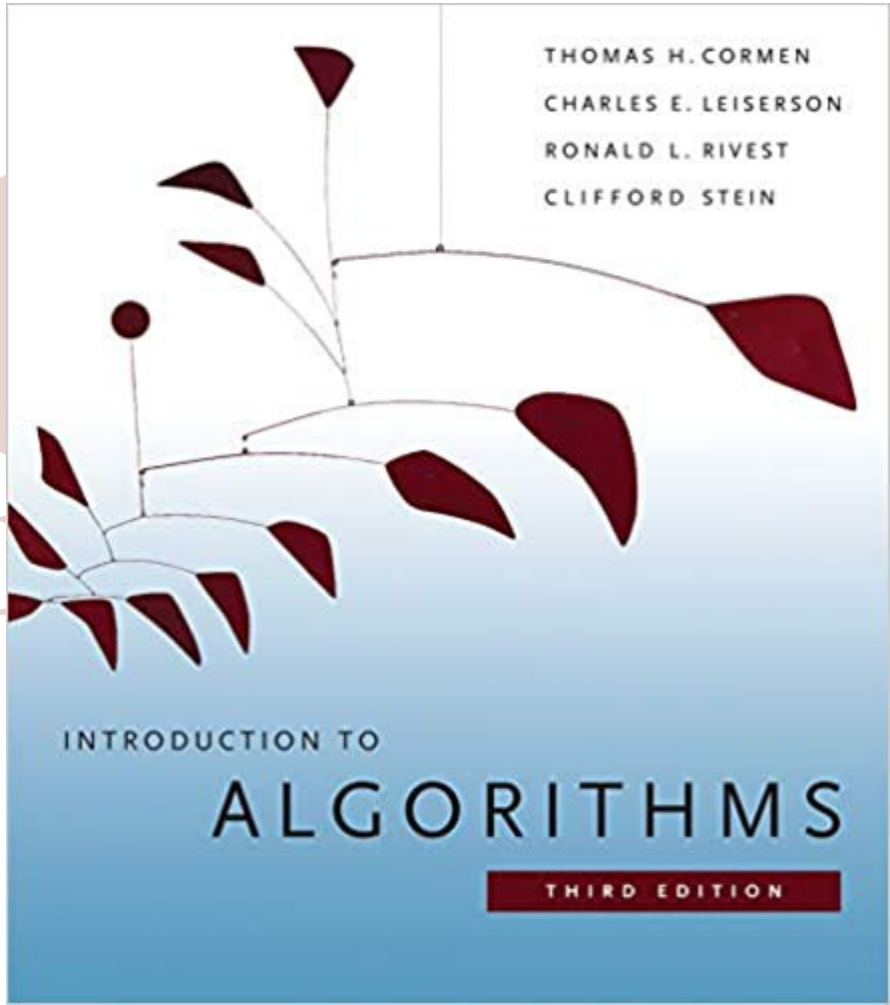
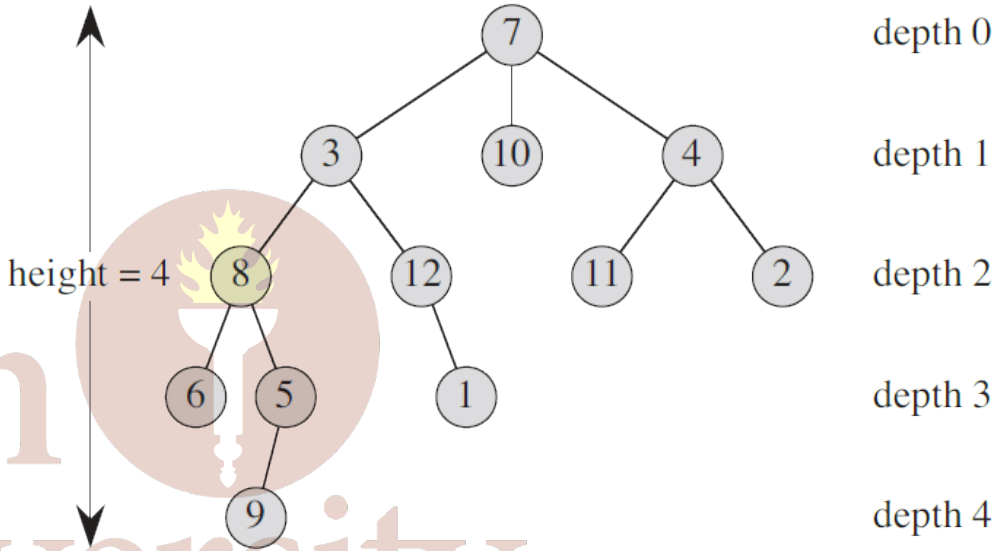
CS 07540 Advanced Design and Analysis of Algorithms

Week 3

- ADT Tree
  - Min-Heap and Max-Heap
  - Heap Sort
- Binary Search Trees
  - Implementation, Search, Minimum, Maximum, Predecessor, Successor, Insert, Delete
- Balanced Search Trees
- AVL Trees



ADT Tree



A tree is a connected undirected graph without cycles. Rooted trees have a special node called **root**. Trees can be represented graphically with nodes and edges such that nodes connected to the root (its **children**) are in a layer below the root. This can be extended to all nodes until we get a graph in which the well-defined distance of a node from the root determines its layer (called **depth**). A node connected

to a node in a lower layer is called its **parent**. Nodes without children are called **leaves**. Only root has no parent.

The length of the longest path from root to a leaf is called **height** of the tree. The structure below (and including) a node is called the **subtree** rooted at that node.

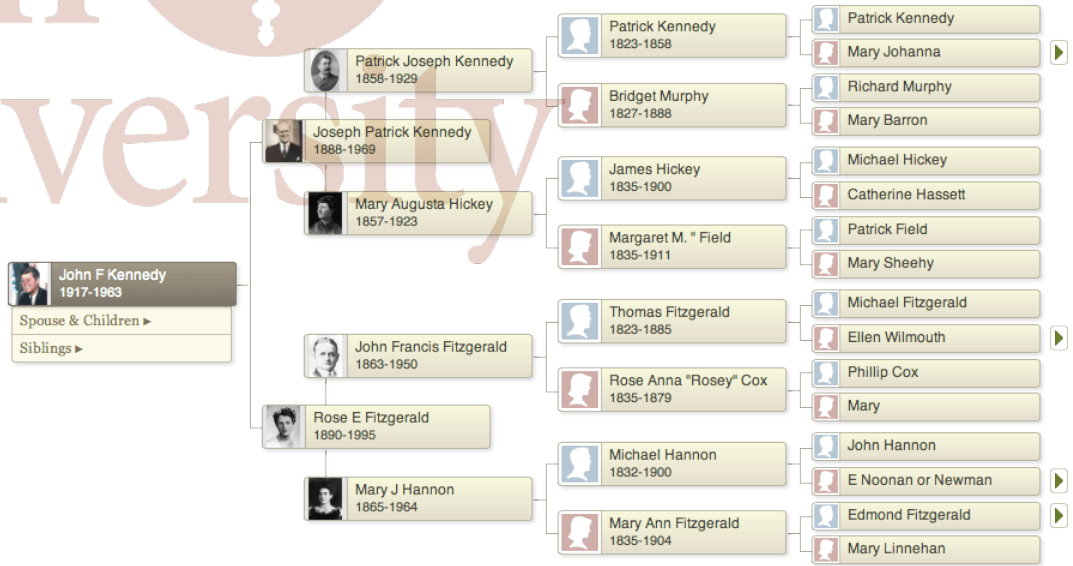
Trees are a widely used abstract data type in computer science. There are many specialized trees. Actual implementations are data structures. They require

- Memory locations/variables for each node
- References/pointers between nodes and children
    - Leaves *may* need to be identified

- Root needs to be identified

Family trees (upside down, with one descendent functioning as root and ancestors functioning as the children/descendants) are complete binary trees. Data **values** are the ancestors' information (as an object), and position (**key**) is determined by relation (姥姥 »lǎolao«: maternal grandmother, 姥爷 »lǎoye«: paternal grandmother).

Search in such a tree (called binary search trees due to their nature) is done by key, while the data is stored in the value.



Summary of ADT Tree Definitions

node element of the tree

root top node

child node connected and directly below given node

parent node connected and directly above child node

sibling nodes which share the same parent

ancestor parent, or parent's parent, or...

leaf node without children

subtree substructure below and connected to a given node

height length of longest path from root to any node

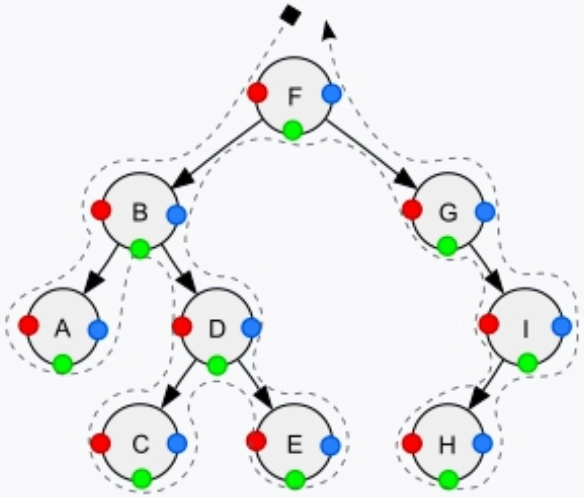
The minimal interface for ADT tree is usually

- Insert
- Delete
- Search

We can also include

- In-order walk
- Pre-order walk
- Post-order walk

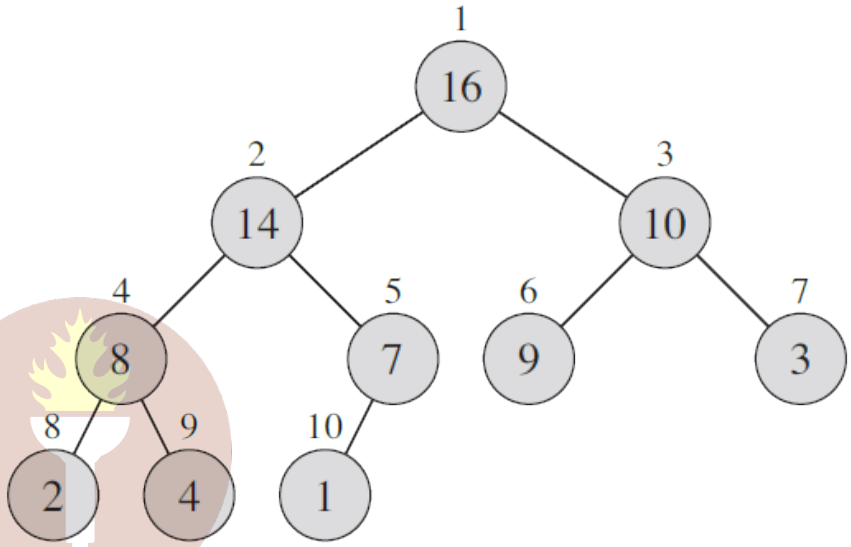
to enumerate all items in the tree (with a tree traversal using Depth-First Search).



Depth-first traversal (dotted path) of a binary tree:
Pre-order (node visited at position red ●):
F, B, A, D, C, E, G, I, H;
In-order (node visited at position green ●):
A, B, C, D, E, F, G, H, I;
Post-order (node visited at position blue ●):
A, C, E, D, B, H, I, G, F.

Heaps

A ***binary heap*** is a nearly complete binary tree filled on all levels except possibly the lowest level where leaves (key-values) are pushed left-most. Heaps are usually implemented as an array.

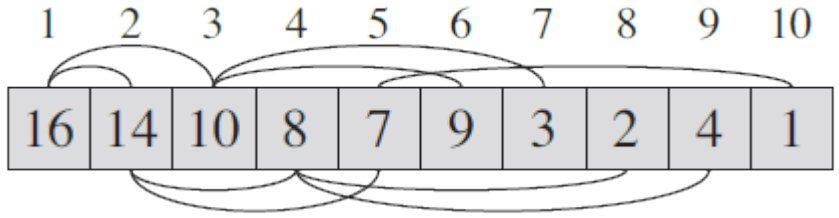


Each node in a heap satisfies the ***heap property***:

max-heap Every node's value is larger than the values of its children.

min-heap: Every node's value is less than the value of its children.

We will focus on creating proper heaps.



Relationship Between ADT Priority Queue and Min-Heap

```
class Node {
    int key;
    int value;
    Node leftChild;
    Node rightChild;
}

class Heap {
    private Node heapArray[];

    public void insert(Node node) {…
    }
    public Node remove() {…
    }
}

class priorityQueue {
    private Heap qHeap;

    public void insert(Node node) {
        qHeap.insert(node);
    }
    public Node remove() {
        return qHeap.remove();
    }
}
```

A priority queue is an ADT which has methods for inserting an item and removing the smallest (or largest) item. As seen, the methods for class **`priorityQueue`** are just wrappers for the underlying **`Heap`** class. A priority queue can be implemented in several ways – a heap being one – but a heap is a more fundamental data structure.

Min-Heap Methods Overview

`Min(H)`

- Return the value of the root element
- Runtime $O(1)$

`Insert(H, k)`

- Place element **`k`** into the right-most position of the last level
- Move up node until it satisfied the (min) heap property
- Runtime $O(\log(n))$

`DeleteMin(H)`

- Remove root
- Place the right-most leaf into the root position
- Move down node to satisfy the heap property
- Runtime $O(\log(n))$

Max-Heap Methods Overview

`Max(H)`

- Return the value of the root element
- Runtime $O(1)$

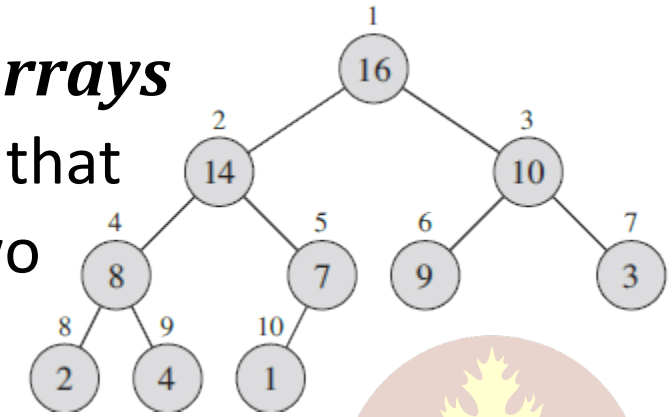
`Insert(H, k)`

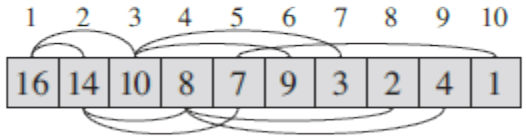
- Place element **`k`** into the right-most position of the last level
- Move up node until it satisfied the (max) heap property
- Runtime $O(\log(n))$

`DeleteMax(H)`

- Remove root
- Place the right-most leaf into the root position
- Move down node to satisfy the heap property
- Runtime $O(\log(n))$

Heaps Implemented as Arrays



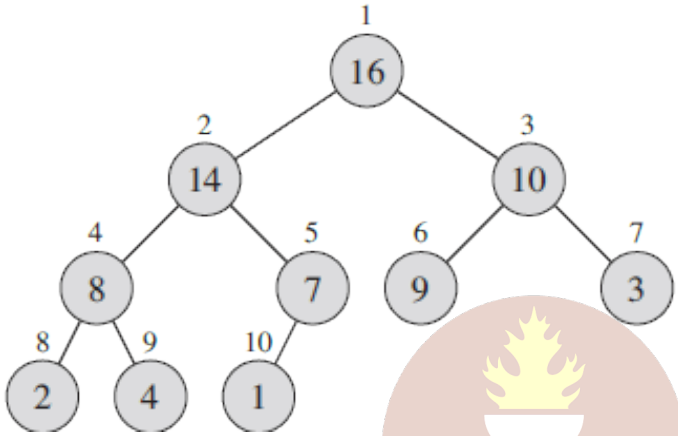


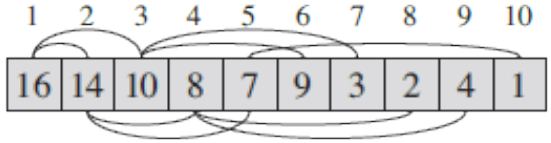
An array A of integer keys that represents a heap has two additional attributes (compared to an array):

- A.length
  - Number of elements the array could hold
- A.heap-size
  - Number of elements stored in the heap

Note that indices start from 1 instead of zero, and node entries are integer keys and not values. The root of the tree is $A[1]$. Entries outside $1 \ldots A$.heap-size are just ignored (no garbage collection).

A.length should be the nearest power of two when rounding up from A.heap-size to accommodate insertion of nodes.





$$A.\text{length} = 2^{\lceil \log_2(A.\text{heap-size}) \rceil}$$

Tree to Array

Because binary trees (where each node has at most two children, usually called ***left*** and ***right***) are so regular in their structure, we can represent heaps in an array without links or references.

```
Parent(i)
   return ⌊i/2⌋
Left(i)
   return 2i
Right(i)
   return 2i + 1
```

Determining position is $O(1)$.

Heap Property

The ***heap property*** is what allows operations to be performed quickly. Since we want to be able to find the minimum (or maximum) quickly, we want the minimum (or maximum) to be the root, and any node should be smaller (or larger) than its descendants.

max-heap

- $A[\text{Parent}(i)] > A[i]$

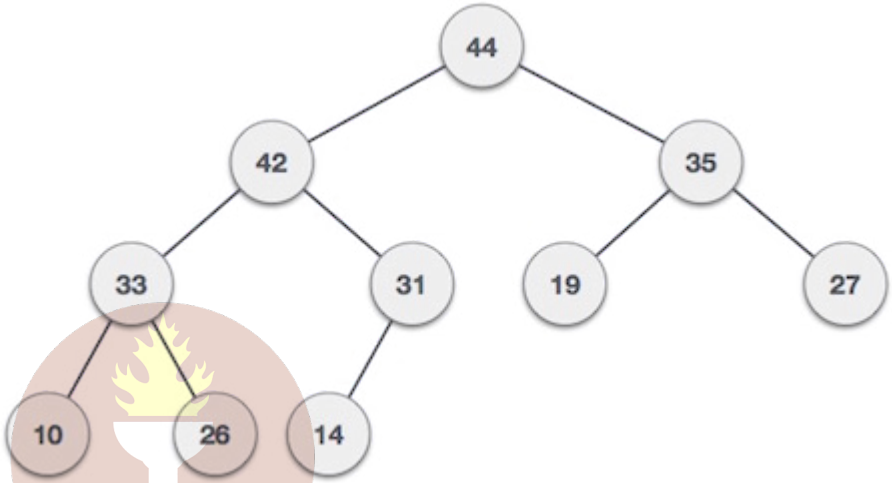
min-heap

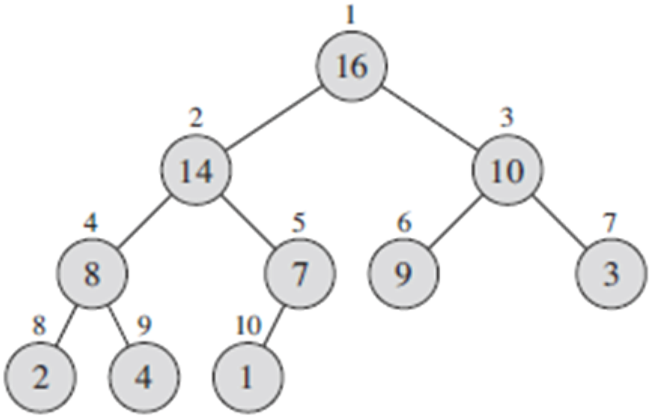
- $A[\text{Parent}(i)] < A[i]$

Where is the maximum in a max-heap? Where is the minimum?

The largest element in a max-heap is stored at the root. The minimum is not necessarily the last element in a max heap!

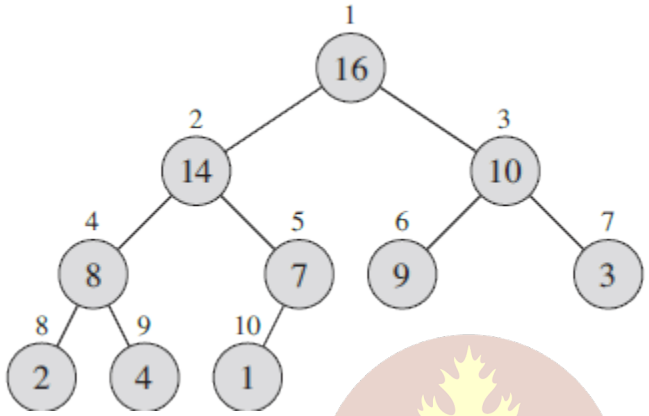
The smallest element in a min-heap is at the root.

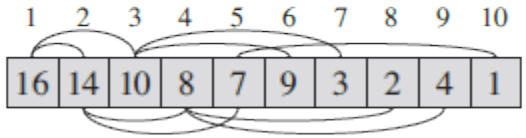




Example

Access calculations:





- $\text{Node}(2) = A[2] = 14$
- $\text{Parent}(2) = \lfloor 2/1 \rfloor = 1$
- $\text{Node}(1) = 16$
- $\text{Left}(2) = 2 \cdot 2 = 4$
- $\text{Node}(4) = 8$
- $\text{Right}(2) = 2 \cdot 2 + 1 = 5$
- $\text{Node}(5) = 7$

Heap property:

- $A[\text{Parent}(2)] = A[2] = 14 > A[\text{Left}(2)] = A[4] = 8$
- $A[\text{Parent}(2)] = A[2] = 14 > A[\text{Right}(2)] = A[5] = 7$

Which of the following arrays is ordered as a max heap?

25, 14, 16, 13, 10, 8, 12

25, 12, 16, 13, 10, 8, 14

25, 14, 12, 13, 10, 8, 16

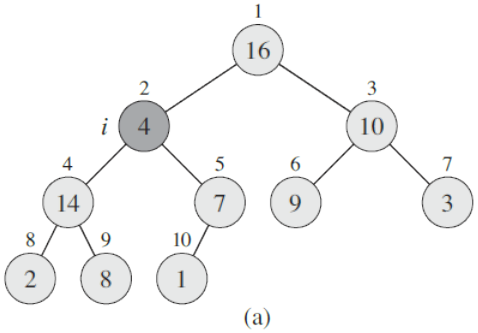
25, 14, 13, 16, 10, 8, 12

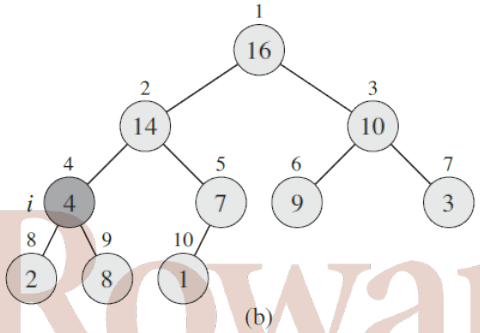
Heap Bottom Up

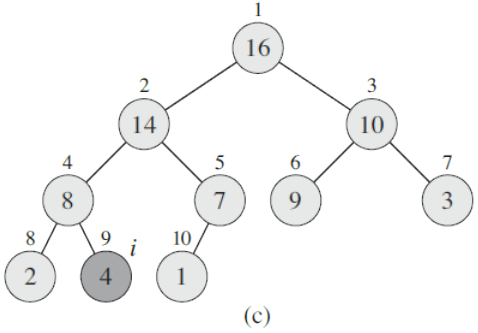
ALGORITHM *HeapBottomUp*$(H[1..n])$

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n    //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];    k ← j
    H[k] ← v
```

The algorithm ensures the local heap property. The local property (the while loop) translates to a global property.







Example: **`Max-Heapify(A,2)`**

1. Exchange $A[2]$ with $A[4]$
2. **`Max-Heapify(A,4)`**
3. Exchange $A[4]$ with $A[9]$
4. **`Max-Heapify(A,9)`**
5. Done

Heap Runtime Analysis

- Call to **MAX-HEAPIFY** costs $O(\log(n))$ due to tree-structure
- **BUILD-MAX-HEAP** calls **MAX-HEAPIFY** $O(n)$ times

Thus, the running time is $O(n \log(n))$. This upper bound is not asymptotically tight.

Time to **MAX-HEAPIFY** depends on the height of the node in the tree. Heights of most nodes are small. An n-element heap

- has height floor $\log(n)$ and
- has at most ceil $\lceil n/2^{h+1} \rceil$ nodes of any height h

The time required by **MAX-HEAPIFY** when called on a node of height h is $O(h)$. Now we put everything together.

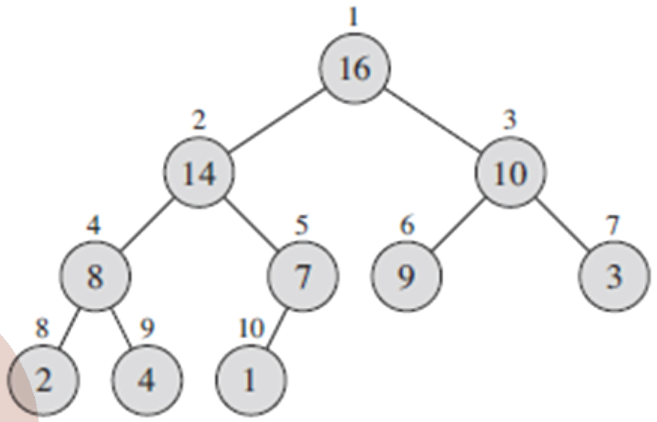
Total Cost of Build-Max-Heap

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) . \end{aligned}$$

Questions about Heaps



- What is the height of the given tree?
- What would its representation in array-form look like?
- If a heap corresponds to a tree of height h, how many elements may it maximally have? How many at a minimum?
- If a heap is implemented as an array, at which index is the minimum in a min-heap located?
- If a binary heap contains 126 elements (nodes), what is the height of the corresponding binary tree?

Heapsort

Heapsort is a comparison-based sorting algorithm. It can be implemented with a Min-Heap or a Max-Heap. Heapsort uses the heap property to identify the minimum (in a Min-Heap) or maximum (in a Max-Heap) to build up a sorted array.

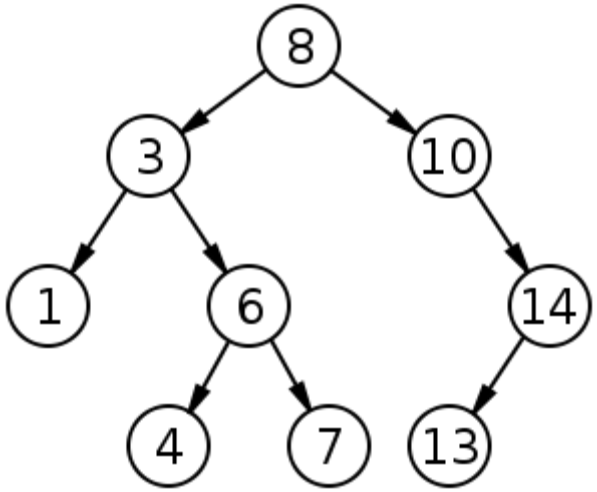
HEAPSORT(A)

```
1. BUILD-MAX-HEAP(A)
2. for i = A.length downto 2
3.     exchange A[1] with A[i]
4.     A.heap-size = A.heap-size - 1
5.     MAX-HEAPIFY(A,1)
```

HeapSort has runtime $O(n\log(n))$. We will prove this fact with the Master Theorem later.

Binary Search Trees

A **binary search tree** (BST, also called an ordered or sorted binary tree) is a linked-node based binary tree which stores key-value pairs (data) in each node. In addition, each node contains references to a



- left child (node)
- right child (node)
- parent (node) (optional, makes it doubly-linked)

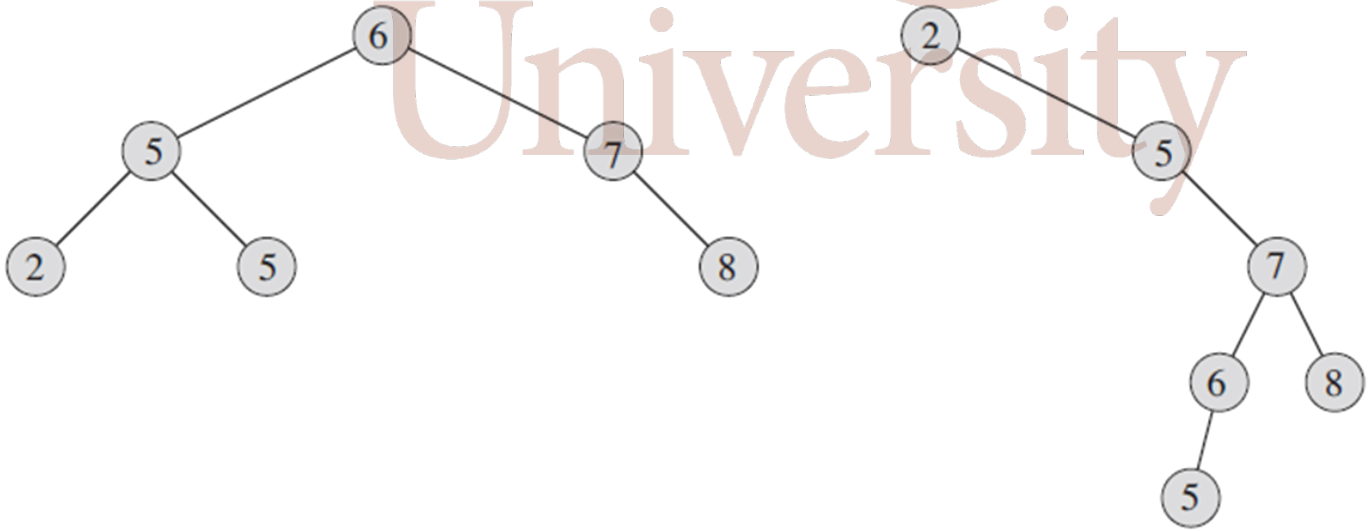
Nodes that are missing are referenced by **`null`** (Java) or **`None`** (Python). These are keywords to show that the object is not referring/pointing to anything. Left and right children are roots of left and right subtrees, respectively. Again, this approach is recursive in nature. All keys satisfy the following property:

- The left subtree contains only nodes with keys less than the node's key.
  - **`node.left.key < node.key`** (where **`node`** is the parent)
- The right subtree contains only nodes with keys greater than the node's key.
  - **`node.key < node.right.key`** (where **`node`** is the parent)
- The left and right subtree each must also be a binary search tree.

This is different from a heap. Max-Heaps have every node key larger than that of its children but no distinction or order for left and right.

These two trees are both valid BSTs containing the same data (meaning keys – values are not listed in this example.) BSTs can take on the form of a tree where each node has only one child (a linked list).

Question: In which order should known/sorted data be inserted to achieve balance? We will look at self-balancing trees later.



Example: The following algorithm seeks to compute the number of leaves in a binary tree. Is this algorithm correct? If it is, prove it; if it is not, make an appropriate correction.

ALGORITHM *LeafCounter*(T)

//Computes recursively the number of leaves in a binary tree
//Input: A binary tree T
//Output: The number of leaves in T
if $T = \varnothing$ **return** 0
else return *LeafCounter*(T_{left})+ *LeafCounter*(T_{right})

What would a recursive insert look like? We need to start at **`root`**, since that is the only known node (which may also be empty!).

```
RecursiveInsert(node, z)
if node == null            // empty child
  return z
if(z.key < node.key)     // left traversal
  node.left=RecursiveInsert(node.left, z)
elif(node.key < z.key) // right traversal
  node.right=RecursiveInsert(node.right, z)
else node=z
return node
```

Simple Integer Binary Search Tree Implementation

```
// binary search tree where key = value (of type int)
// nodes are singly-linked
public class intBST {
        private Node root;

        private class Node {
            private int key;
            private Node left, right;

            public Node(int key) {
                this.key = key;
            }
        }

        public intBST() {
        }

        public boolean isEmpty() {
            return size() == 0;
        }
```

```java
// public interface for tree size
public int size() {
    return size(root);
}

// implementation of size of a tree rooted at node
// returns number of non-empty node in subtrees
private int size(Node node) {
    if (node == null)
        return 0;
    else
        return 1 + size(node.left) + size(node.right);
}

// membership test
public boolean contains(int key) {
    return get(key) != null;
}
```

```java
// public interface for get
public Node get(int key) {
    return get(root, key);
}

// recursive implementation for get
private Node get(Node node, int key) {
    if (node == null)
        return null;
    if  (key < node.key)
        return get(node.left, key);
    else if (node.key < key)
        return get(node.right, key);
    else
        return node;
}

// public interface for insert
public void insert(int key) {
    root = insert(root, key);
}
```

```
        // recursive implementation of insert
        private Node insert(Node node, int key) {
            if (node == null) {
                node = new Node(key);
                return node;
            }
            if (key < node.key)
                node.left = insert(node.left, key);
            else if (key > node.key)
                node.right = insert(node.right, key);
            return node;
    }
}
```


Note that in order to keep the interface clean, we use **`public`** and **`private`** keywords for functions.

A similar object (with key-value pairs) can be implemented in Python:

```
class Node:
    def __init__(self, key, value):
        self.left = None
        self.right = None
        self.key = key
        self.value = value

class BST:
    def __init__(self):
        self.root = None

    def isEmpty(self):
        return self.root == None
```


Python does not have public/private keywords, but we can create private functions by naming them with the double underscore (this is Python-specific), such as **`insert`** (public interface) and **`__insert`** (private). In addition, Python has built-in "dunder" functions such as **`__getitem__`** which we can overload to get custom behavior with standard notation.

```
# Python size method
def __len__(self):
    return self.size(self.root)

def size(self, node):
    if node is None:
        return 0
    return 1 + self.size(node.left) + self.size(node.right)

# Python <is in> method
def __contains__(self, key):
    return self.isElementOf(self.root, key)

def isElementOf(self, node, key):
    if node is None:
        return False
    if key < node.key:
        return self.get(node.left, key)
    elif node.key < key:
        return self.get(node.right, key)
    else:
        return True
```

```
# Python accessor value = BST[key]
def __getitem__(self, key):
    return self.__get(self.root, key)

# recursive part of getter __getitem__
def __get(self, node, key):
    if node is None:
        raise KeyError
    if key < node.key:
        return self.__get(node.left, key)
    elif node.key < key:
        return self.__get(node.right, key)
    else:
        return node.value
```

```
# public interface of insert
def insert(self, key, value):
    self.root = self.__insert(self.root, key, value)

# recursive part of insert
def __insert(self, node, key, value):
    if node is None:
        return Node(key, value)
    if key < node.key:
        node.left = self.__insert(node.left, key, value)
    elif node.key < key:
        node.right = self.__insert(node.right, key, value)
    else:
        node.key = key
        node.value = value
    return node

# Python modifier BST[key] = value
def __setitem__(self, key, value):
    self.root = self.insert(self.root, key, value)
```

Overloading **`__setitem__`** affords us to use notation

```
BST[key] = value
```

instead of **`BST.insert(key, value)`** (or in addition to) while overloading **`__getitem__`** makes

```
value = BST[key]
```

work.

The BST property affords us to retrieve all data ***in order*** in $O(n)$ time by performing an in-order tree walk.

- In-order walk
  - left, root, right
- Pre-order walk
  - root, left, right
- Post-order walk
  - left, right, root


```
INORDER-WALK(node)
  if (node != null)
    INORDER-WALK(node.left)
    Print node.key
    INORDER-WALK(node.right)
```

Java Implementation

We will pass a list by reference to collect all keys.

```
# public interface for in-order walk
public List<Integer> inOrder() {
    List<Integer> keys = new LinkedList<>();
    inOrder(this.root, keys);
    return keys;
}

# recursive private implementation for in-order walk
private void inOrder(Node node, List<Integer> keys) {
    if (node != null) {
        inOrder(node.left, keys);
        keys.add(node.key);
        inOrder(node.right, keys);
    }
}
```

Python Implementation

```
def inOrder(self):
    keys = []
    self._inOrder(self.root, keys)
    return keys

def __inOrder(self, node, keys):
    if node is not None:
        self.__inOrder(node.left, keys)
        keys.append(node.key)
        self.__inOrder(node.right, keys)
```


This recursive in-order walk collects all keys in the list **`keys`**. Note that **`append`** adds elements to the end of a list. Once again, we use a private member method by starting its name with double underscore.

Python Example with JSON Data

JSON (JavaScript Object Notation) is a lightweight data-interchange format using key-value pairs, possibly nested. Python has support for JSON in form of the **`json`** library. Due to the similarity of data formats, we can think of JSON objects as Python dictionaries (a key-value data structure, possibly nested). In this example, we will associate episode information with an alpha-numeric key of form **`SxxEyy`** (of type string).

```python
import json
# using "with" closes resource automatically
def readJSONfile2Dict(filename):
    with open(filename, 'r') as jsonFile:
        data = json.load(jsonFile)
        return data
```

`json.load()` will read the json file sequentially, so the order in which data was stored in the file will appear in the dictionary. When we store the contents of the dictionary in a binary search tree, this order will determine the height of the search tree. Hence, we want to write a function which orders the entries first so that it minimizes the height of the tree.

```
from bst import BST
from helper import readJSONfile2Dict, dict2balancedDict
from copy import deepcopy

if __name__ == '__main__':
    episodeGuide = BST()
    episodes = readJSONfile2Dict("WildKrattsDB.json")
    for episode in episodes:
        episodeGuide.insert(episode, episodes[episode])
    print("size of dictionary from ordered file: ", len(episodes))
    print("height of tree:", episodeGuide.depth())
    episodeGuide2 = BST()
    episodes2 = readJSONfile2Dict("WildKrattsDB2.json")
    for episode in episodes2:
        episodeGuide2.insert(episode, episodes2[episode])
    print("size of dictionary from unordered file: ", len(episodes2))
    print("height of tree:", episodeGuide2.depth())
    episodeGuide3 = BST()
    episodes3 = dict2balancedDict(episodes)
    for episode in episodes3:
        episodeGuide3.insert(episode, episodes3[episode])
    print("size of dictionary from ordered file: ", len(episodes))
    print("height of tree from balanced dictionary:", episodeGuide3.depth())
```

```
size of dictionary from ordered file:  119
height of tree: 119
size of dictionary from unordered file:  79
height of tree: 14
size of dictionary from ordered file:  119
height of tree from balanced dictionary: 7
```

The key observation is to traverse the ordered list data in a ***pre-order walk*** (median, left list, right list). We can do a little better by overwriting the dunder method **`__repr__`**.

```
def __repr__(self):
    if self.root == None:
        return "empty binary search tree"
    else:
        size = self.size(self.root)
        depth = self.depth()
        return f"binary search tree of size {size} and depth {depth}"
```

Python also provides an easy to use interface to create a generator object which we can use to implement an iterator for an in-order walk.

```
class Node:
…
    def __iter__(self):
        if self.left:
            yield from self.left
        yield self
        if self.right:
            yield from self.right


class BST:

…

def __iter__(self):
    if self.root:
        for node in self.root.__iter__():
            yield node.key
```

```
for episode in episodeGuide: # unbalanced
    print(episode)
```

```
S01E01
S01E02
S01E03
S01E06
S01E07
…
```

```
for episode in episodeGuide3: # balanced
    print(episode)
```

```
S01E01
S01E02
S01E03
S01E06
S01E07
…
```


In-order walks provide the correct order regardless of the tree balance.

Search

We want to traverse the tree until the key is found or we reached a **`null`** node. Return the pointer to the node with given key or **`null`** if not found. Run-time will be $O(h)$, where h is the height of the binary search tree. Note that height is always between $\log(n)$ (in case of a perfectly balanced BST) and $n-1$ (in case of a linked list).

```
TREE-SEARCH(node, key)
if (node == null or key == node.key)
  return node
if (node == null or key == node.key)
  return TREE-SEARCH(node.left, key)
else
  return TREE-SEARCH(node.right, key)
```

Search is just an alias for the method **`get`** already implemented. Note that this is a recursive implementation. Java code for a **`key==value`** of integer type would look as follows:

```
// public interface for get
public Node get(int key) {
    return get(root, key);
}

// recursive implementation for get
private Node get(Node node, int key) {
    if (node == null)
        return null;
    if  (key < node.key)
        return get(node.left, key);
    else if (node.key < key)
        return get(node.right, key);
    else
        return node;
}
```

We can unroll the recursive calls into a loop as follows.

```
ITERATIVESEARCH(node, key)
  while node != null and key != node.key
    if key < node.key
      node = node.left
    else
      node = node.right
  return node
```

Minimum and Maximum

We can always find a minimum value in a binary search tree by following the left child pointers from root until we encounter a leaf (whose left child will be **`null`**).

```
MINIMUM(node)
   while node.left != null
      node = node.left
   return node
```


The binary search tree property guarantees that this minimum will be correct since any key in a right subtree would have a larger value than the current node. Run-time is $O(h)$ where h is height.

```
MAXIMUM(node)
   while node.right != null
      node = node.right
   return node
```

Successor

We want to find the successor of a given node in sorted. If all keys are distinct then the successor of a **node** is the node with the smallest key greater than **node.key**. The structure of a BST lets us determine the successor of a node without ever comparing keys. The successor of **node** is either the minimum node in the right subtree of **node** or the first parent of **node** for which **node** is located in the "left parent sub-tree"

```
SUCCESSOR(node)
  if node.right != null
    return MINIMUM(node.right)
  parent = node.parent
  while parent != null and node == parent.right
    node = parent
    parent = parent.parent
  return parent
```

Predecessor

The predecessor of **node** is the node with the largest key smaller than **node.key**. Hence the predecessor of **node** is either the maximum node in the left subtree of node or the first parent of node for which node is located in the "right parent sub-tree".

```
PREDECESSOR(node)
  if node.left != null
    return MAXIMUM(node.left)
  parent = node.parent
  while parent != null and node == parent.left
    node = parent
    parent = parent.parent
  return parent
```

In order to use the parent pointer, we need to update our methods for node creation and insertion.

```
public void insert(int key) {
    this.root = insert(this.root, key);
    this.root.parent = null;
}

private Node insert(Node node, int key) {
    if (node == null) {
        node = new Node(key);
        return node;
    }
    if (key < node.key) {
        node.left = insert(node.left, key);
        node.left.parent = node;
    }
    else if (key > node.key) {
        node.right = insert(node.right, key);
        node.right.parent = node;
    }
    return node;
}
```

If we change the interface to the user world, predecessor should be the next smaller integer in our tree (or a predefined value saying “Not found” – for integers, the **MIN_VALUE** in Java or **None** in Python make sense).

```
public int predecessor(int key) {
    Node node = search(key);
    Node pre = predecessor(node);
    if (pre != null)
        return pre.key;
    else
        return Integer.MIN_VALUE;
}

private Node predecessor (Node node) {
    if (node.left != null)
        return maximum(node.left);
    return maximum(node.right);
}
```

Insert

As with **SEARCH**, the **INSERT** method we implemented was recursive. A non-recursive approach should take a BST T and a node with preset **`key`** and **`null`** children. Remember that a tree can be represented by its root. We start at the root and trace a simple path downward looking to find a **`null`** node to be replaced with a new node and value. We want to modify the data structure in such a way that the binary search tree property continues to hold.

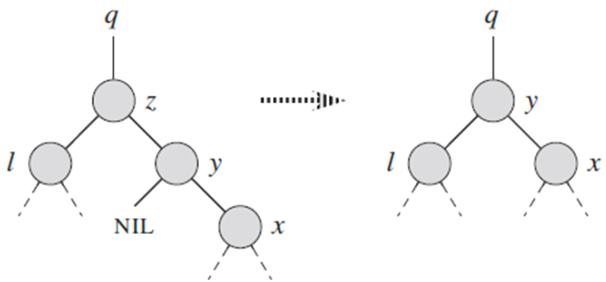
```
ITERATIVEINSERT(Tree, node)
  p = null                          // trailing parent ptr
  r = Tree.root                     // root of subtree
  while r != null
    p = r
    if node.key < r.key             // go down left
      r = r.left
    else
      r = r.right                   // go down right
  node.parent = p
  if p == null                      // empty tree
    T.root = node
  elseif node.key < p.key           // attach as left…
    p.left = node                   //  …subtree of parent
  else
    p.right = node                  // attach as right…
```

Delete

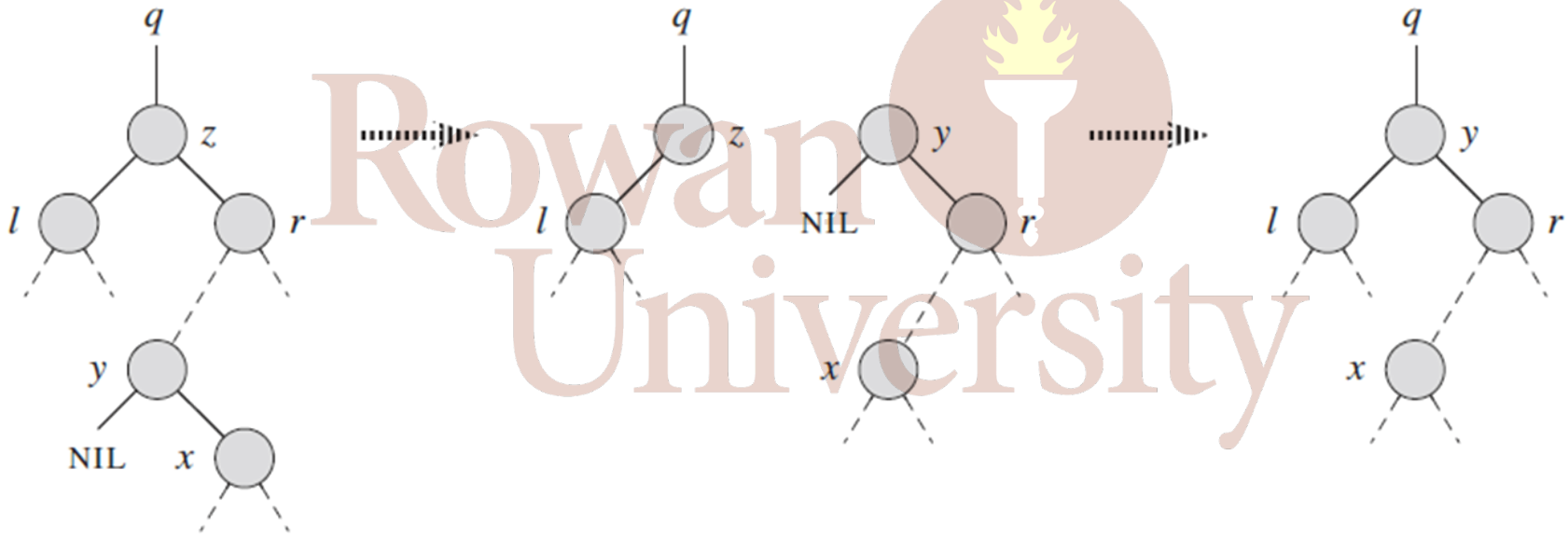
Deleting a node from a BST involves three cases:

- If node z does not have children, we remove it by replacing its parent pointer to z with **`null`**.
- If z has one child we elevate this child to take its place in the tree.

- If z has two children then we find z's successor y, which must be in z's right subtree.
  - What should be true about y's left subtree if it is z's successor?
  - What should be true about z if its successor is not one of its children?
- **Case 1**: y is z's right child:
  - Have y take z's position in the tree
  - The rest of z's original right subtree becomes y's new right subtree
  - z's left subtree becomes y's new left subtree. Why is this possible?



- **Case 2**: y is not z's right child, y lies within z's right subtree
  - Replace y with its right child
  - Replace z with y



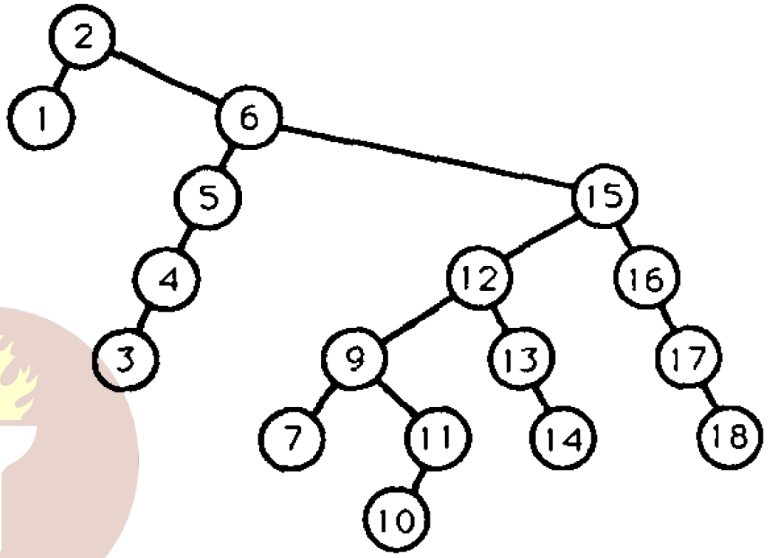
Transplant

To move subtrees within a BST create a method **TRANSPLANT(T,u,v)** which replaces the subtree rooted at node **u** with the subtree rooted at node **v**. Node **u**'s parent becomes node **v**'s parent. Node **u**'s parent ends up having **v** as its appropriate child.

```
TRANSPLANT(T, u , v)
  if (u.parent == null)
    T.root = v              // u is the root of T
  elseif (u == u.parent.left)
    u.parent.left = v       // u is the left child
  else
    u.parent.right = v      // u is the right child
  if (v != null)
    v.parent = u.parent
```

```
TreeDelete(T,  z)
  if (z.left == NIL and z.right == null) // z is a leaf
    if (z == z.parent.left)
      z.parent.left = null
    else
      z.parent.right = null
    elseif (z.left == null)     // z has only right child
      Transplant(T, z, z.right)
  elseif (z.right == null)      // z has only leftchild
    Transplant(T, z, z.left)
  else y = Minimum(z.right)     // find z succ right tree
    if (y.parent != z)          // y not left child of z
      Transplant(T, y, y.right) // replace y w/ right chld
      y.right = z.right         // set y's right child
      y.right.parent = y
    Transplant(T, z, y)         // replace z with y
    y.left = z.left
    y.left.parent = y
  remove z
```

Binary Search Tree Traversal



Drawing trees automatically is extremely challenging. However, once in a while it would be nice to have a method that prints the keys of a BST in a human-readable text form.

For example, the given tree from the Galperin/Rivest 1993 paper introducing Scapegoat Trees should yield the string

```
2(1,6(5(4(3,  ),  ),15(12(9(7,11(10,  )),13(  ,14)),16(  ,17(  ,18)))));.
```

Example: Devise a **`toString`** method for a BST that prints keys as given.

Solution:

```
private void toStringRecursive(Node<Key, Value> node, StringBuilder rep) {
    if(node != null) {
        rep.append(node.key);
        if(node.left != null || node.right !=null) {
            rep.append("(");
            toStringRecursive(node.left, rep);
            rep.append(",");
            toStringRecursive(node.right, rep);
            rep.append(")");
        }
    }
    else
        rep.append(" ");
}

// Java String is immutable so we need StringBuilder
@Override
public String toString() {
    StringBuilder representation = new StringBuilder("ScapegoatTree of size "+this.size()+" and height "+this.depth()+": ");
    toStringRecursive(root, representation);
    return representation.toString();
}
```

BST Summary

BST operations (Member, Min, Max, Insert, Delete) **always** have run-time $O(h)$, where h is the height of the tree. However, the height of the BST varies depending on the sequence of operations.

The worst-case scenario is when $h = n$. Elements are inserted into a tree in order and form a singly-linked list. As discussed earlier, if all data is known we can create a balanced BST by starting with the ***median*** as root and then recursively insert the left subset and right subset (median by median). There is a formula to accomplish this in a loop using entries at index $\left(\frac{n}{2}, \frac{n}{4}, \frac{3n}{4}, \frac{n}{8}, \frac{3n}{8}, \frac{5n}{8}, \frac{7n}{8}, \ldots\right)$ from an ordered list, but the rounding needs to be done carefully.

We can randomize data to get an average case.

Balanced Search Trees

A self-balancing search tree is any node-based search tree that automatically keeps its height small during arbitrary item insertions and deletions. Examples include

- AVL Trees
- Red-Black Trees
- Scapegoat Trees
- 2-3 Trees (not binary)
- B-Trees (not binary)

A very pretty solution to the problem of maintaining a good search tree was discovered in 1962 by two Russian mathematicians, G. M. Adelson-Velsky and E. M. Landis [*Doklady Akademiâ Nauk SSSR* **146** (1962), 263–266; English translation in *Soviet Math.* **3**, 1259–1263]. Their method requires only two extra bits per node, and it never uses more than $O(\log N)$ operations to search the tree or to insert an item. In fact, we shall see that their approach also leads to a general technique that is good for representing arbitrary *linear lists* of length N, so that each of the following operations can be done in only $O(\log N)$ units of time:

i) Find an item having a given key.

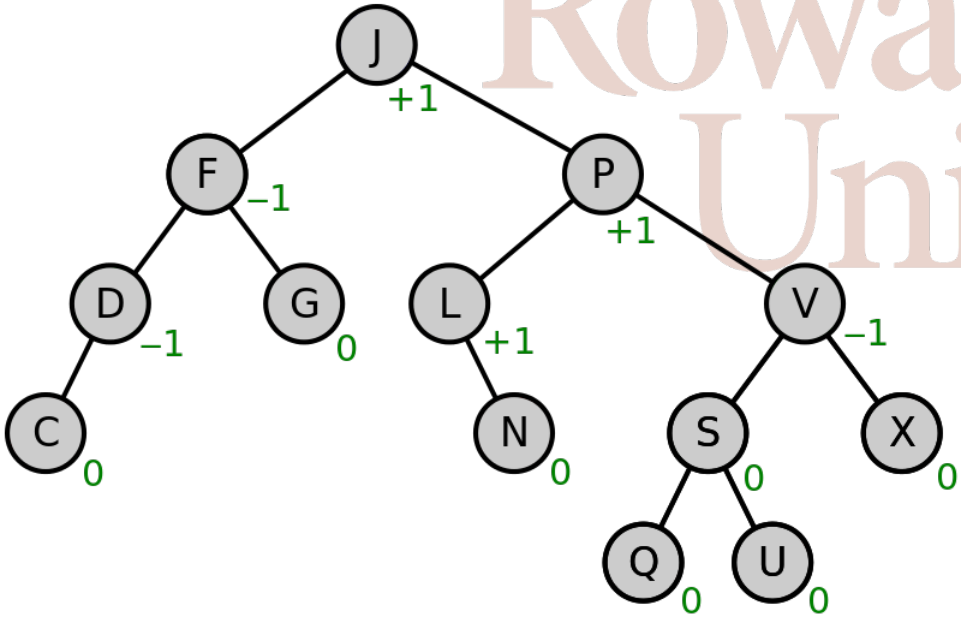
ii) Find the kth item, given k.

iii) Insert an item at a specified place.

iv) Delete a specified item.

AVL-Trees

An AVL tree (named after its two inventors Adel'son-Vel'skii and Landis) is a self-balancing binary search tree where the height of every node and that of its sibling differ by at most 1. AVL trees are ***height-balanced***. Height can be calculated recursively; however, it is generally faster for each node to keep track of its subtree height during insertions and deletions. While this is also done recursively, only the subtrees with a direct path from a new node to root need to be involved.




```
public class AVLTree<Key extends
Comparable<Key>, Value>  {
```

```
public class Node<Key, Value> {
    Key key;
    Value value;
    int height;
    Node<Key, Value> left;
    Node<Key, Value> right;

    Node(Key key, Value value) {
        this.key = key;
        this.value = value;
        this.height = 0;
        this.left = null;
        this.right = null;
    }
}

private Node<Key, Value> root;
private final Comparator<Key> comparator;
```

```java
public AVLTree() {
    root = null;
    comparator = new Comparator<Key>() {
        @Override
        public int compare(Key o1, Key o2) {
            return o1.compareTo(o2);
        }
    };
}

public AVLTree(Comparator<Key> keyComparator) {
    root = null;
    comparator = keyComparator;
}

public int size() { return size(this.root); }

private int size(Node<Key, Value> node) {
    return node == null ? 0 : 1 + size(node.left) + size(node.right);
}
```

Now we can instantiate an AVL tree as follows:

```
AVLTree<Integer, String> T = new AVLTree<>();
```

Before we begin to fill our AVL tree, let's recognize that we already have a method for searching, because it is just BST search (with generic keys).

Search

```
/**
 * <p>Non-recursive implementation of search for a BST (AVL Tree)</p>
 */
public Value search(Key key) throws FindException {
    Node<Key, Value> node = root;
    while (node != null) {
        int compareResult = comparator.compare(key, node.key);
        if (compareResult == 0) {
            break;
        }
        node = compareResult < 0 ? node.left : node.right;
    }
    if(node != null)
        return node.value;
    else
        throw new FindException("Key not found.");
}
```

Insert

A node n of a tree T is called ***balanced*** if the absolute value of the difference between the heights of its children is at most 1. Otherwise, it is called unbalanced.

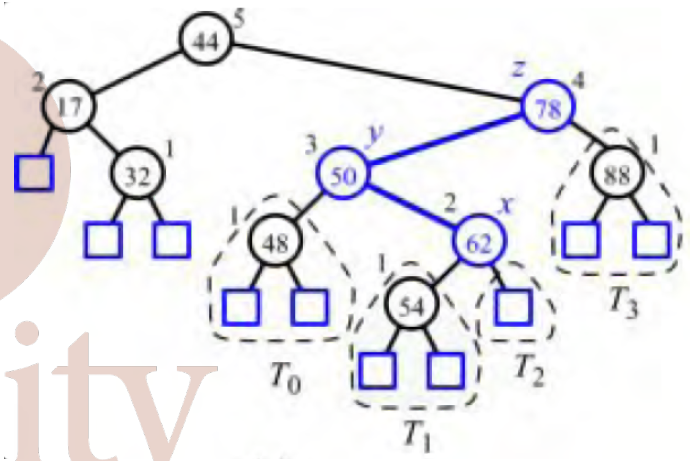
An insertion in an AVL tree T begins as an **INSERT** operation for a standard binary search tree. This insertion may violate the height-balance property since the height of some subtrees may have increased by 1. Those subtrees are exactly the trees that are on the path from the new node to root.

```java
public void insert(Key key, Value value) {
    root = insert(root, key, value);
}

private Node<Key, Value> insert(Node<Key, Value> node, Key key, Value value) {
    if (node == null) {
        return new Node<Key, Value>(key, value);
    } else {
        int compareResult = comparator.compare(key, node.key);
        if (compareResult < 0) {
            node.left = insert(node.left, key, value);
        } else if (compareResult > 0) {
            node.right = insert(node.right, key, value);
        } else {
            // duplicate key
            // may throw runtime exception or quietly overwrite value
            node.value = value;
        }
    }
    return rebalance(node);
}
```

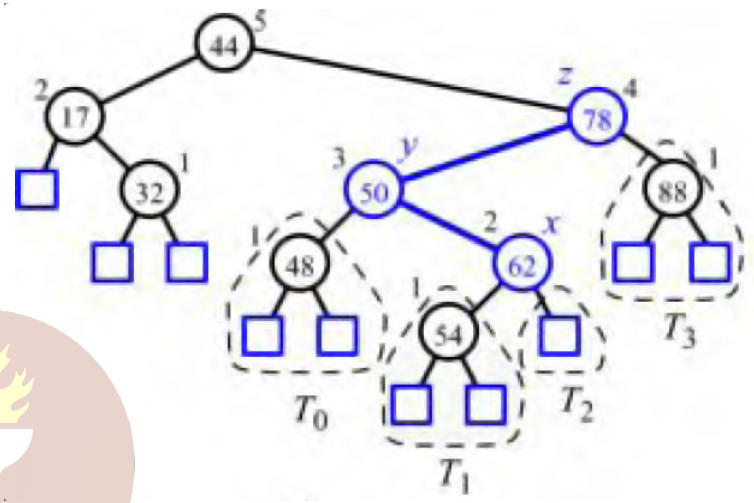
Rebalancing an AVL Tree

We restore the balance of the nodes in the BST T using a search-and-repair strategy. Numbers next to nodes in subsequent pictures are the subtree heights + 1 (counting null children).



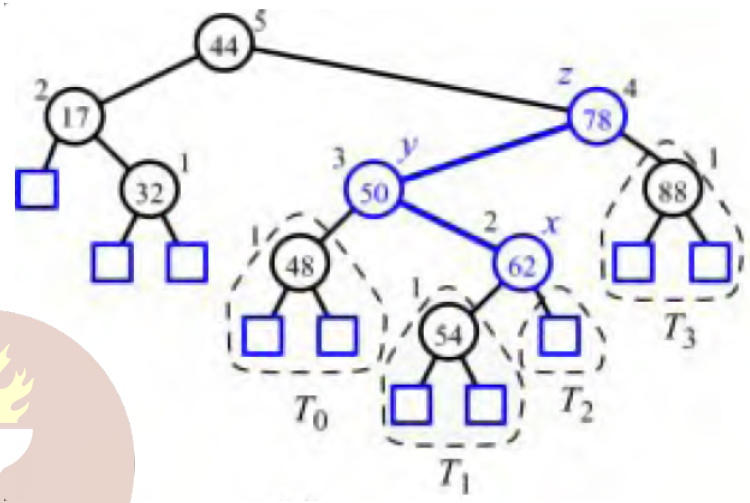
Let z be the first node encountered going up from newly inserted node n (key 54) toward the root of T such that z is unbalanced. Let y denote the child of z with higher height (note that y must be an ancestor of n because n added height to an imbalance of **2**). Let x be the child of y with higher height (there cannot be a tie and x must be an ancestor of n, possibly itself, again because of path length from n to z of at least **2**).

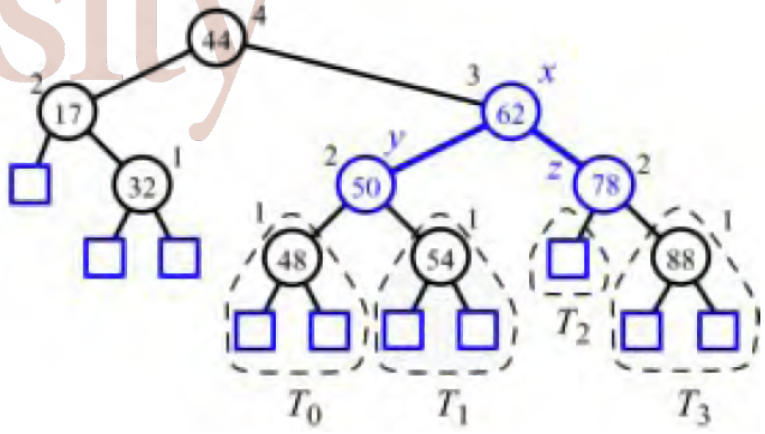
The repair is done with a tree rotation. We need an auxiliary function that checks for imbalance:




```
Balance(node z)
   return height(z.right) - height(z.left)
```

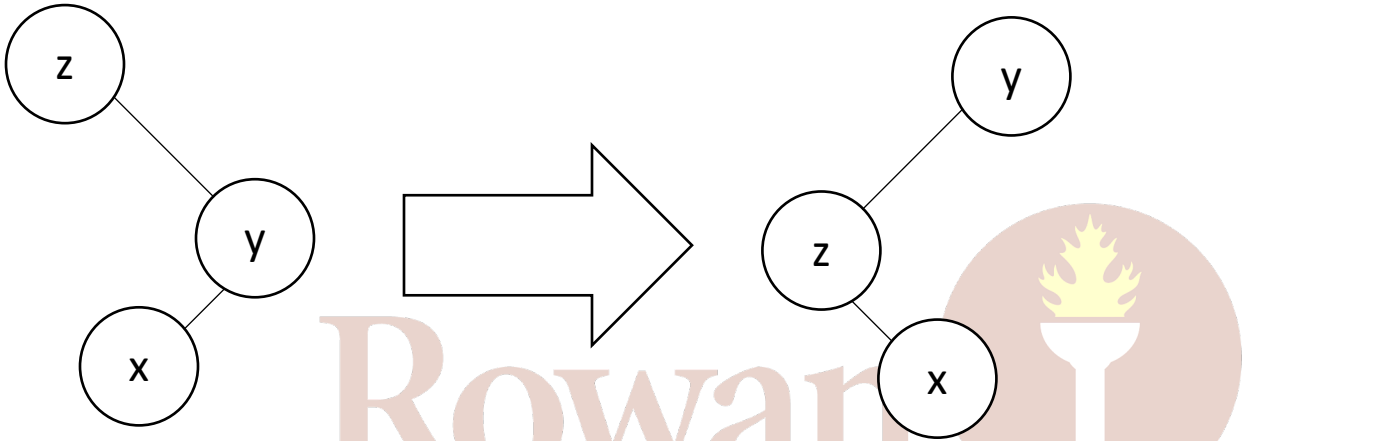
```
Rebalance(node z)
  if Balance(z)==2
    y = z.right
    if Balance(y)==2
      x = y.right
      z = rotateLeft(z)
    else
      x = y.left
      y = rotateRight(y)
      z = rotateLeft(z)
  else
    y = z.left
    if Balance(y)==-2
      x = y.left
      z = rotateRight(z)
    else
      x = y.right
      y = rotateLeft(y)
      z = rotateRight(z)
```



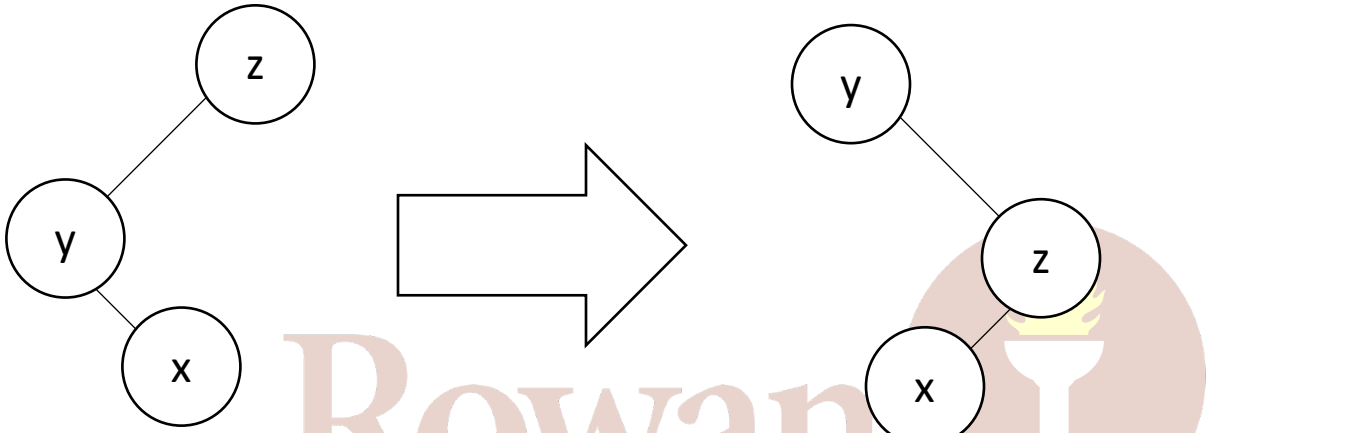


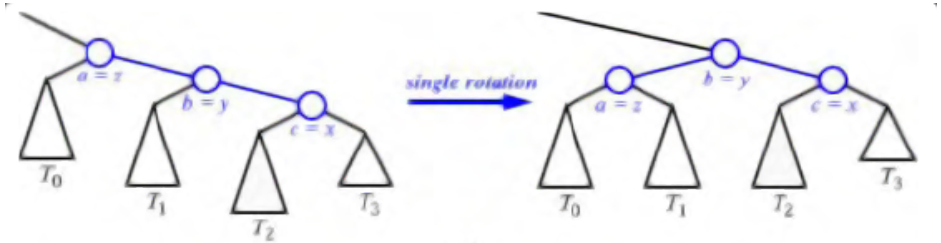
```
private Node<Key, Value> rebalance(Node<Key, Value> z) {
    updateHeight(z);
    int balance = getBalance(z);
    if (balance > 1) {
        if (height(z.right.right) > height(z.right.left)) {
            z = rotateLeft(z);
        } else {
            z.right = rotateRight(z.right);
            z = rotateLeft(z);
        }
    } else if (balance < -1) {
        if (height(z.left.left) > height(z.left.right)) {
            z = rotateRight(z);
        } else {
            z.left = rotateLeft(z.left);
            z = rotateRight(z);
        }
    }
    return z;
}
```

RotateLeft

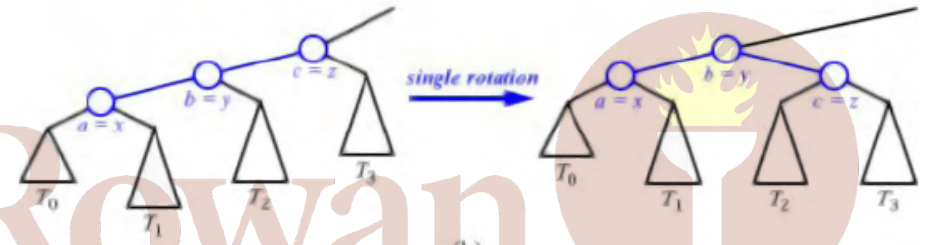


RotateRight

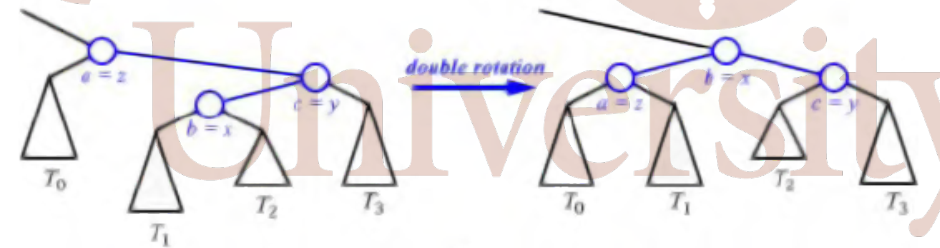




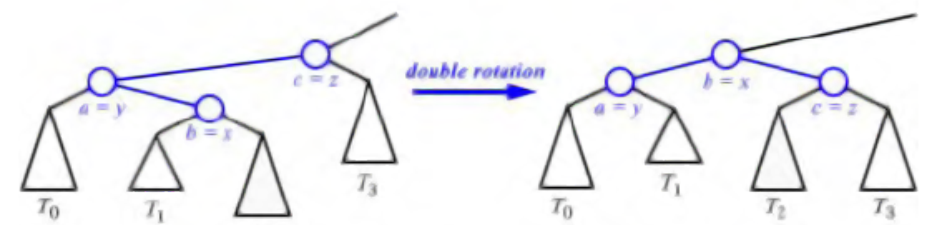
(a)



(b)

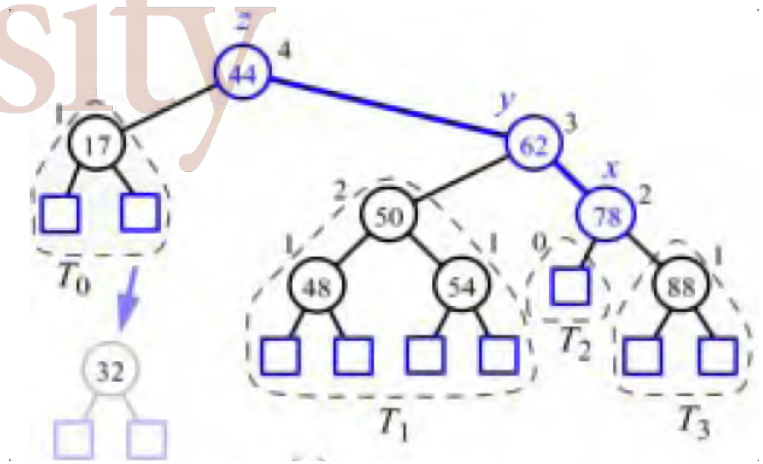


(c)

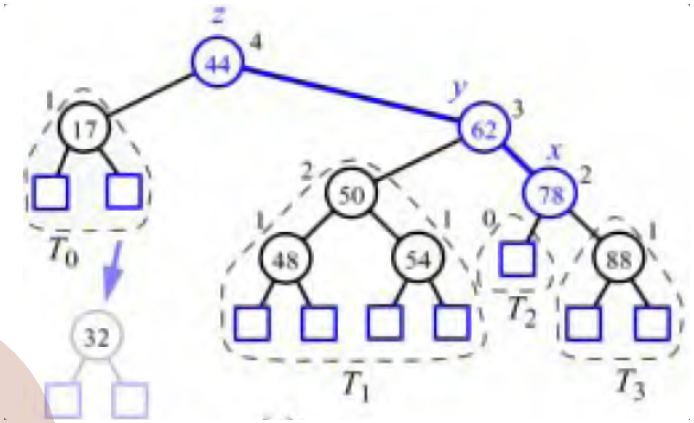


Delete

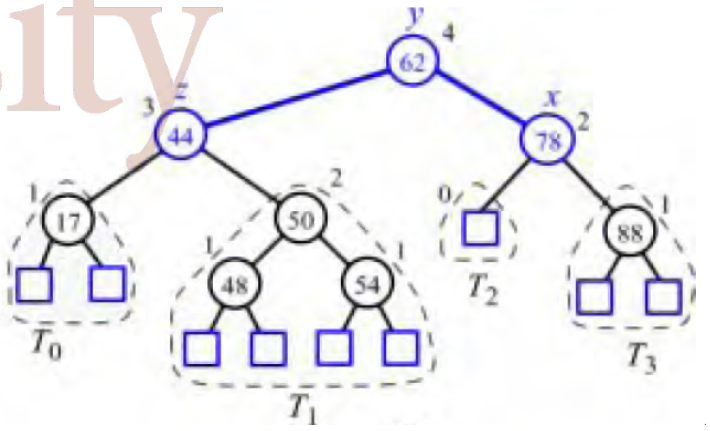
Removal of an element in an AVL tree also starts with the standard BST **DELETE**. Again, it may violate the height-balance property. After removing an internal node and elevating one of its children into its place, there may be an unbalanced node in T on the path from the parent node of the previously removed node to the root of T. However, there can be at most one such unbalanced node.



Rebalancing



Perform a **`Rebalance(x)`** operation. This restores the height-balance property locally at z. Rebalancing locally may reduce the height of the subtree, which may cause an ancestor to become unbalanced. Hence, after rebalancing z, continue to walk up T looking for unbalanced nodes until there are none or root has been rebalanced. In worst case, $O(\log(n))$ rebalancing operations are needed to restore balance globally. Per node a rotation takes $O(1)$ time.




```
public void delete(Key key) {
    root = delete(root, key);
}
```

```
private Node<Key, Value> delete(Node<Key, Value> node, Key key) {
    if (node == null) {
        return node;
    } else {
        int compareResult = comparator.compare(key, node.key);
        if (compareResult < 0) {
            node.left = delete(node.left, key);
        } else if (compareResult > 0) {
            node.right = delete(node.right, key);
        } else {
            if (node.left == null || node.right == null) {
                node = (node.left == null) ? node.right : node.left;
            } else {
                Node<Key, Value> leftMostChild = leftMostChild(node.right);
                node.key = leftMostChild.key;
                node.right = delete(node.right, node.key);
            }
        }
    }
    if (node != null) {
        node = rebalance(node);
    }
    return node;
}
```

Properties of AVL Trees

- Find/Search/Get in an AVL tree is $O(\log(n))$.
- For every internal node of an AVL tree, the heights of the two children subtree differ by at most 1.
- A subtree of an AVL tree is itself an AVL tree.
- The height of an AVL tree storing n entries is $O(\log(n))$.
  - Can be proven with a recursion formula bottom up.