

Lecture 6: Binary Search Trees (BST) and Red-Black Trees (RBTs)

Instructor: Saravanan Thirumuruganathan

- ① Data Structures for representing Dynamic Sets
 - Binary Search Trees (BSTs)
 - Balanced Search Trees
 - Balanced Binary Trees - Red Black Trees (RBTs)

- **URL:** `http://m.socrative.com/`
- **Room Name:** **4f2bb99e**

Key Things to Know for Data Structures

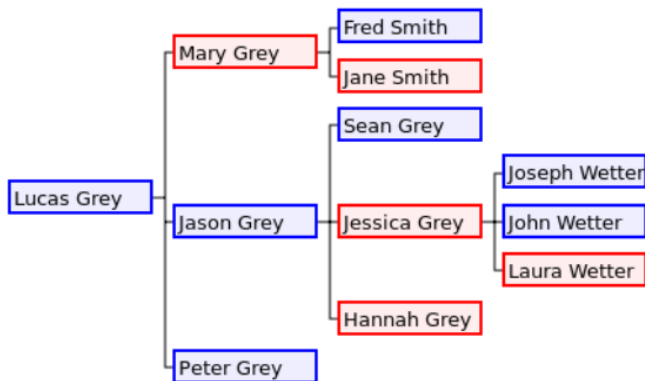
- Motivation
- Distinctive Property
- Major operations
- Key Helper Routines
- Representation
- Algorithms for major operations
- Applications

Non-Linear Data Structures:

- Very common and useful category of data structures
- Most popular one is **hierarchical**

Trees - Applications¹

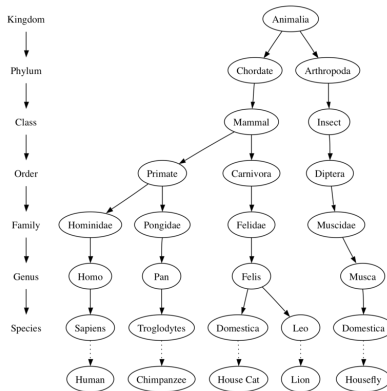
Family Tree:



¹<http://interactivepython.org/runestone/static/pythonds/Trees/trees.html>

Trees - Applications²

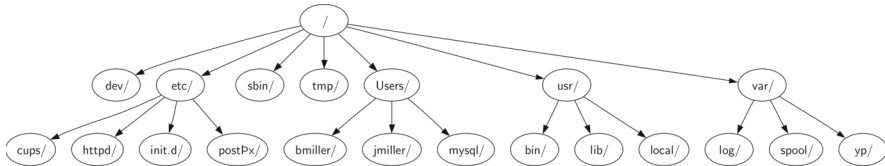
Taxonomy Tree:



²<http://interactivepython.org/runestone/static/pythonds/Trees/trees.html>

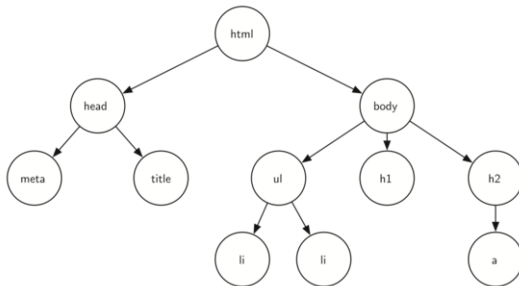
Trees - Applications³

Directory Tree:



³<http://interactivepython.org/runestone/static/pythonds/Trees/trees.html>

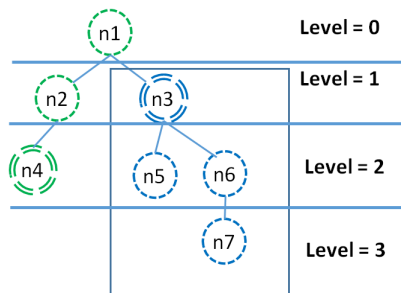
HTML DOM (Parse) Tree:



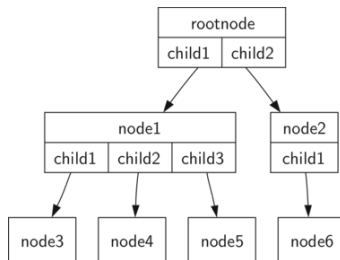
⁴<http://interactivepython.org/runestone/static/pythonds/Trees/trees.html>

Tree - Terminology

- Node
- Edge
- Root
- Children
- Parent
- Sibling
- Subtree
- Leaf/External node
- Internal node
- Level (node)
- Height (tree)
- Arity



Tree - Abstract Representation⁵



⁵<http://interactivepython.org/runestone/static/pythonds/Trees/trees.html>

Motivation

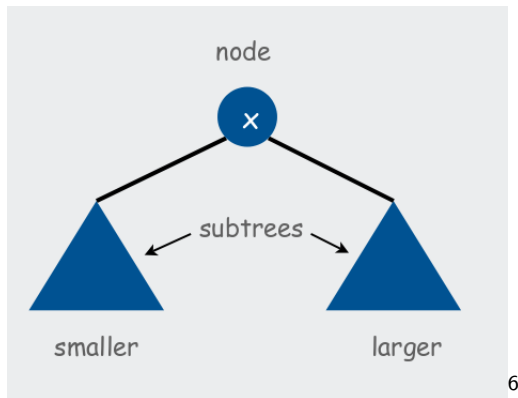
- Store dynamic set efficiently
- Use good ideas from ordered list (OL) and ordered doubly linked list (ODLL)
- Use hierarchical storage to avoid pitfalls of OL and ODLL
- First attempt at hierarchical data structure that **tries** to implement all 7 operations efficiently

Binary Trees

- Each node has at most 2 children
- Commonly referred to as *left* and *right* child
- The descendants of *left* child constitute *left* subtree
- The descendants of *right* child constitute *right* subtree

BST Property

- For every node x , the keys of left subtree $\leq \text{key}(x)$
- For every node x , the keys of right subtree $\geq \text{key}(x)$

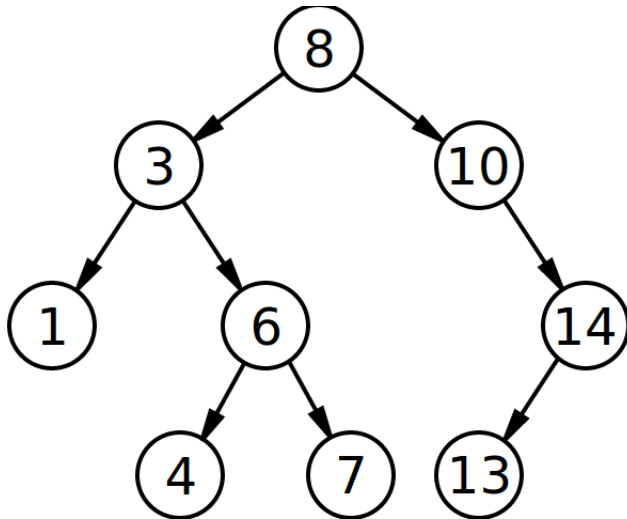


6

⁶[http:](http://www.cs.princeton.edu/~rs/AlgsDS07/08BinarySearchTrees.pdf)

[//www.cs.princeton.edu/~rs/AlgsDS07/08BinarySearchTrees.pdf](http://www.cs.princeton.edu/~rs/AlgsDS07/08BinarySearchTrees.pdf)

BST Examples⁷



⁷http://en.wikipedia.org/wiki/Binary_search_tree

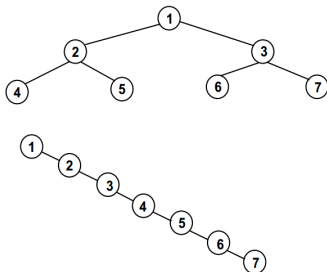
BST Height⁸

- There exists multiple possible BSTs to store same set of elements
- Minimum and Maximum Height:

⁸<https://engineering.purdue.edu/~ee608/handouts/lec10.pdf>

BST Height⁸

- There exists multiple possible BSTs to store same set of elements
- Minimum and Maximum Height: $\lg n$ and n
- Best and worst case analysis (or specify analysis wrt height)

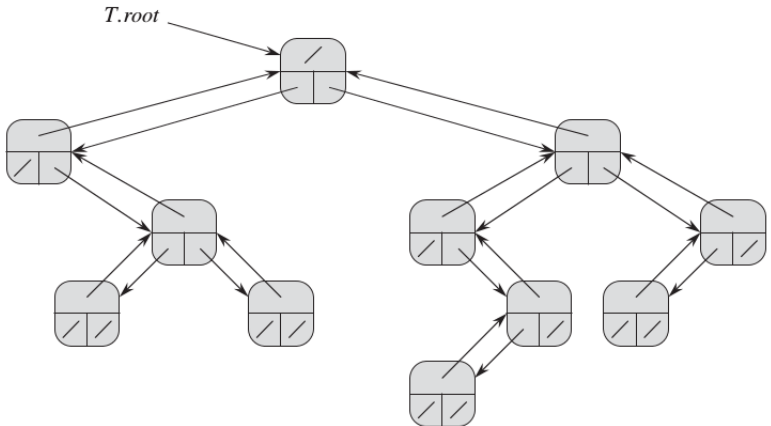


⁸<https://engineering.purdue.edu/~ee608/handouts/lec10.pdf>

Representation - I

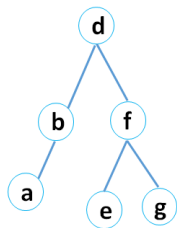
- **key:** Stores key information that is used to compare two nodes
- **value:** Stores satellite/auxillary information
- **parent:** Pointer to parent node. $\text{parent}(\text{root}) = \text{NULL}$
- **left:** Pointer to left child if it exists. Else NULL
- **right:** Pointer to right child if it exists. Else NULL

Representation - I⁹



⁹CLRSFig10.9

Representation - II



Index	Key	Left	Right
1	d	2	3
2	b	4	NULL
3	f	5	6
4	a	NULL	NULL
5	e	NULL	NULL
6	g	NULL	NULL

Major Operations

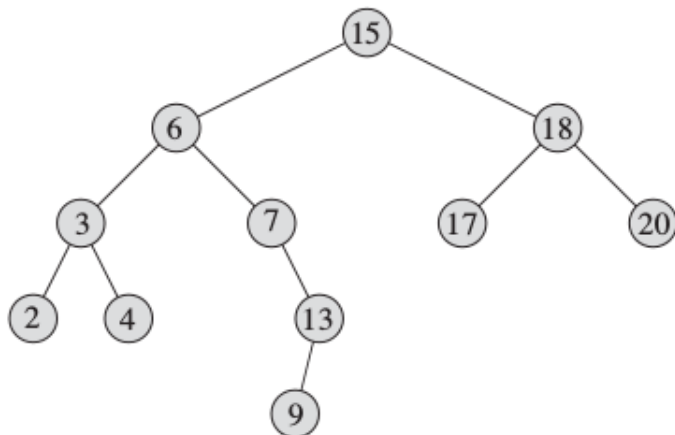
- Search
- Insert
- Minimum/Maximum
- Successor/Predecessor
- Deletion
- Traversals

Key Helper Routines

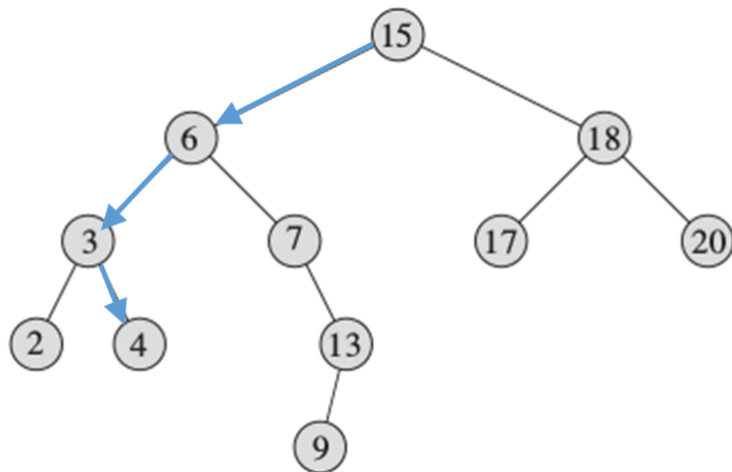
- Successor/Predecessor
- Traversals
- Case by Case Analysis:
 - Analysis by number of children : 0, 1, 2
 - Analysis by type of children: *left*, *right*

BST: Search

Search: 4



BST: Search




```
Tree-Search(x, k):  
    if x == NULL or k == x.key  
        return x  
    if k < x.key  
        return Tree-Search(x.left, k)  
    else  
        return Tree-Search(x.right, k)
```

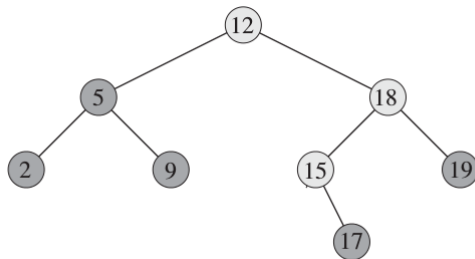
- **Analysis:**

```
Tree-Search(x, k):  
    if x == NULL or k == x.key  
        return x  
    if k < x.key  
        return Tree-Search(x.left, k)  
    else  
        return Tree-Search(x.right, k)
```

- **Analysis:** $O(h)$
- **Best Case:** $\lg n$ and **Worst Case:** $O(n)$

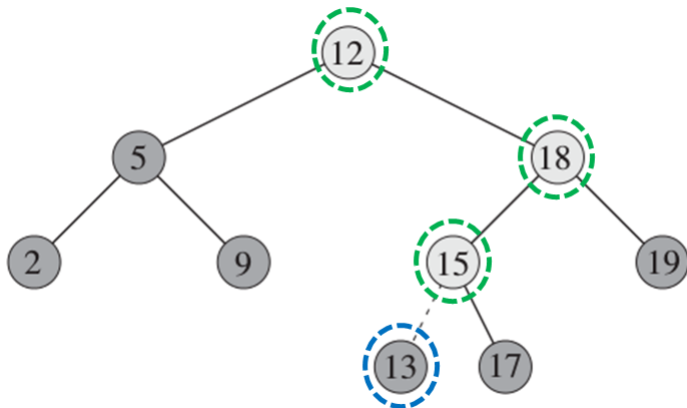
BST: Insert

Insert: 13



BST: Insert

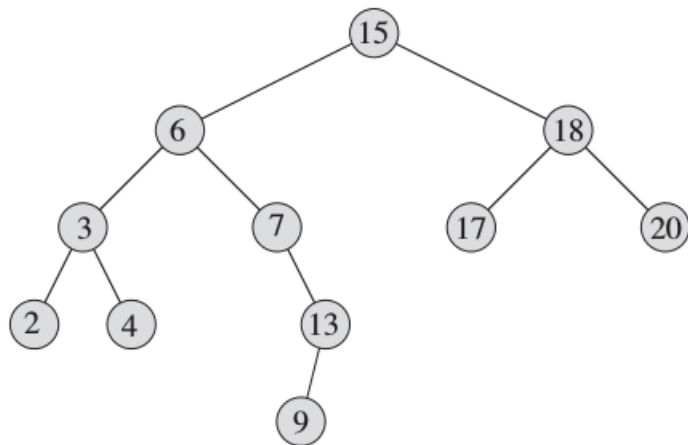
Insert: 13



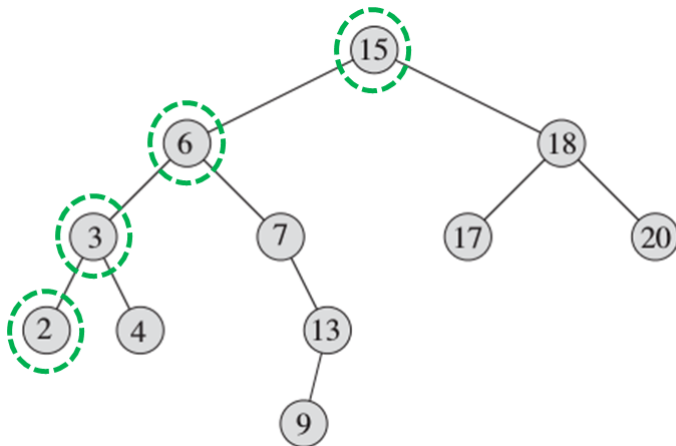
- **Analysis:**

- **Analysis:** $O(h)$
- **Best Case:** $\lg n$ and **Worst Case:** $O(n)$

BST: Minimum



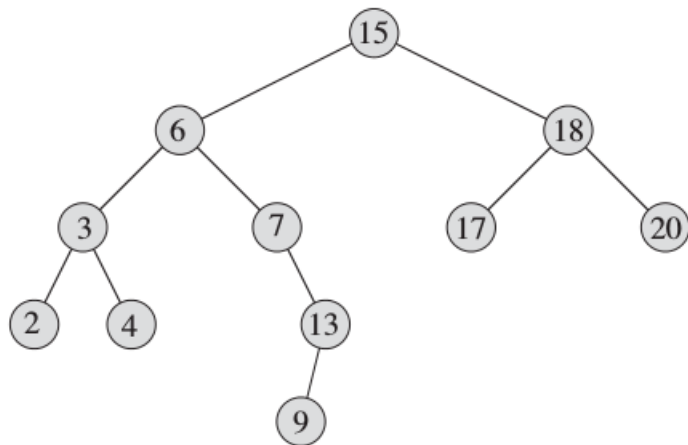
BST: Minimum



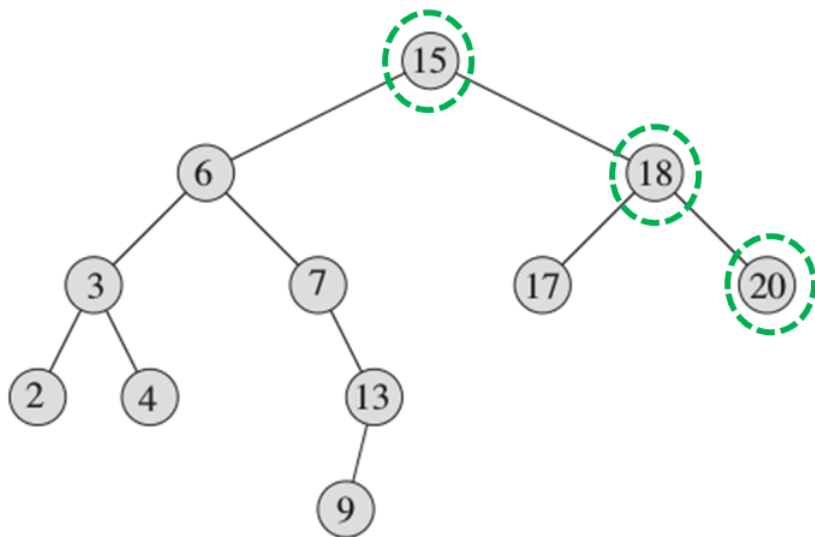

```
Tree-Minimum(x)
    while x.left is not NULL
        x = x.left
    return x
```

- **Analysis:** $O(h)$
- **Best Case:** $\lg n$ and **Worst Case:** $O(n)$

BST: Maximum



BST: Maximum

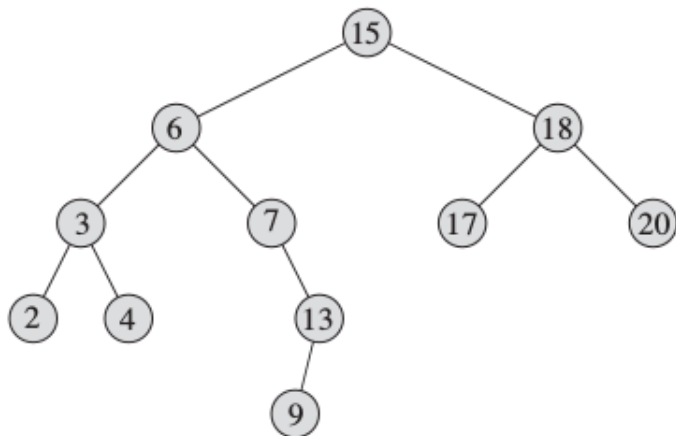


```
Tree-Maximum(x)
    while x.right is not NULL
        x = x.right
    return x
```

- **Analysis:** $O(h)$
- **Best Case:** $\lg n$ and **Worst Case:** $O(n)$

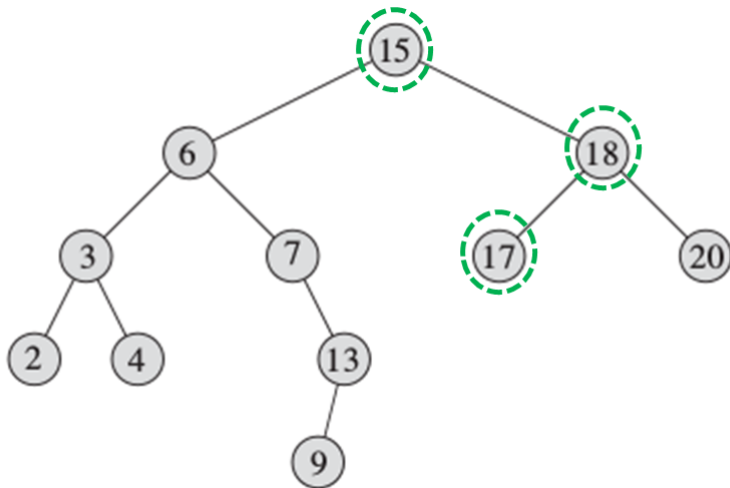
BST: Successor

Successor: 15



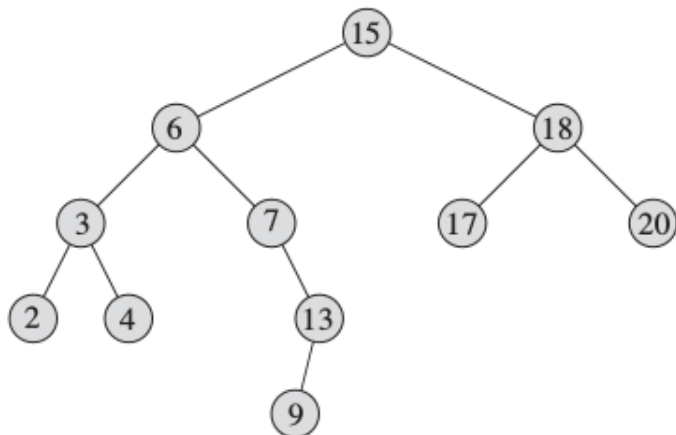
BST: Successor

Successor: 15



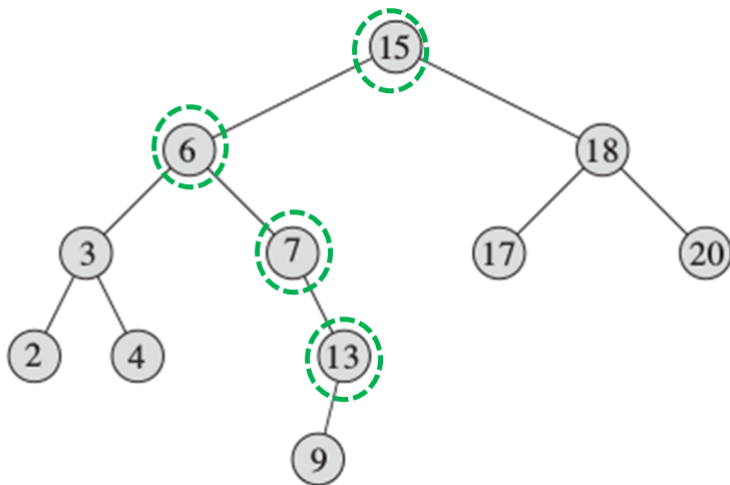
BST: Successor

Successor: 13

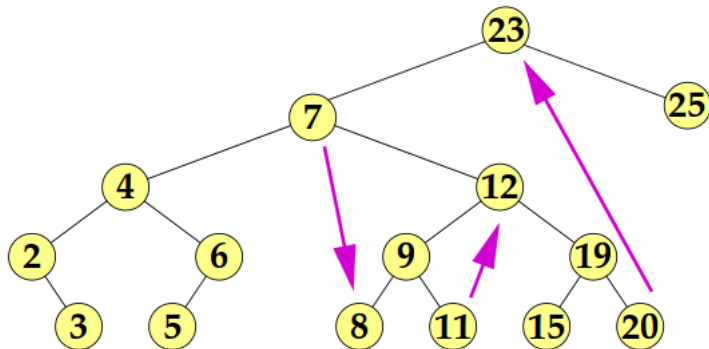


BST: Successor

Successor: 13



BST: Successor

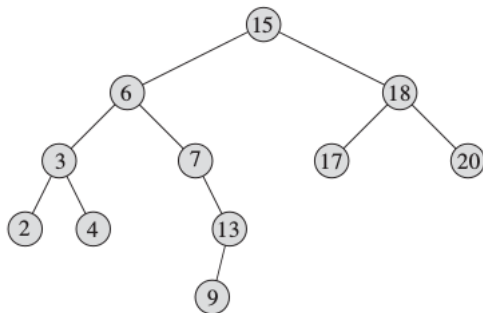


```
Tree-Successor(x):  
    if x.right is not NULL  
        return Tree-Minimum(x.right)  
    y = x.parent  
    while y is not NULL and x == y.right  
        x = y  
        y = y.parent  
    return y
```

- BST Property allowed us to find successor **without** comparing keys
- **Analysis:** $O(h)$
- **Best Case:** $\lg n$ and **Worst Case:** $O(n)$

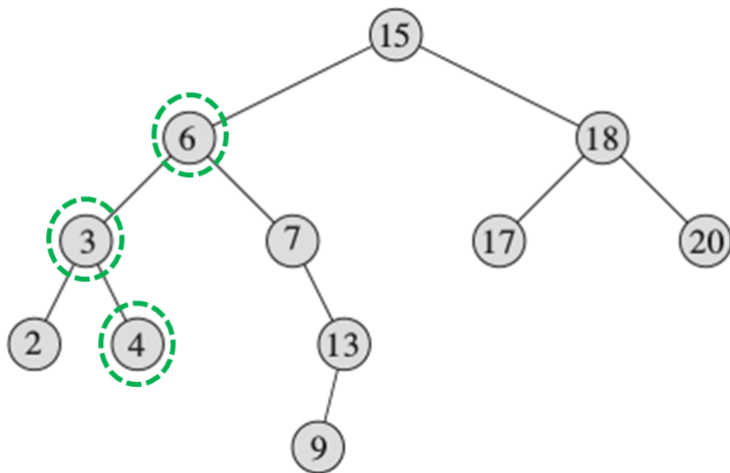
BST: Predecessor

Predecessor: 6



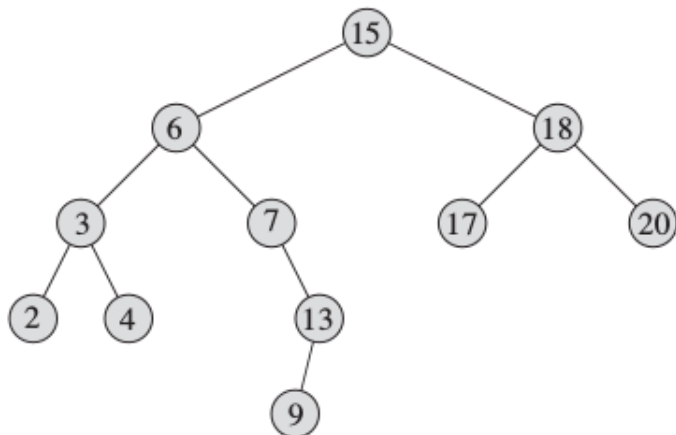
BST: Predecessor

Predecessor: 6



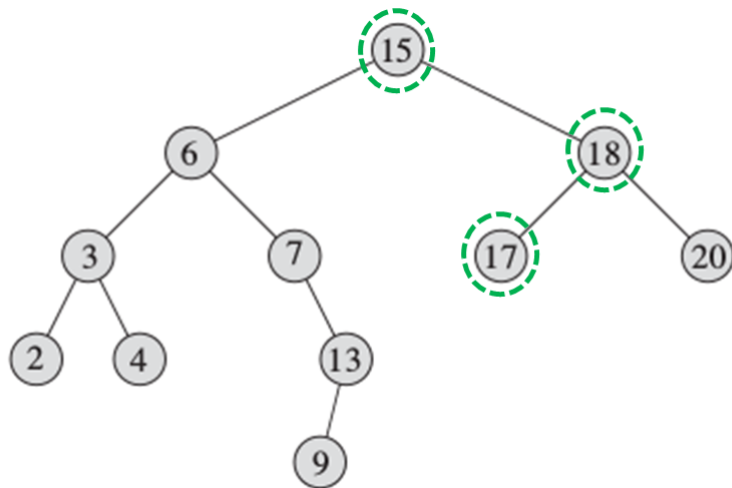
BST: Predecessor

Predecessor: 17



BST: Predecessor

Predecessor: 17



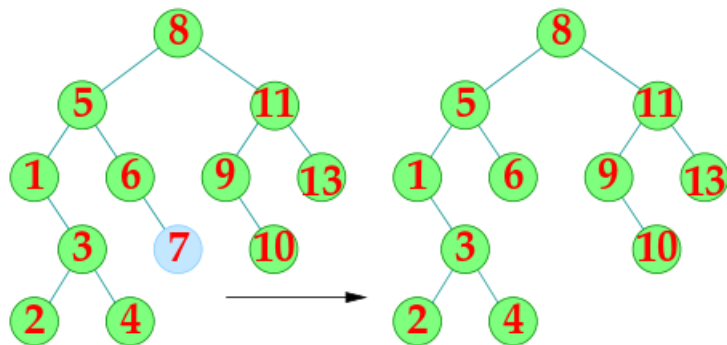
```
Tree-Predecessor(x):  
    if x.left is not NULL  
        return Tree-Maximum(x.left)  
    y = x.parent  
    while y is not NULL and x == y.left  
        x = y  
        y = y.parent  
    return y
```

- BST Property allowed us to find predecessor **without** comparing keys
- **Analysis:** $O(h)$
- **Best Case:** $\lg n$ and **Worst Case:** $O(n)$

Trickiest Operation! Suppose we want to delete node z

- ① z has no children: Replace z with NULL
- ② z has one children c : Promote c to z 's place
- ③ z has two children:
 - (a) Let z 's successor be y
 - (b) y is either a leaf or has **only** right child
 - (c) Promote y to z 's place
 - (d) Fix y 's loss via Cases 1 or 2

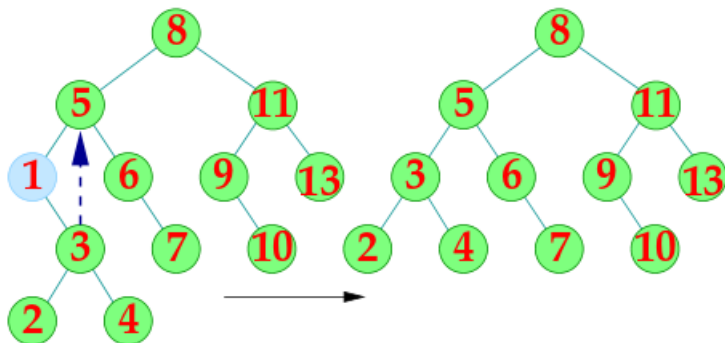
BST: Deletion Case I¹⁰



¹⁰[https:](https://www.cs.rochester.edu/u/gildea/csc282/slides/C12-bst.pdf)

[//www.cs.rochester.edu/u/gildea/csc282/slides/C12-bst.pdf](https://www.cs.rochester.edu/u/gildea/csc282/slides/C12-bst.pdf)

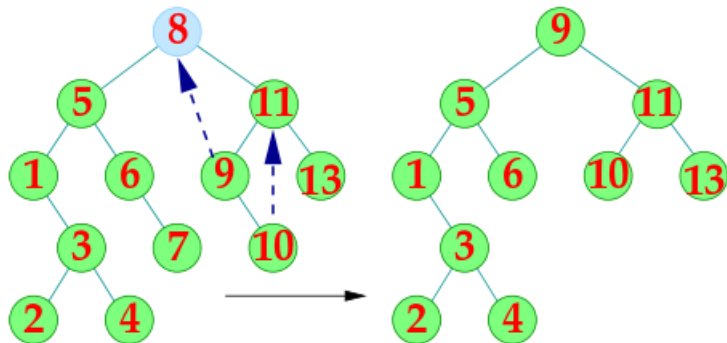
BST: Deletion Case II¹¹



¹¹[https:](https://www.cs.rochester.edu/u/gildea/csc282/slides/C12-bst.pdf)

[//www.cs.rochester.edu/u/gildea/csc282/slides/C12-bst.pdf](https://www.cs.rochester.edu/u/gildea/csc282/slides/C12-bst.pdf)

BST: Deletion Case III¹²



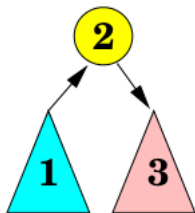
¹²[https:](https://www.cs.rochester.edu/u/gildea/csc282/slides/C12-bst.pdf)

[/www.cs.rochester.edu/u/gildea/csc282/slides/C12-bst.pdf](https://www.cs.rochester.edu/u/gildea/csc282/slides/C12-bst.pdf)

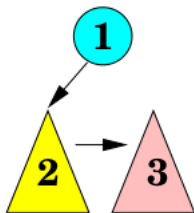
- Perfectly fine if you cannot do deletion by memory
- Things will become hairier in RBT
- As long as you remember the key ideas and operations, you will be fine

BST: Traversal

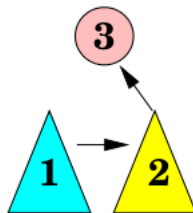
- **Traversal:** Visit all nodes in a tree
- Many possible traversal strategies
- Three are most popular: Pre-Order, In-Order, Post-Order



inorder



preorder



postorder

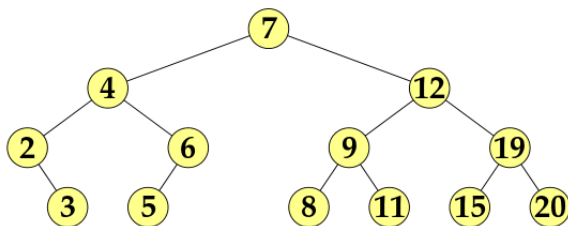
```
In-Order-Walk(x):  
    if x == NULL  
        return  
    In-Order-Walk(x.left)  
    Print x.key  
    In-Order-Walk(x.right)
```

- **Analysis:**

```
In-Order-Walk(x):  
    if x == NULL  
        return  
    In-Order-Walk(x.left)  
    Print x.key  
    In-Order-Walk(x.right)
```

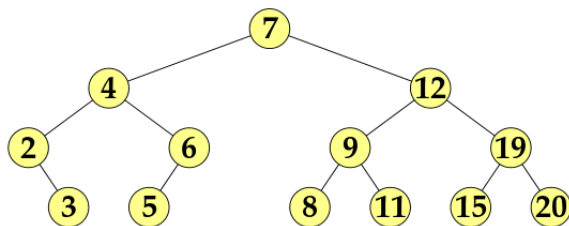
- **Analysis:** $O(n)$
- Holds true for all three traversals

BST: Traversal



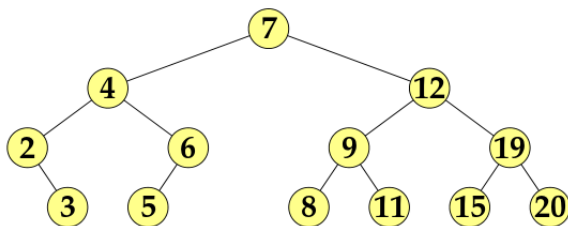
- In-Order:

BST: Traversal



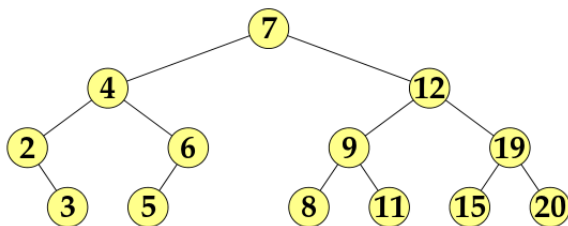
- **In-Order:** 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 19, 20.
- **Pre-Order:**

BST: Traversal



- **In-Order:** 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 19, 20.
- **Pre-Order:** 7, 4, 2, 3, 6, 5, 12, 9, 8, 11, 19, 15, 20.
- **Pre-Order:**

BST: Traversal



- **In-Order:** 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 19, 20.
- **Pre-Order:** 7, 4, 2, 3, 6, 5, 12, 9, 8, 11, 19, 15, 20.
- **Post-Order:** 3, 2, 5, 6, 4, 8, 11, 9, 15, 20, 19, 12, 7.

- Notice anything special about In-Order traversal?

- Notice anything special about In-Order traversal?
 - Returns items in sorted order!
- Successor/Predecessor can be expressed in terms on In-Order traversal

Tree-Sort:

- Construct a BST out of elements
- Do In-Order traversal

Analysis:

- **Time Complexity:** Time Complexity for Constructing Tree + Time Complexity for In-Order traversal
- **Best Case:**

Tree-Sort:

- Construct a BST out of elements
- Do In-Order traversal

Analysis:

- **Time Complexity:** Time Complexity for Constructing Tree + Time Complexity for In-Order traversal
- **Best Case:** $n \times \lg n + n = O(n \lg n)$
- **Worst Case:**

Tree-Sort:

- Construct a BST out of elements
- Do In-Order traversal

Analysis:

- **Time Complexity:** Time Complexity for Constructing Tree + Time Complexity for In-Order traversal
- **Best Case:** $n \times \lg n + n = O(n \lg n)$
- **Worst Case:** $n \times n + n = O(n^2)$

Balanced Binary Trees

- Time complexity of BST operations depends on height
- Can vary between $O(\lg n)$ to $O(n)$
- BST operations do not take any special care to keep tree balanced
- If we can do the balancing efficiently, then all operations become faster
- Self-balancing - Do not run balancing algorithms periodically

Notion of Balance

- Maintaining perfectly balanced trees is very hard and expensive
- So we resort to BSTs that are **approximately** balanced
- Need to define notion of balance
- Ideas?

Notion of Balance

- Maintaining perfectly balanced trees is very hard and expensive
- So we resort to BSTs that are **approximately** balanced
- Need to define notion of balance
- Ideas?
 - Informally, ensure the longest path in tree is not “too” long
 - Many ways of formally specifying it
 - Eg: $|\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})| \leq 1$
- When can balance be broken?

Notion of Balance

- Maintaining perfectly balanced trees is very hard and expensive
- So we resort to BSTs that are **approximately** balanced
- Need to define notion of balance
- Ideas?
 - Informally, ensure the longest path in tree is not “too” long
 - Many ways of formally specifying it
 - Eg: $|\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})| \leq 1$
- When can balance be broken? Insertion, Deletion

Balanced Search Trees

- Red-Black trees
- AVL trees
- 2 – 3 and 2 – 3 – 4 trees
- B-trees and other variants
- Treaps
- Skip trees
- Splay trees
- and many many more

Red-Black Trees

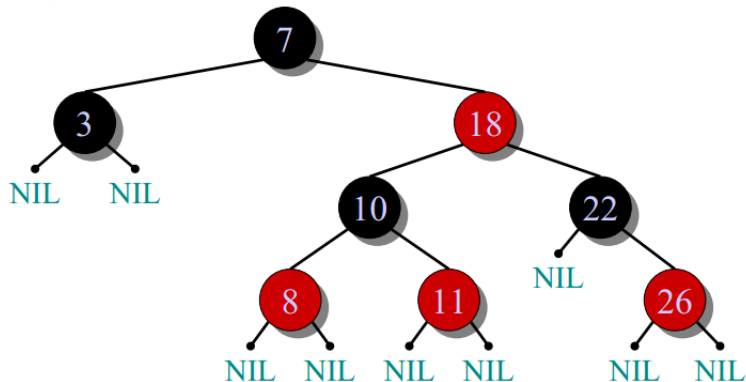
RBT: Motivation

- Most important self-balancing BST
- Invented by Guibas-Sedgewick
- Simplifies/Unifies various balanced tree algorithms
- Became popular due to its simplicity in implementation
- Stores additional information about color of node (1 bit)
- All operations are logarithmic!

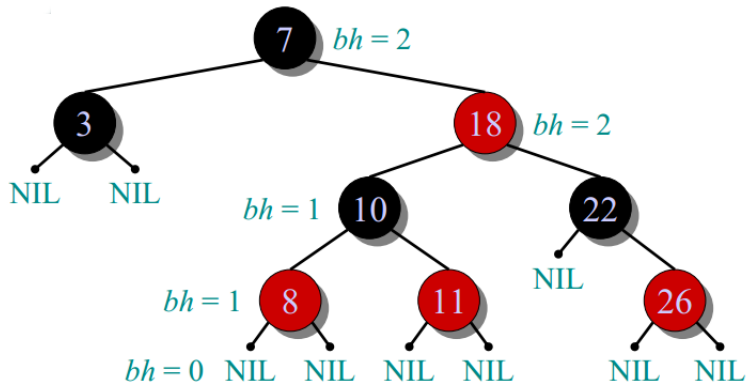
RBT Property

- Every node is either red or black
- Root and leaves are black
- If node is red, then its parent is black
- All *simple* paths from any node v to a descendent leaf have same number of black nodes. Aka *black-height*(v)

RBT Example



RBT Example



Major Operations

- Search
- Insert
- Minimum/Maximum
- Successor/Predecessor
- Deletion

Key Helper Routines

- Rotations - Right and Left
- Case by Case Analysis:
 - Analysis by number of type of children and siblings
 - Analysis by color of children

Theorem (RBT Theorem)

A red-black tree with n keys has height

$$h \leq 2 \lg(n + 1)$$

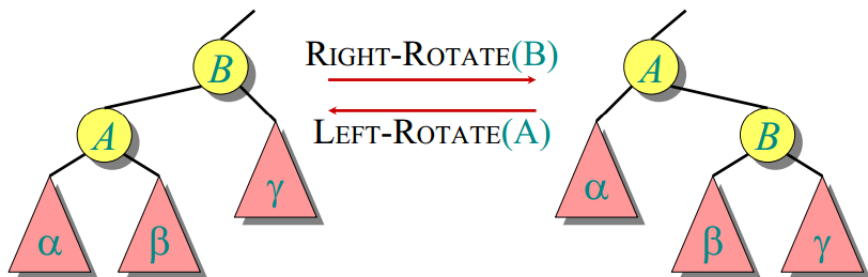
Corollary

Operations Search, Min, Max, Successor, Predecessor all run in $O(\lg n)$ time on a Red-Black tree with n nodes.

RBT - Modifying Operations

- Insert and Delete need to be more complex
- They cause “changes” to tree
 - Change of color
 - Change of tree structure via “rotations”

Rotations



- Rotation can be done in $O(1)$ time.
- Rotations maintain in-order ordering of keys:
 $a \in \alpha, b \in \beta, c \in \gamma \Rightarrow a \leq A \leq b \leq B \leq c$

Insertion in RBT

- Insert x in tree
- Color x red
- Only red-black property 3 can be violated
- Move violation up the tree by **recoloring** until it can be fixed with **rotations** and **recoloring**.

Major Concepts:

- Binary Search Trees
- Concept of Self-Balancing
- Red-Black Trees