then the BST obtained is shown below.



Data Structures

Balanced Binary Search Trees

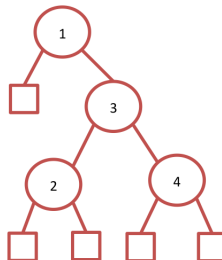
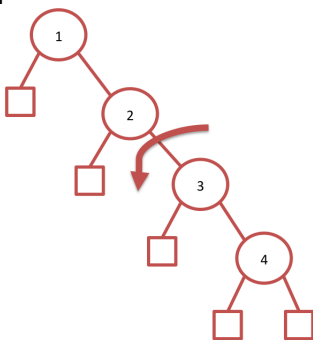
- Suppose we start with an empty BST and insert the keys 1, 2, 3, 4, then the BST obtained is shown below.
- This tree is not balanced. Can you think of a way to balance this tree?



Data Structures

Balanced Binary Search Trees

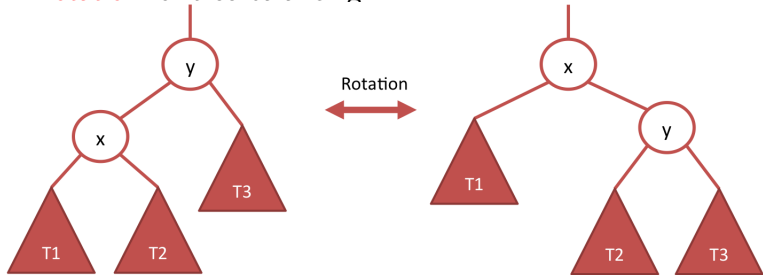
- Suppose we start with an empty BST and insert the keys 1, 2, 3, 4, then the BST obtained is shown below.
- This tree is not balanced. Can you think of a way to balance this tree?



Data Structures

Balanced Binary Search Trees

- **Rotation** for tree balancing.



Data Structures

Balanced Binary Search Trees

- **Tri-node restructuring** for a node x , its parent y , and its grandparent z .

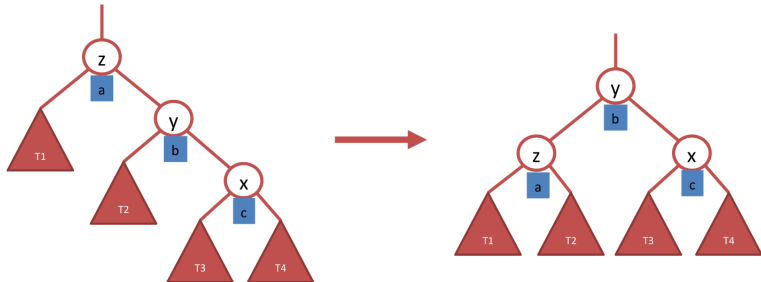


Figure: Case #1

Data Structures

Balanced Binary Search Trees

- **Tri-node restructuring** for a node x , its parent y , and its grandparent z .

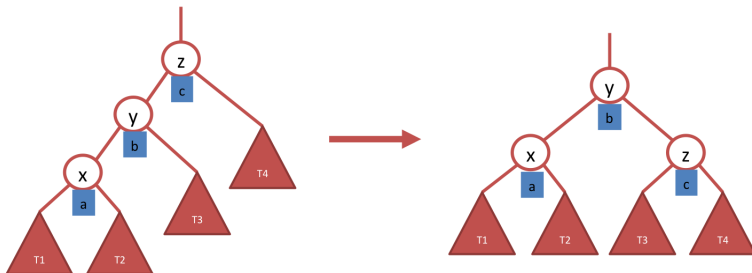


Figure: Case #2

Data Structures

Balanced Binary Search Trees

- **Tri-node restructuring** for a node x , its parent y , and its grandparent z .

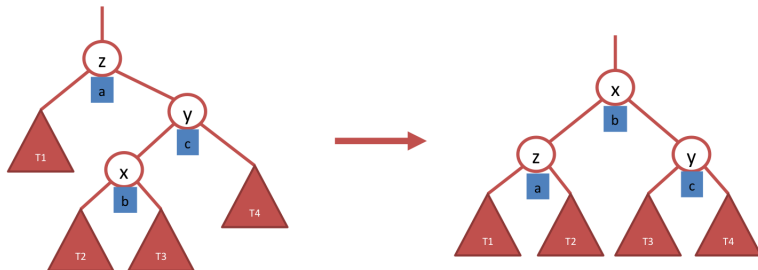


Figure: Case #3

Data Structures

Balanced Binary Search Trees

- **Tri-node restructuring** for a node x , its parent y , and its grandparent z .

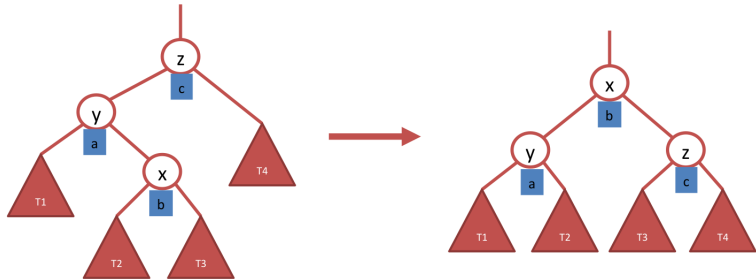


Figure: Case #4

Data Structures

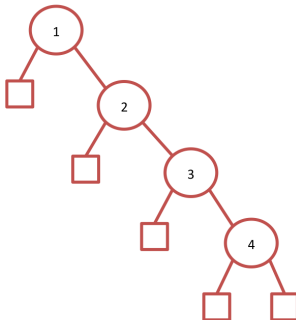
Balanced Binary Search Trees → AVL Trees

- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.

Data Structures

Balanced Binary Search Trees → AVL Trees

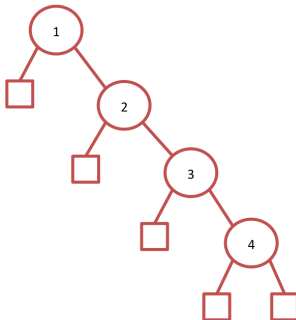
- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Is the binary search tree below an AVL tree?



Data Structures

Balanced Binary Search Trees → AVL Trees

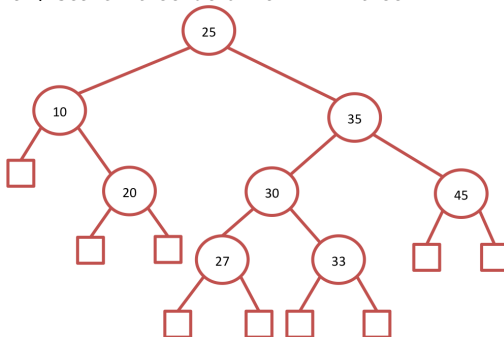
- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Is the binary search tree below an AVL tree? No



Data Structures

Balanced Binary Search Trees → AVL Trees

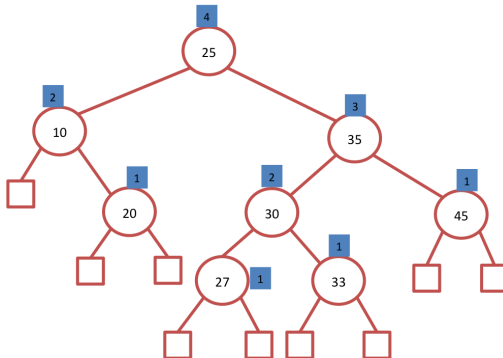
- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Is the binary search tree below an AVL tree?



Data Structures

Balanced Binary Search Trees → AVL Trees

- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Is the binary search tree below an AVL tree? **Yes**



Data Structures

Balanced Binary Search Trees → AVL Trees

- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Claim: The height of any AVL tree storing n nodes is $O(\log n)$.

End