COL106: Data Structures and Algorithms

Ragesh Jaiswal, IIT Delhi

Data Structures: Heaps and Priority Queues

- What is the running time of each of these operations in the array based implementation of Min-Heap?
  - `insert(k, v)`:
  - `min()`:
  - `removeMin()`:

- What is the running time of each of these operations in the array based implementation of Min-Heap?
  - `insert(k, v)`: $O(\log n)$
  - `min()`: $O(1)$
  - `removeMin()`: $O(\log n)$

Problem

Given n entries create a min-heap of these entries.

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations.
  - What is the running time?

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations.
    - What is the running time? $O(n \log n)$

Problem

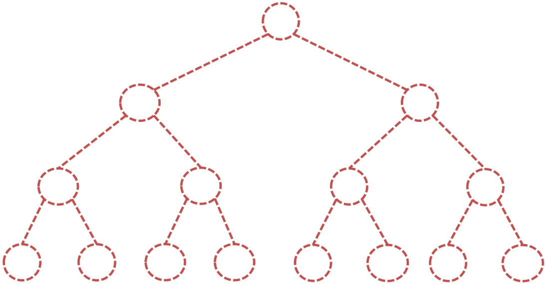
Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction
  - Question: Suppose we have a min-heap H_1 and H_2 both containing $2^h - 1$ entries and an entry E. Can you construct a min-heap for all entries in H_1, H_2 and E combined? What is the running time for your combination algorithm?

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction

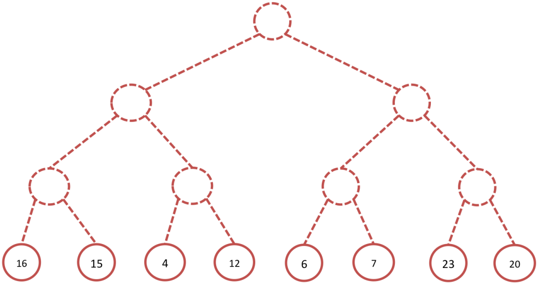


14	5	8	25	9	11	7	16	15	4	12	6	7	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction

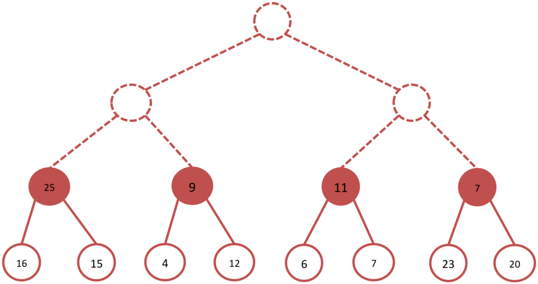


14	5	8	25	9	11	7	16	15	4	12	6	7	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction

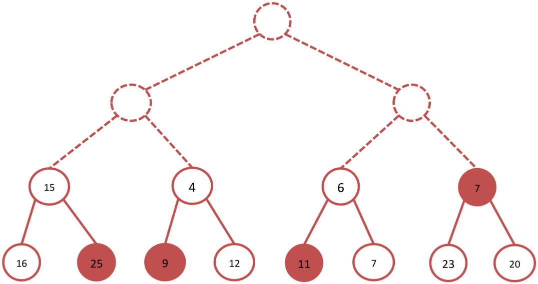


14	5	8	25	9	11	7	16	15	4	12	6	7	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction

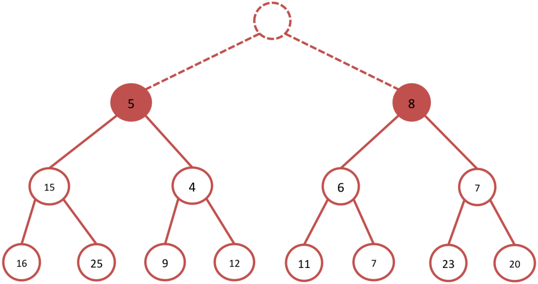


14	5	8	15	4	6	7	16	25	9	12	11	7	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction

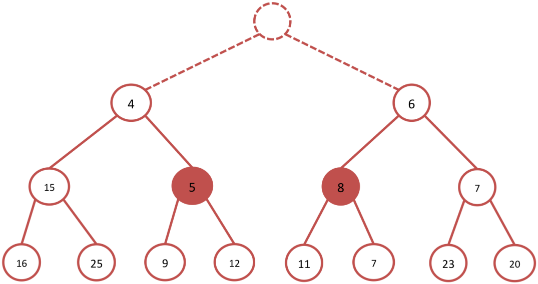


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
14	5	8	15	4	6	7	16	25	9	12	11	7	23	20

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction

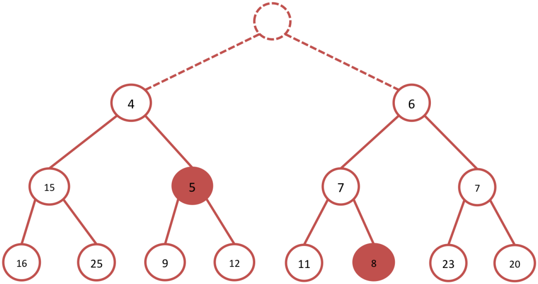


14	4	6	15	5	8	7	16	25	9	12	11	7	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction

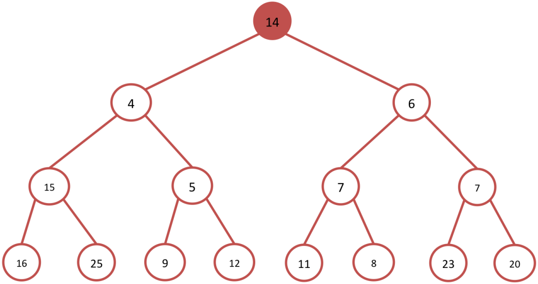


14	4	6	15	5	7	7	16	25	9	12	11	8	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction

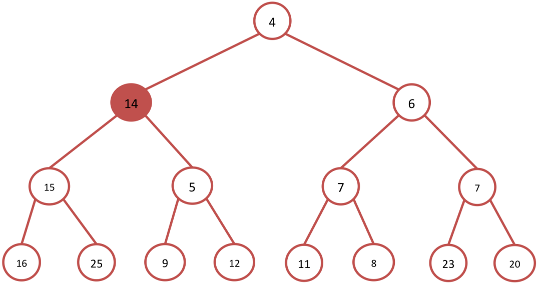


14	4	6	15	5	7	7	16	25	9	12	11	8	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction

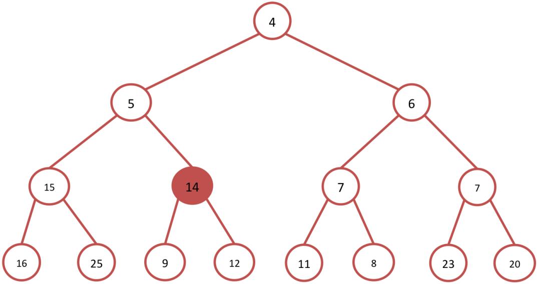


4	14	6	15	5	7	7	16	25	9	12	11	8	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction

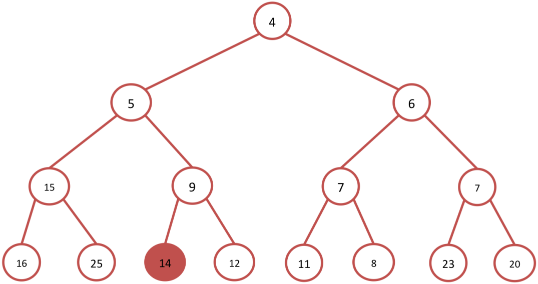


4	5	6	15	14	7	7	16	25	9	12	11	8	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction

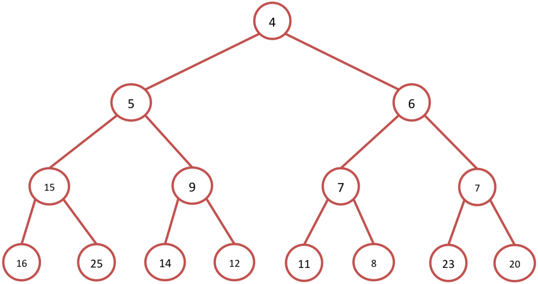


4	5	6	15	9	7	7	16	25	14	12	11	8	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction



4	5	6	15	9	7	7	16	25	14	12	11	8	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction
  - Suppose this construction is performed on an array with $n = 2^{h+1} - 1$ entries. What is the running time?

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction
  - Suppose this construction is performed on an array with $n = 2^{h+1} - 1$ entries. What is the running time?
  - Claim: The worst case running time is given by the expression:

$$F(h) = 2^{h-1} \cdot 1 + 2^{h-2} \cdot 2 + ... + 2^{h-h} \cdot h$$

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction
  - Suppose this construction is performed on an array with $n = 2^{h+1} - 1$ entries. What is the running time?
  - Claim: The worst case running time is given by the expression:

$$F(h) = 2^{h-1} \cdot 1 + 2^{h-2} \cdot 2 + ... + 2^{h-h} \cdot h$$

  - How do we simplify the above expression?

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction
  - Suppose this construction is performed on an array with $n = 2^{h+1} - 1$ entries. What is the running time?
  - Claim: The worst case running time is given by the expression:

$$F(h) = 2^{h-1} \cdot 1 + 2^{h-2} \cdot 2 + ... + 2^{h-h} \cdot h$$

  - We can write:

$$\begin{aligned} F(h) &= \sum_{i=0}^{h-1} 2^i + \sum_{i=0}^{h-2} 2^i + ... + \sum_{i=0}^{0} 2^i \\ &= (2^h - 1) + (2^{h-1} - 1) + ... + (2^1 - 1) \\ &= \sum_{i=1}^{h} 2^i - h \\ &= 2^{h+1} - 2 - h \end{aligned}$$

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction
    - Suppose this construction is performed on an array with $n = 2^{h+1} - 1$ entries. What is the running time?
    - Claim: The worst case running time is given by the expression:

$$F(h) = 2^{h-1} \cdot 1 + 2^{h-2} \cdot 2 + ... + 2^{h-h} \cdot h$$

    - We can write:

$$\begin{aligned} F(h) &= \sum_{i=0}^{h-1} 2^i + \sum_{i=0}^{h-2} 2^i + ... + \sum_{i=0}^{0} 2^i \\ &= (2^h - 1) + (2^{h-1} - 1) + ... + (2^1 - 1) \\ &= \sum_{i=1}^{h} 2^i - h \\ &= 2^{h+1} - 2 - h \leq 2^{h+1} - 1 = n \end{aligned}$$

    - So, the running time of bottom-up heap construction is $O(n)$.

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction in $O(n)$ time.

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction in $O(n)$ time.
- Question: Suppose you are given an unsorted array, can you use min-heap to sort the elements of the array?

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction in $O(n)$ time.
- Question: Suppose you are given an unsorted array, can you use min-heap to sort the elements of the array?

Algorithm

`HeapSort`(A, n)
- Perform bottom-up heap construction on the array A and let H denote the heap
- for $i = 1$ to n
  - $B[i] \leftarrow H.removeMin()$
- return(B)

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction in $O(n)$ time.
- Question: Suppose you are given an unsorted array, can you use min-heap to sort the elements of the array?

Algorithm

`HeapSort`(A, n)
- Perform bottom-up heap construction on the array A and let H denote the heap
- for $i = 1$ to n
  - $B[i] \leftarrow H.removeMin()$
- return(B)

- The above algorithm is called Heap Sort. What is the running time of this algorithm?

Problem

Given n entries create a min-heap of these entries.

- Method 1: Perform n `insert` operations in $O(n \log n)$ time.
- Method 2: Bottom-up heap construction in $O(n)$ time.
- Question: Suppose you are given an unsorted array, can you use min-heap to sort the elements of the array?

Algorithm

`HeapSort`(A, n)
- Perform bottom-up heap construction on the array A and let H denote the heap
- for $i = 1$ to n
  - $B[i] \leftarrow H.removeMin()$
- return(B)

- The above algorithm is called Heap Sort. What is the running time of this algorithm? $O(n \log n)$

Data Structures: Binary Search Trees

- Suppose we want to store n data entries where each data entry consist 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.

- Suppose we want to store n data entries where each data entry consist 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):

- Suppose we want to store n data entries where each data entry consist 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)`: $O(n)$

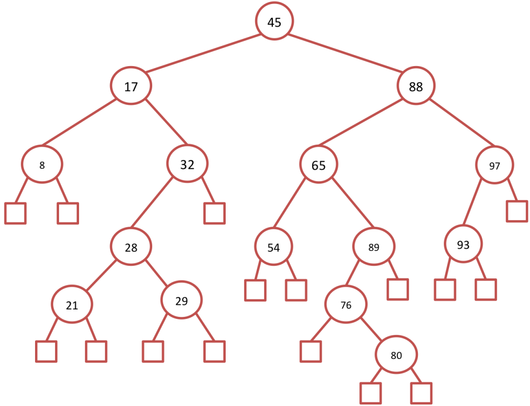
- Suppose we want to store n data entries where each data entry consist of a key (this can be thought of as a unique integer ID) and a value.
- We would like to perform the following operations:
  - get(k): Search an entry with key k and return the value.
  - put(k, v): Associate value v with key k, 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):

- Suppose we want to store n data entries where each data entry consist 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): $O(n)$

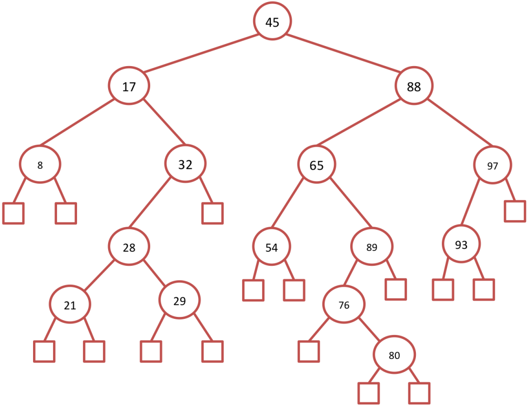
- Suppose we want to store n data entries where each data entry consist of a key (this can be thought of as a unique integer ID) and a value.
- We would like to perform the following operations:
  - get(k): Search an entry with key k and return the value.
  - put(k, v): Associate value v with key k, 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): $O(\log n)$

- 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

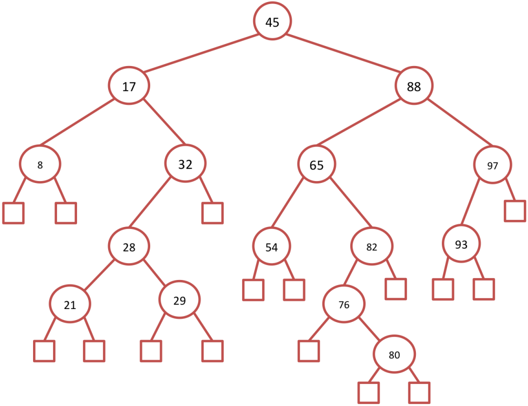
- Binary Search Tree: Binary Search Trees are proper binary trees such that each internal node p stores a key-value pair such that:
  1. Keys stored in the left sub-tree of p are less than k
  2. Keys stored in the right sub-tree of p are greater than k
- Is the tree below a binary search tree?



- 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



- 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?



- 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

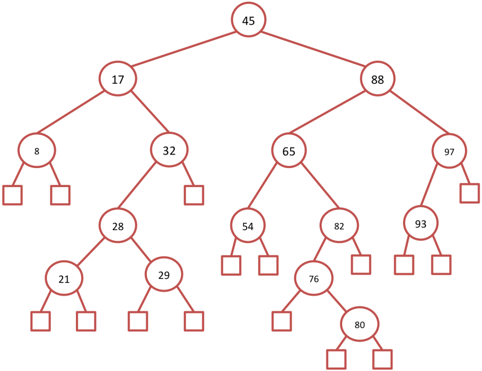
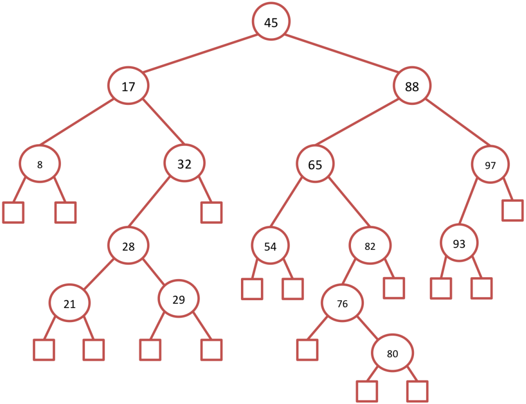
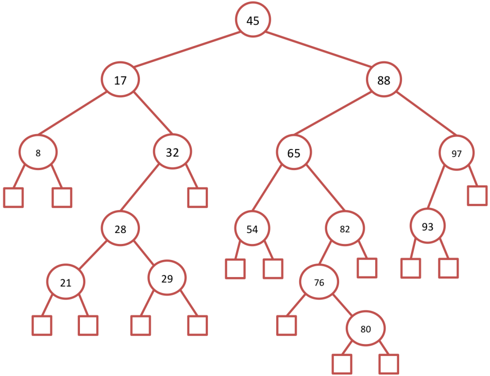


Figure : The "value" of entries are not shown.

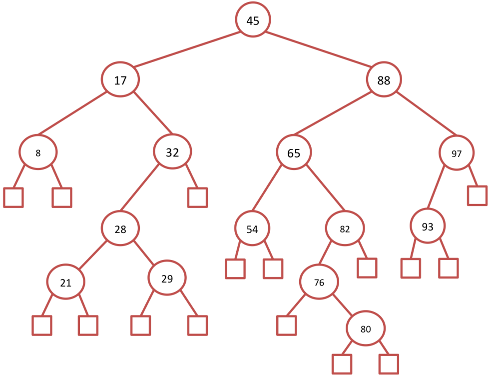
- Binary Search Tree: Binary Search Trees are proper binary trees such that each internal node p stores a key-value pair such that:
  1. Keys stored in the left sub-tree of p are less than k
  2. Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do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



- 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 `get(k)`?
  - How do we search for the key 68 in the binary search tree below?



- 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 `get(k)`?
  - Now try searching 76 in the tree.

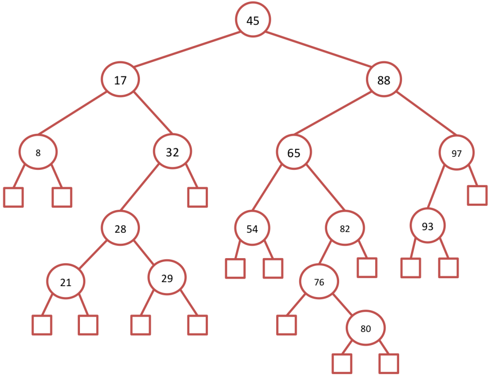


- Consider the following implementation:

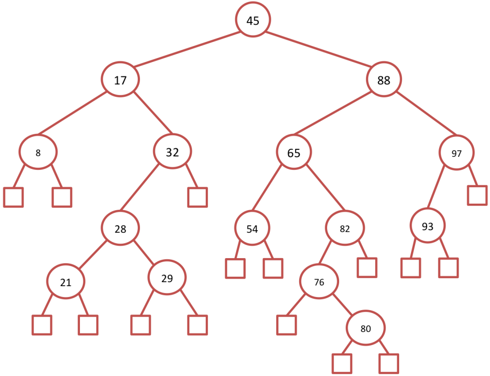
Code

```
class Node{
   public int key;
   public String value;
   public Node leftChild;
   public Node rightChild;
   public Node parent;
}
public class BST{
   public int size;
   public Node root;
   public BST(){
       size = 0;root = null;
   }
   public boolean isLeaf(Node N){//To be written}
   public String get(int k){//To be written}
}
```

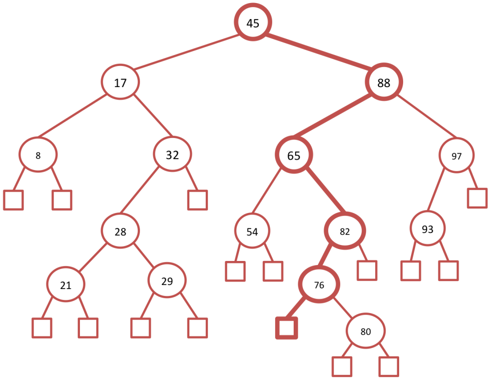
- Binary Search Tree: Binary Search Trees are proper binary trees such that each internal node p stores a key-value pair such that:
  1. Keys stored in the left sub-tree of p are less than k
  2. Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do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



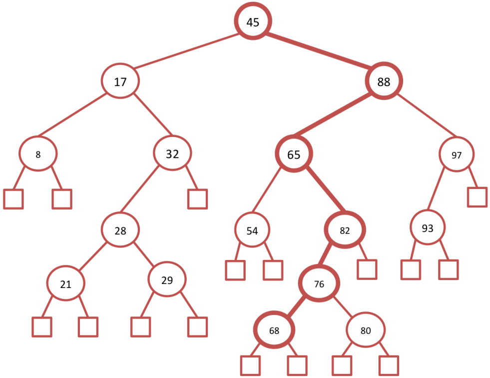
- Binary Search Tree: Binary Search Trees are proper binary trees such that each internal node p stores a key-value pair such that:
  1. Keys stored in the left sub-tree of p are less than k
  2. Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do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



- 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 `put(k, v)`?
  - Suppose we want to perform `put(68, "A")` in the BST below.



- 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 `put(k, v)`?
  - Suppose we want to perform `put(68, "A")` in the BST below.

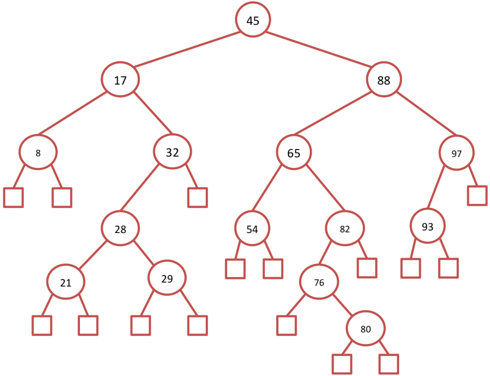


- Consider the following implementation:

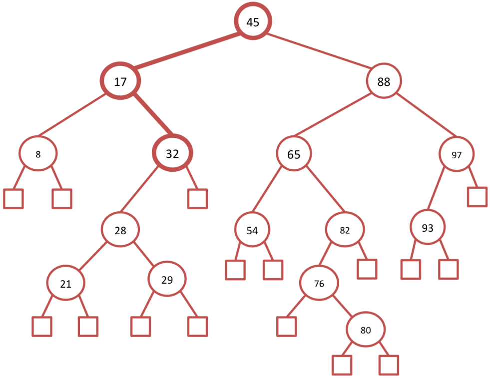
Code

```
class Node{
  public int key;
  public String value;
  public Node leftChild;
  public Node rightChild;
  public Node parent;
}
public class BST{
  public int size;
  public Node root;
  public BST(){
    size = 0;root = null;
  }
  public boolean isLeaf(Node N){//To be written}
  public String get(int k){//To be written}
  public void put(int k, String v){//To be written}
}
```

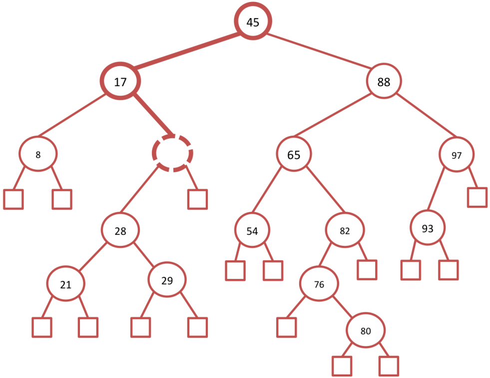
- Binary Search Tree: Binary Search Trees are proper binary trees such that each internal node p stores a key-value pair such that:
  1. Keys stored in the left sub-tree of p are less than k
  2. Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do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



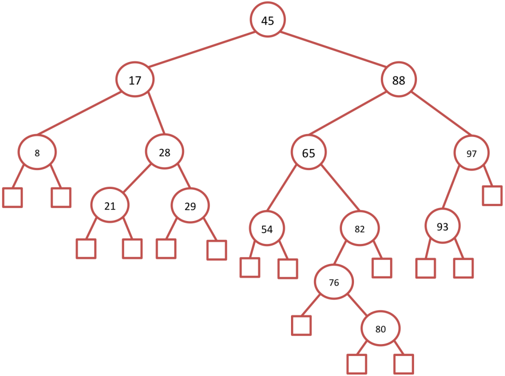
- Binary Search Tree: Binary Search Trees are proper binary trees such that each internal node p stores a key-value pair such that:
  1. Keys stored in the left sub-tree of p are less than k
  2. Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do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



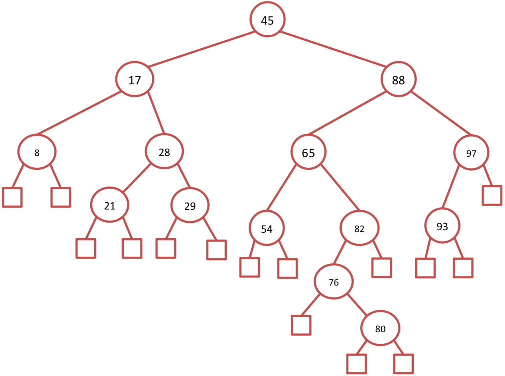
- Binary Search Tree: Binary Search Trees are proper binary trees such that each internal node p stores a key-value pair such that:
  1. Keys stored in the left sub-tree of p are less than k
  2. Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do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



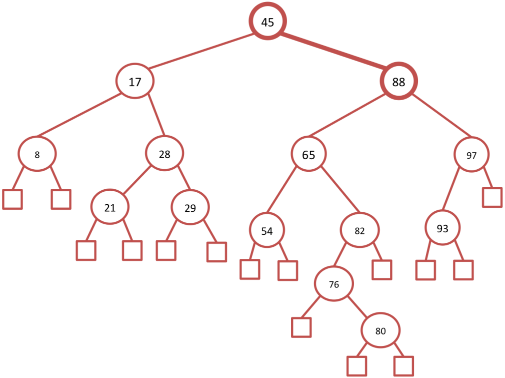
- Binary Search Tree: Binary Search Trees are proper binary trees such that each internal node p stores a key-value pair such that:
  1. Keys stored in the left sub-tree of p are less than k
  2. Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do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



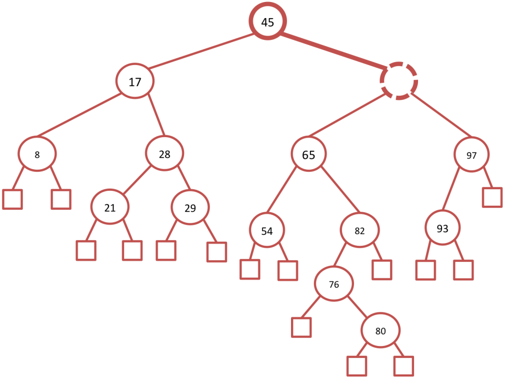
- Binary Search Tree: Binary Search Trees are proper binary trees such that each internal node p stores a key-value pair such that:
  1. Keys stored in the left sub-tree of p are less than k
  2. Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do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



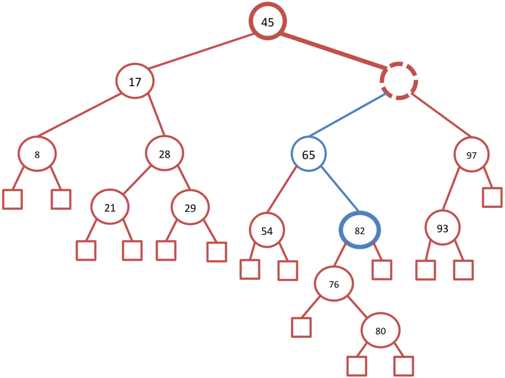
- Binary Search Tree: Binary Search Trees are proper binary trees such that each internal node p stores a key-value pair such that:
  1. Keys stored in the left sub-tree of p are less than k
  2. Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do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



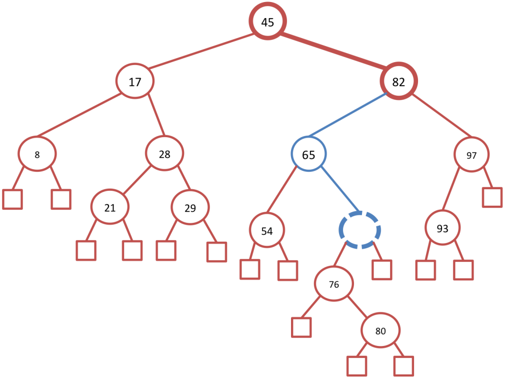
- Binary Search Tree: Binary Search Trees are proper binary trees such that each internal node p stores a key-value pair such that:
  1. Keys stored in the left sub-tree of p are less than k
  2. Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do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



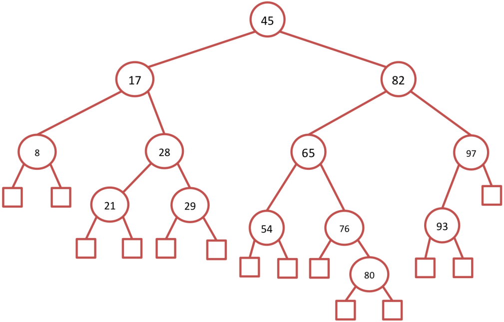
- Binary Search Tree: Binary Search Trees are proper binary trees such that each internal node p stores a key-value pair such that:
  1. Keys stored in the left sub-tree of p are less than k
  2. Keys stored in the right sub-tree of p are greater than k
- Question: Given a binary search tree, how do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



- 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 `remove`(k)?
  - Suppose we want to perform `remove(88)` in the BST below.



- 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 `remove(k)`?
  - Suppose we want to perform `remove(88)` in the BST below.



- Consider the following implementation:

Code

```
class Node{
   public int key;
   public String value;
   public Node leftChild;
   public Node rightChild;
   public Node parent;
}
public class BST{
   public int size;
   public Node root;
   public BST(){
      size = 0;root = null;
   }
   public boolean isLeaf(Node N){//To be written}
   public String get(int k){//To be written}
   public void put(int k, String v){//To be written}
   public void remove(int k){//To be written}
}
```

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

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

- 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 are equal number of nodes in its left and right sub-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

- Consider the following implementation:

Code

```
class Node{
   public int key;
   public String value;
   public Node leftChild;
   public Node rightChild;
   public Node parent;
}
public class BST{
   public int size;
   public Node root;
   public BST(){
      size = 0;root = null;
   }
   public boolean isLeaf(Node N){//To be written}
   public String get(int k){//To be written}
   public void put(int k, String v){//To be written}
   public void remove(int k){//To be written}
}
```

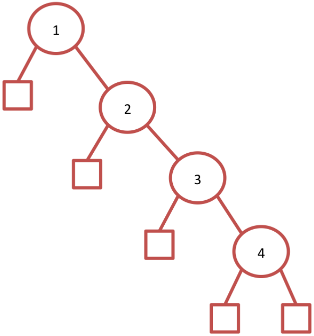
- What is the worst case running time of each of the following operations?
  - `get(k)`:
  - `put(k, v)`:
  - `remove(k)`:

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

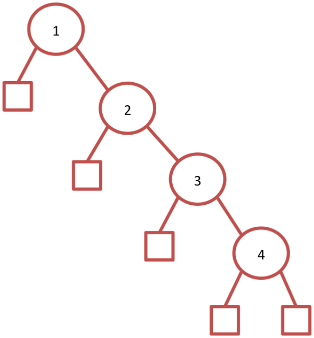
- What is the worst case running time of each of the following operations when the BST is balanced?
  - `get(k)`:
  - `put(k, v)`:
  - `remove(k)`:
- A BST is perfectly balanced if for every internal node, there are equal number of nodes in its left and right sub-trees.

- 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.

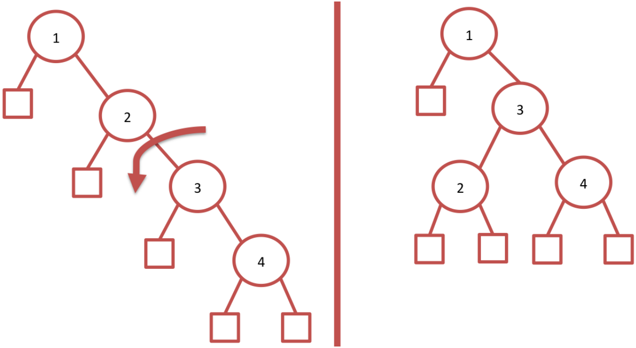
- Suppose we start with an empty BST and insert the keys $1, 2, 3, 4$, then the BST obtained is shown below.



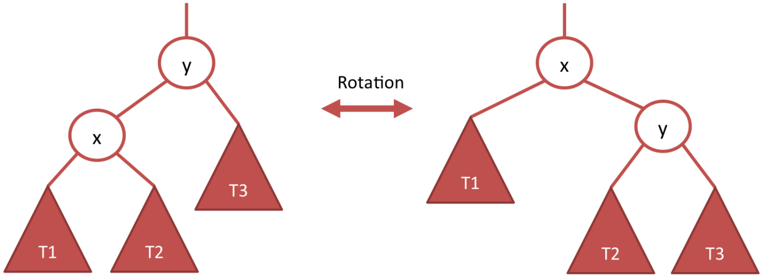
- Suppose we start with an empty BST and insert the keys $1, 2, 3, 4$, then the BST obtained is shown below.
- This tree is not balanced. Can you think of a way to balance this tree?



- 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?



- Rotation for tree balancing.



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

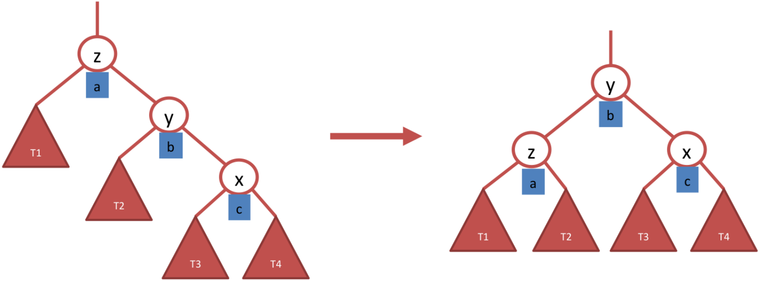


Figure : Case #1

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

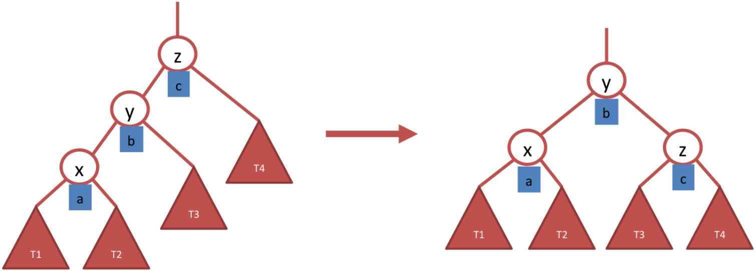


Figure : Case #2

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

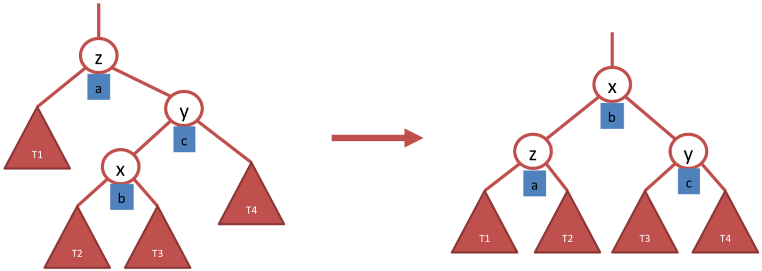


Figure : Case #3

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

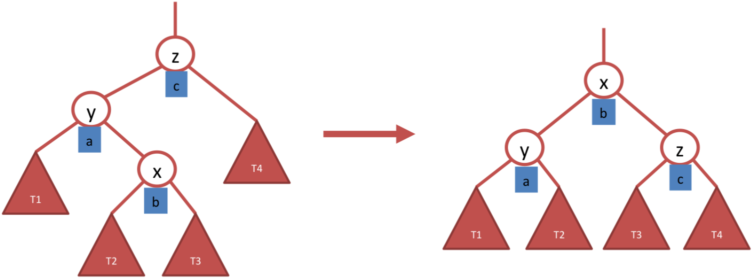
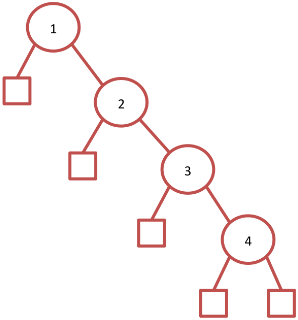


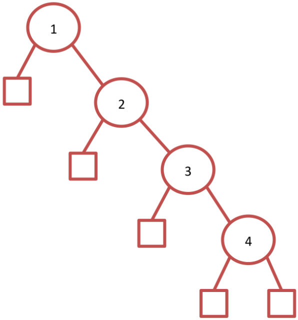
Figure : Case #4

- 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.

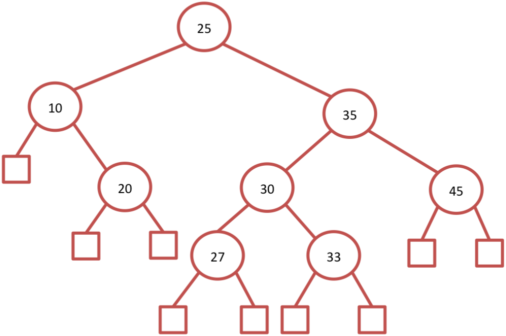
- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
  Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Is the binary search tree below an AVL tree?



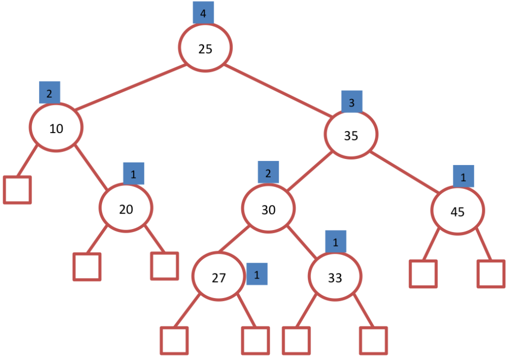
- AVL Tree: An AVL tree is a binary search tree that satisfies the following property:
  Height balance property: For every internal node of the tree, the heights of its children differ by at most 1.
- Is the binary search tree below an AVL tree? No



- 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?



- 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



- 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