

# Binary Search Trees

---

Dr. Anirban Ghosh

**School of Computing**  
**University of North Florida**



# Maps

## Definition

A **record** is a key-value pair:  $(k, v)$

A **map** is an abstract data type for maintaining a set of records

- No two records can have the same key
- However, two records can have same values though
- Association of keys to values define a **mapping**:  $f(\text{key}) = \text{value}$

## Examples of maps

- UNF maintains a map of (N#, student information) records
- A social media company maintains a map of (email address, user account information) records
- An assembler maintains a symbol table (a map) of (opcode, hex) records
- A text-editor maintains a map of (color, RGB representation) records

# How to implement a map?

## Common map operations

- **Insert** a record  $(k, v)$
- **Retrieve** a record having key  $k$
- **Delete** a record having key  $k$

## Approach 1: maintain a sorted list of records

- **Insertion.** will take  $O(\log n)$  time for figuring out the correct spot for the incoming record using binary search; then  $O(n)$  time for shifting items to the right to accommodate the new record; total time taken is  $O(\log n) + O(n) = O(n)$
- **Retrieval.** will take  $O(\log n)$  time using binary search
- **Deletion.** will take  $O(\log n)$  time to locate it using a binary search; then then  $O(n)$  time for left shifting items to fill the empty spot; total time taken  $O(\log n) + O(n) = O(n)$

# How to implement a map?

## Common map operations

- **Insert** a record  $(k, v)$
- **Retrieve** a record having key  $k$
- **Delete** a record having key  $k$

## Approach 2: maintain an unsorted list of records

- **Insertion.** will take  $O(1)$  time (add the new record at the end)
- **Retrieval.** will take  $O(n)$  time using a linear search; may need to search the whole list in the worst-case
- **Deletion.** will take  $O(n)$  time to locate it using a linear search; then  $O(n)$  time for left shifting items to fill the empty spot; total time taken  $O(n) + O(n) = O(n)$

## Common map operations

- **Insert** a record  $(k, v)$
- **Retrieve** a record having key  $k$
- **Delete** a record having key  $k$

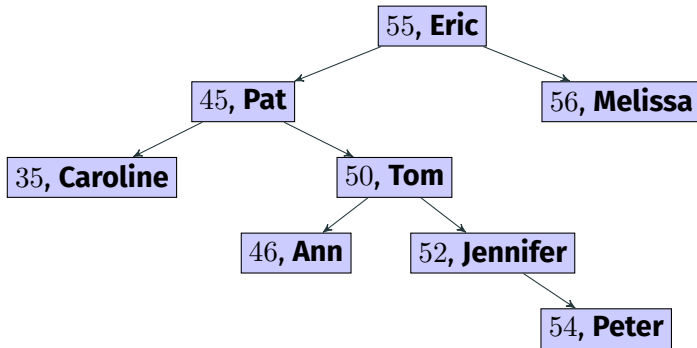
- ☞ To accomplish the above three tasks in  $O(\log n)$  time each
- ☞ **Balanced** binary search trees is the solution; stay tuned ...

# The map ADT

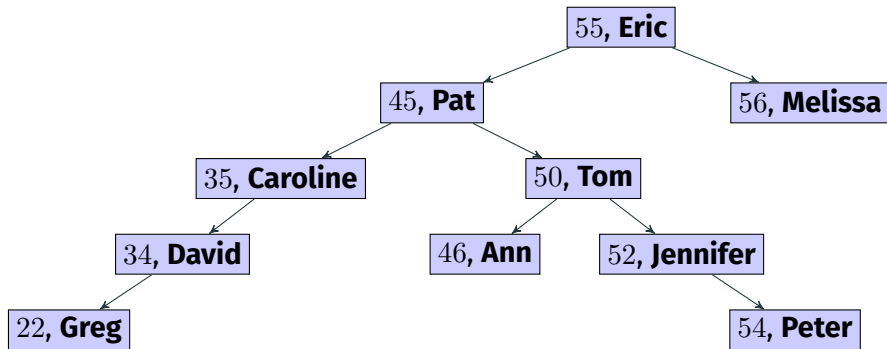
```
public interface MapADT<K,V> {  
    boolean put(K key, V value); // adds a new record with key key and value value  
    V remove(K key); // removes the record having key key  
    V get(K key); // return the value part of the record whose key is key  
    V updateValue(K key, V newValue); // updates the value part of the record whose key is key with a new value  
    int size(); // returns the number of records stored in the map  
    void clear(); //Removes all records from the map  
}
```

# What is a Binary Search Tree?

- It is a **binary tree** where every node contains a **<key, value>** pair (a **record**); keys must be **comparable** but the values don't need to be
- Moreover, for every node  $p$  in the tree, the following 2 properties hold
  - 1 Keys stored in the left subtree of  $p$  are  $<$  the key stored at  $p$
  - 2 Keys stored in the right subtree of  $p$  are  $>$  the key stored at  $p$



# What is a Binary Search Tree?



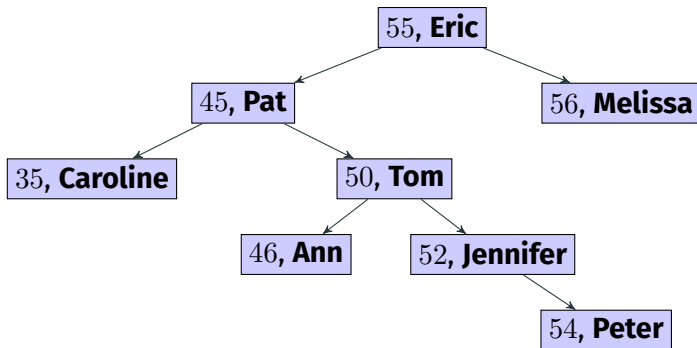
## Use

BSTs can be used to implement **maps** and are commonly used for fast searching (typically need far less comparisons than lists)



## An important property of BSTs

An inorder traversal of a BST always gives the sorted sequence based on the keys



### Inorder traversal

35, **Caroline**; 45, **Pat**; 46, **Ann**; 50, **Tom**; 52, **Jennifer**; 54, **Peter**; 55, **Eric**; 56, **Melissa**

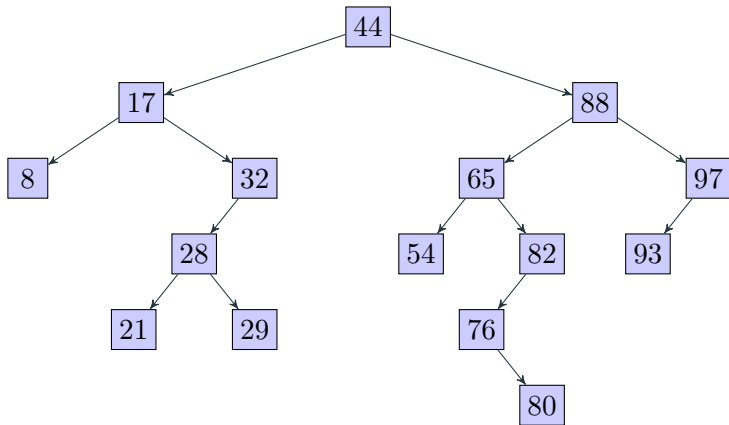
Let's say you need to look for the record that has the key  $k$ ; how will you do this?

### Algorithm

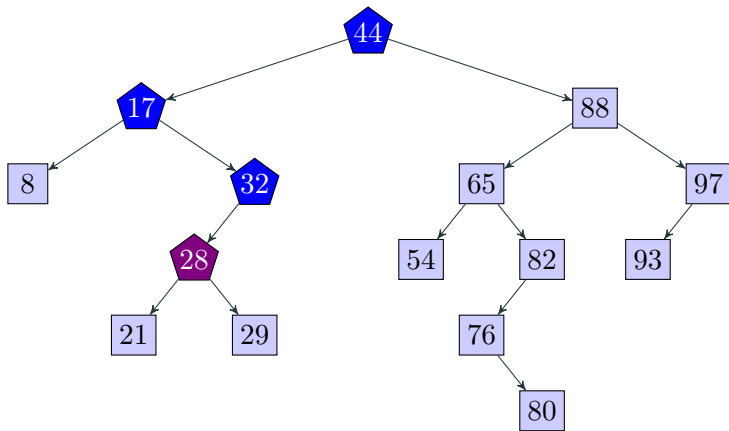
- Start at the root
- If the root's key is  $k$ , then search is successful
- If  $k < \text{root's key}$ , search recursively (or iteratively) in the left subtree of the root
- Otherwise, search recursively (or iteratively) in the right subtree of the root
- If we have reached a null link, no record exists in the tree with key  $k$

To save space in the figures, we will write only the record keys inside the nodes and avoid the corresponding values

## Example

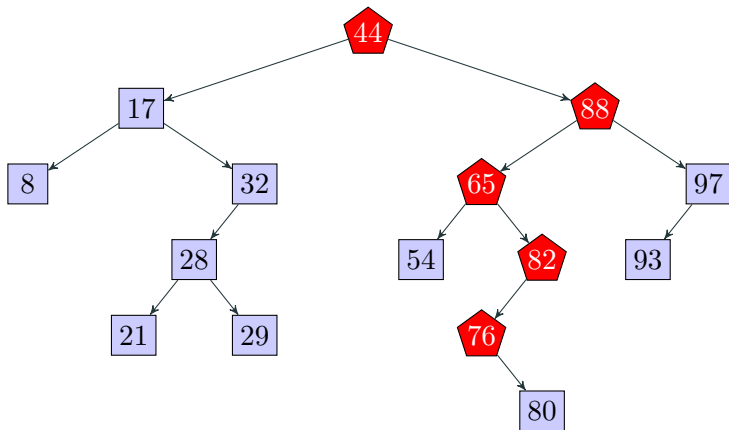


## Example: search for 28



Found!

## Example: search for 68



Not found; a null link is reached (the left link of 76)

$O(h)$ , where  $h$  is the height of the BST under consideration, since, for searching, we need to traverse a path whose length is  $h$  in the worst-case

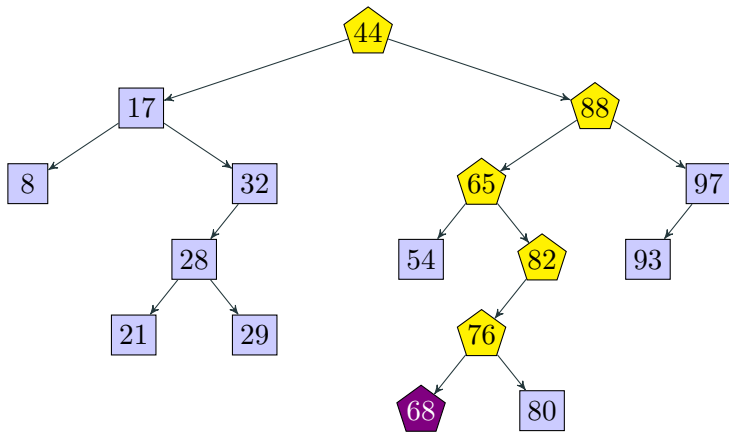
How to insert a record into a BST having key  $k$ ?

### Algorithm

- Start at the root
- If the root is `null`, replace empty tree with a new tree with the new record as the root, and signal **SUCCESS**
- If  $k$  equals root's key, signal **FAILURE** since a record with key  $k$  already exists
- If  $k <$  root's key, insert recursively (or, iteratively) in the left subtree of the root
- Otherwise, insert recursively (or, iteratively) in the right subtree of the root



## Example: insert 68



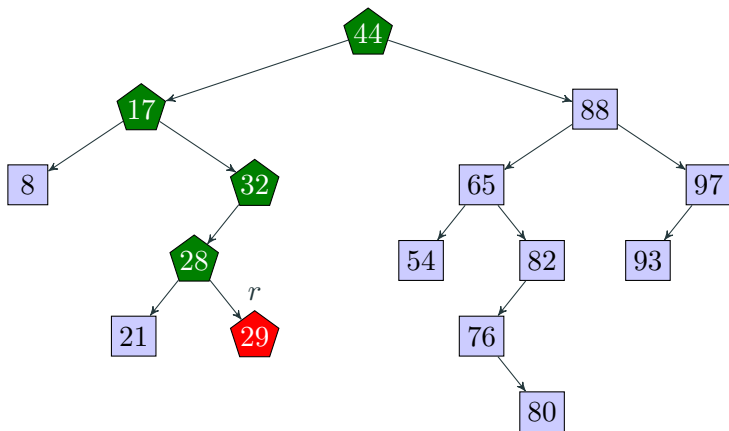
$O(h)$ , where  $h$  is the height of the BST under consideration, since, for inserting a new record, we need to traverse a path whose length is  $h$  in the worst-case

### How to delete the record from a BST having key $k$ ?

#### Idea

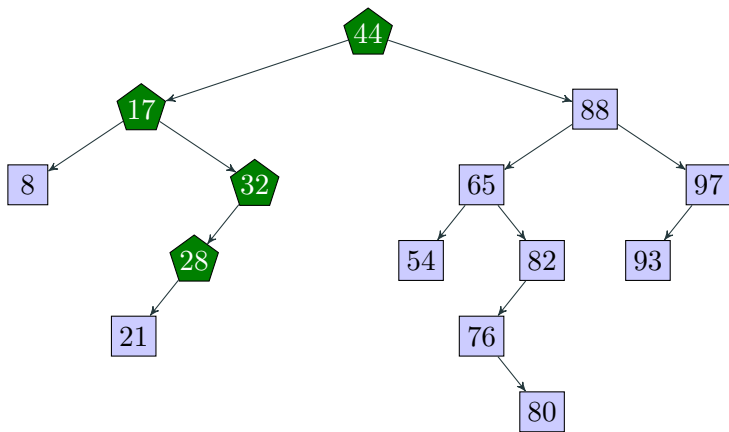
- Traverse through the tree to find the record having key  $k$
- If the record cannot be found, no action is needed
- Now assume that we have found the record that has key  $k$  at node  $r$ 
  - 1  $r$  is a leaf node (no child)
  - 2  $r$  has exactly one child (either left or right)
  - 3  $r$  has two children (both left and right)

## Case 1: $r$ is a leaf node, delete 29

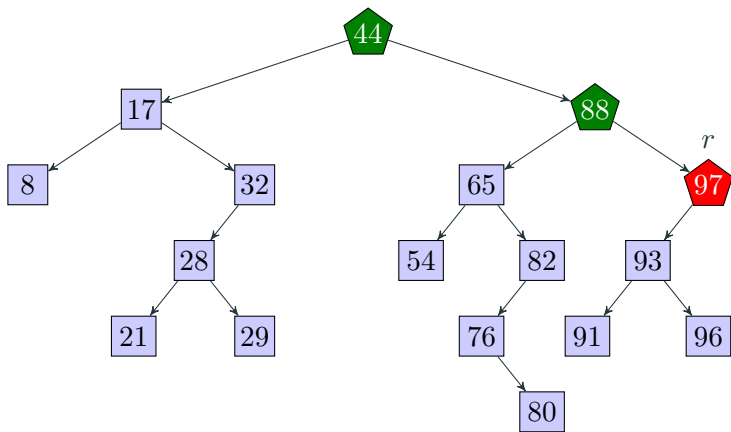


Easy! just delete it right away

## Case 1: $r$ is a leaf node, delete 29

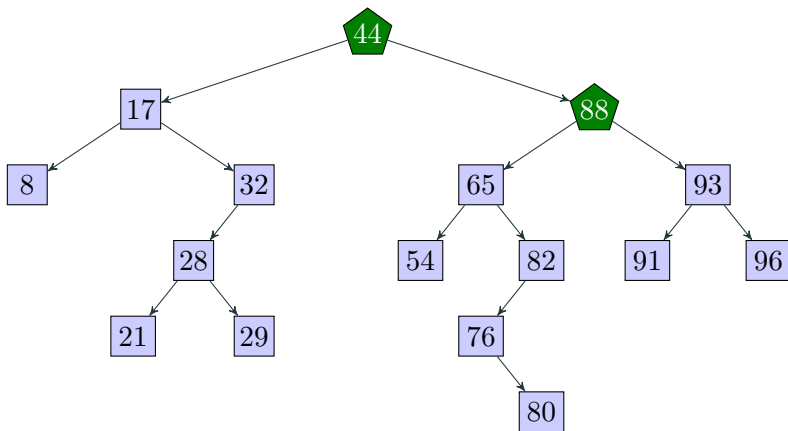


## Case 2: $r$ has one child, delete 97

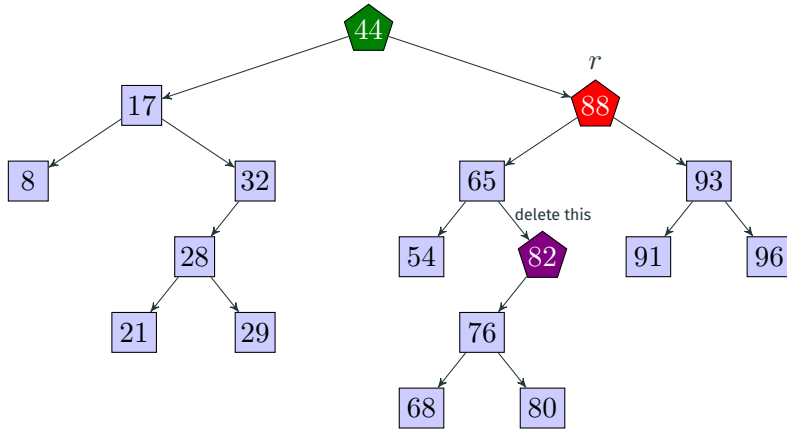


Make the parent of  $r$  point to the only child of  $r$  instead of  $r$

## Case 2: $r$ has one child, delete 97



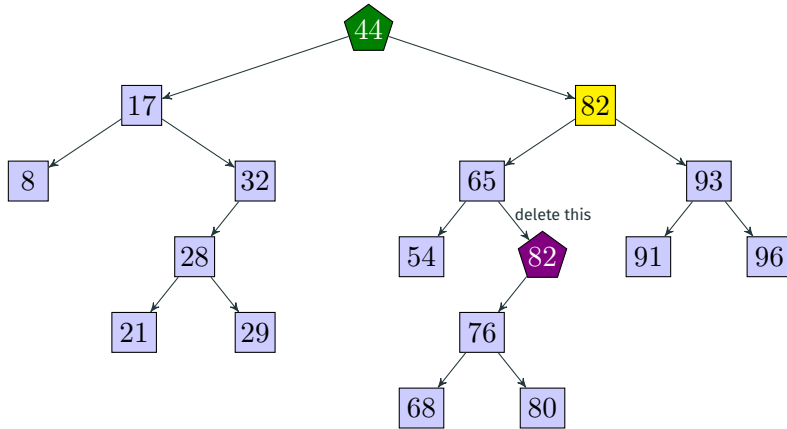
### Case 3: $r$ has two children, delete 88



Replace the record at  $r$  with the record at the inorder predecessor  $p$  of  $n$ ; then delete  $p$  using Case 1 or 2, depending on the number of children of  $p$ . Note that the inorder predecessor is either a leaf node or has a left child only (no right child).

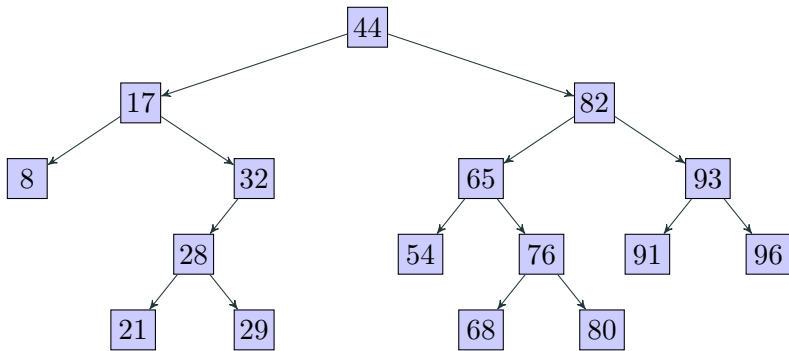


### Case 3: $r$ has two children, delete 88



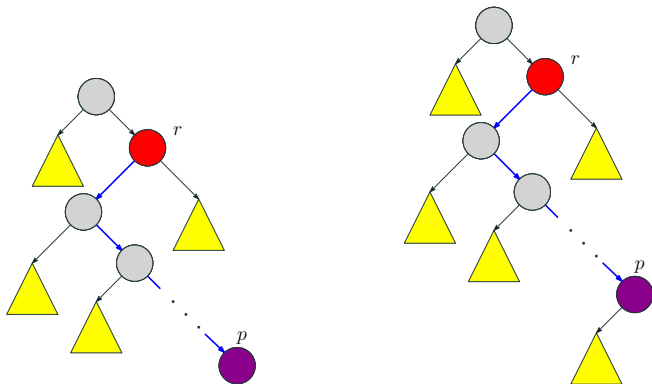
Replace the record at  $r$  with the record at the inorder predecessor  $p$  of  $n$ ; then delete  $p$  using Case 1 or 2, depending on the number of children of  $p$ . Note that the inorder predecessor is either a leaf node or has a left child only (no right child).

### Case 3: $r$ has two children, delete 88



Replace the record at  $r$  with the record at the inorder predecessor  $p$  of  $n$ ; then delete  $p$  using Case 1 or 2, depending on the number of children of  $p$ . Note that the inorder predecessor is either a leaf node or has a left child only (no right child).

## Finding the inorder predecessor in Case 3



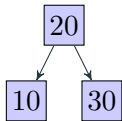
The inorder predecessor  $p$  of node  $r$  is either a leaf node (left figure) or has a left child but no right child (right figure). So, to find  $p$ , go left of  $r$  and then keep on moving right as long as possible. It takes  $O(h)$  time to find  $p$ , where  $h$  is the height of the tree. The yellow triangles represent subtrees (some of them could be empty).

## Time complexity of deletion

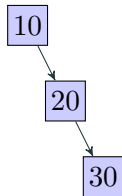
- It takes  $O(h)$  time for locating the record to be deleted, where  $h$  is the height of the BST
- Case 1 takes  $O(1)$  time to delete a record
- Case 2 takes  $O(1)$  time to delete a record
- Case 3 takes  $O(h)$  time to delete a record since we need to locate the inorder predecessor  $p$  of node  $r$  in  $O(h)$  time and then delete  $p$  in  $O(1)$  time using Case 1 or 2
- So, a single deletion takes  $O(h)$  time in the worst-case

See the class `TreeMapBST`

## BSTs are not unique



Insertion sequence: 20, 10, 30



Insertion sequence: 10, 20, 30



***Their structures really depend  
on the insertion sequence of records***

## An application

### Problem

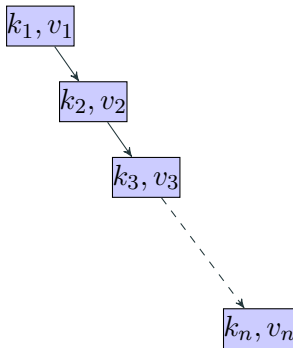
Given a text, find out the unique words in it along with their counts. We also need to output the distinct words with their count.

### A solution

- Use a BST  $T$  where the key-type is `String` and the value-type is `Integer`. This means every node will store a word from the text and its count in the same text.
- For every word  $w$  in the text, first check if a node exists in  $T$ , where the stored key is  $w$ 
  - If such a node does not exist, insert a new node in  $T$  with key  $w$  and value 1
  - If such a node exists in  $T$ , increment the stored value (essentially a counter) by 1

See the class [WordCounter](#)

## Worst case scenario: skewed binary trees



In this case,  $k_1 < k_2 < k_3 < \dots < k_n$  and the tree is right-skewed  
Note that when  $k_n < k_{n-1} < \dots < k_1$ , the tree will be left-skewed (try!)

So, in the worst-case,  $h = n - 1 = O(n)$

Therefore, searching, insertion, deletion take  $O(h) = O(n)$  time each!

This is as bad as using singly linked-lists!

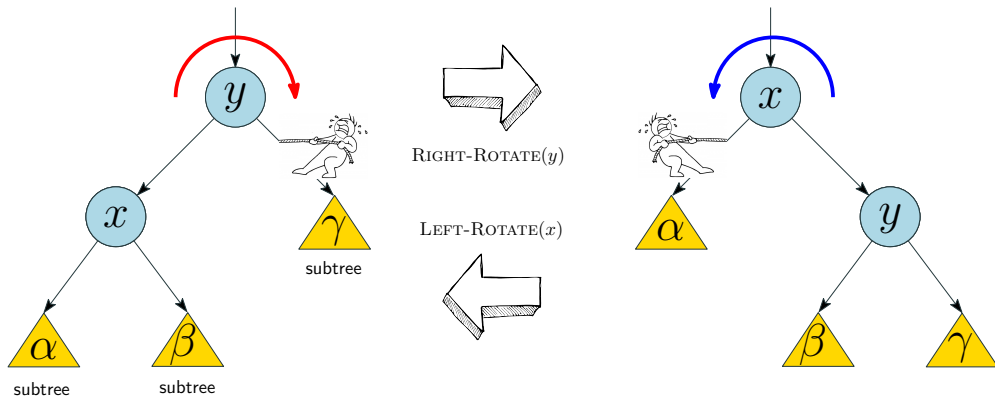


Do something so that  $h$  remains under control (logarithmic)  
We aim for  $h = O(\log n)$

*A solution.* use **Red-Black** trees

**Red-Black** trees are never skewed or close to being skewed unlike the plain binary search trees we just talked about 👍

# Rotation on BSTs

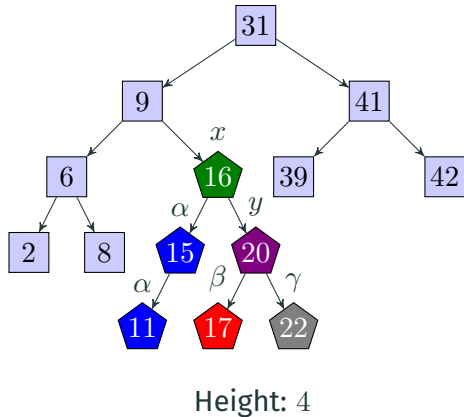
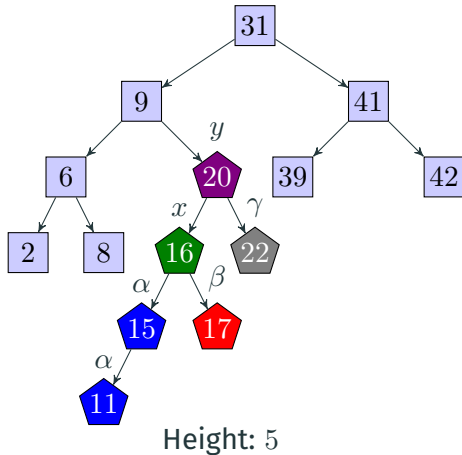


Rotations help in **reducing** height of BSTs; this means faster operations on BSTs

A single rotation can be done in  $O(1)$  time

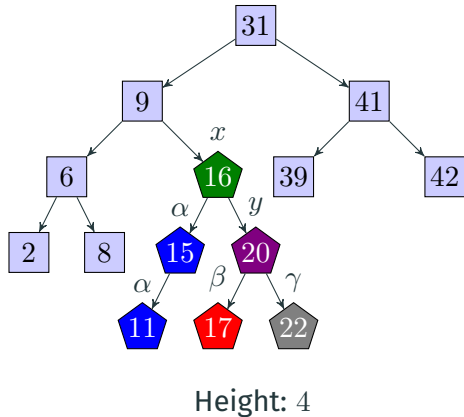
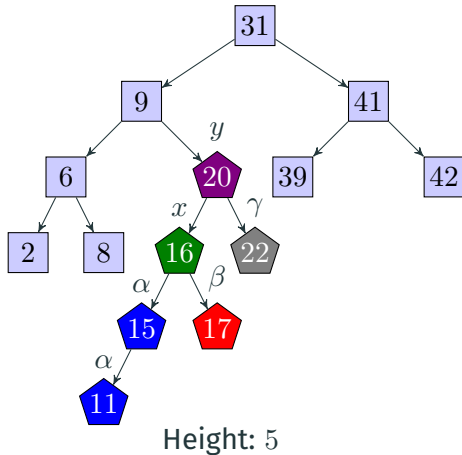
☞ Some of the three subtrees ( $\alpha, \beta, \gamma$ ) can be empty (devoid of nodes).

## An example of rotation



- ☞ If a **right** rotation is performed at node  $y$  in the left tree, we get the tree on the right.
- ☞ If a **left** rotation is performed at node  $x$  in the right tree, we get back the tree on the left.

## An example of rotation



Note that rotations never alter the inorder traversal sequences. It also means that after a rotation, the resulting binary tree is still a binary search tree.

# Code for rotation

## Right rotation at node $y$

```
private void rightRotateAt(Node<K,V> y) {
    Node<K,V> x = y.left;
    y.left = x.right;

    if( x.right != null )
        x.right.parent = y;

    x.parent = y.parent;

    if( y.parent == null )
        root = x;
    else if( y == y.parent.right )
        y.parent.right = x;
    else
        y.parent.left = x;

    x.right = y;
    y.parent = x;
}
```

## Left rotation at node $x$

```
private void leftRotateAt(Node<K,V> x) {
    Node<K,V> y = x.right;
    x.right = y.left;

    if( y.left != null )
        y.left.parent = x;

    y.parent = x.parent;

    if( x.parent == null )
        root = y;
    else if( x == x.parent.left )
        x.parent.left = y;
    else
        x.parent.right = y;

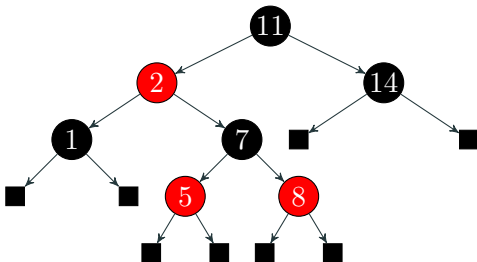
    y.left = x;
    x.parent = y;
}
```

A single rotation (left/right) takes  $O(1)$  time

## Definition

A **red-black** tree is a self-balancing BST (height never gets too bad) that has the following properties:

- 1 Every node is either **red** or **black**
- 2 The root is **black**
- 3 If a node is **red**, then both its children are **black**. The **null** links (shown using black squares in the figure) of the leaves are **black**.
- 4 The number of **black** nodes in any path from the root to a leaf is the same



## How to maintain the colors?

Since the colors are bichromatic, a boolean variable `color` is enough to specify the color of a node. Set `color` to `false`, when the node is painted **RED**, or else, set it to `true` to specify the color is **BLACK**.

☞ There is no magic behind these two colors; feel free to use any two colors

# A node class implementing for RB-tree

```
public class TreeMapRBTree<K extends Comparable<K>, V> {
    private static final boolean RED = false, BLACK = true;

    private static class Node<K, V> {
        final private K key;
        private V val;

        private Node<K, V> left, right, parent;

        private boolean color;

        public Node(K k, V v) {
            key = k; val = v;
            left = right = parent = null;
            color = RED;
        }

        public String toString() {
            String colorString = (color == RED)? "RED" : "BLACK";
            if (val != null) return "<" + key.toString() + ", " + val.toString() + ", " + colorString + ">";
            else return key.toString();
        }

        // other variables and methods
    }
}
```



## Height of a red-black tree

- A RB-tree containing  $n$  records has height  $\leq 2\log_2(n+1) = O(\log n)$
- **Intuition.** By constraining the node colors on any simple path from the root to a leaf (property 4), it can be ensured that no such path is more than twice as long as any other so that the tree's height is always logarithmic.

### Implication

When  $n = 1,000,000$ ,  $h \leq 2\log_2(n+1) < 40$ .

To search for a record, at most 41 comparisons are required in this case.

In contrast, when a plain binary search tree is used,  $h \leq 999,999$ .

For searching for a record, at most 1,000,000 comparisons are required in this case.

This clearly demonstrates the advantage of using RB-trees over plain BSTs and also shows that RB-trees can never be skewed.

 Searching, insertion, and deletion in RB-trees take  $O(h) = O(\log n)$  time each.

## The three primary operations

- ❶ **Search.** same as the search operation for plain BSTs; takes  $O(h) = O(\log n)$  time (note that RB-trees are also BSTs, so the same search algorithm works here too!)
- ❷ **Insertion.** we **will** discuss this; takes  $O(\log n)$  time
- ❸ **Deletion.** we **won't** discuss this; takes  $O(\log n)$  time

👉 The TreeMap class in Java implements RB-Tree

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/TreeMap.html>

## Inserting a new node $z$ into a RB-tree

- 1 Color  $z$  **red** and insert it as you would in a plain BST
- 2 If necessary, start fixing the tree (using rotations and recoloring) as long as you see  $z$ 's parent is **red** ( $z$  may change as we climb up the tree); see the cases next

```
while( z.parent != null && z.parent.color == RED ){  
    // deal with the cases inside this loop  
}
```

- 3 At the end, color the root using **black**

```
root.color = BLACK;
```

# Terminologies

## Uncle of a node

The uncle of a node is the sibling of its parent. In some cases, it could be a null link if there is no such sibling node.

```
private boolean isLeftChild( Node<K,V> node ) {  
    return node.parent != null && node.parent.left == node;  
}  
  
private Node<K,V> uncle( Node<K,V> node ){  
    if( node == null ) return null;  
    return ( isLeftChild(node.parent) ) ? node.parent.parent.right : node.parent.parent.left;  
}
```

## Grandparent of a node

The grandparent of a node is the parent of its parent. In some cases, it could be a null link if there is no such grandparent node.

```
private Node<K,V> grandParent( Node<K,V> node ){  
    return node.parent.parent;  
}
```

## The six cases of insertion

- (Case 1a)  $z$ 's uncle  $y$  is **RED** and  $z$  is a right child
- (Case 1b)  $z$ 's uncle  $y$  is **RED** and  $z$  is a left child
- (Case 2a)  $z$  is a right child,  $z$ 's parent is a left child, and  $z$ 's uncle  $y$  is **BLACK**
- (Case 2b)  $z$  is a left child,  $z$ 's parent is a right child, and  $z$ 's uncle  $y$  is **BLACK**
- (Case 3a)  $z$  is a left child,  $z$ 's parent is left child, and  $z$ 's uncle  $y$  is **BLACK**
- (Case 3b)  $z$  is a right child,  $z$ 's parent is a right child, and  $z$ 's uncle  $y$  is **BLACK**

### Things to note

- 1 The first node inserted into an empty red-black tree is always painted **BLACK**
- 2 Since the null links are considered to be **BLACK**, make sure that you consider their existence and color while finding the uncle of nodes
- 3 In each of the six cases, the parent of  $z$  is always **RED** (if it is **BLACK**, no fixing would be required)

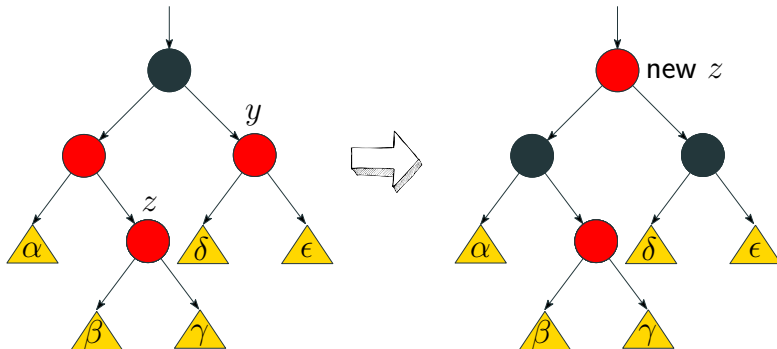
# Case 1



(Case 1a)  $z$ 's uncle  $y$  is **RED** and  $z$  is a right child

(Case 1b)  $z$ 's uncle  $y$  is **RED** and  $z$  is a left child

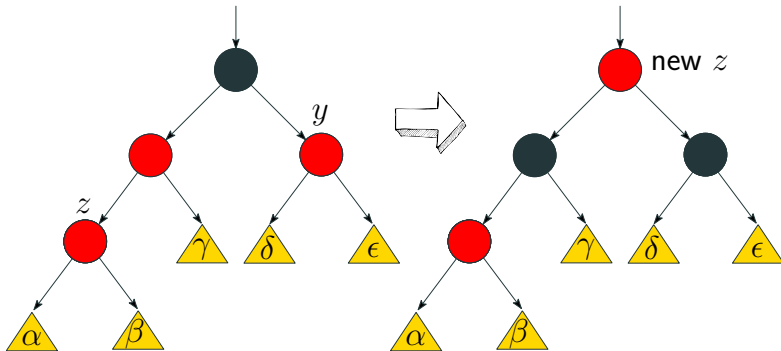
## Case 1a



$z$ 's uncle  $y$  is **red** and  $z$  is a right child; recoloring is needed but no rotation; takes  $O(1)$  time; **continue fixing up** using the new  $z$  node

- `z.parent.color = BLACK; y.color = BLACK; grandParent(z).color = RED;`
- `z = grandParent(z); // continue fixing up using the new 'z'`

## Case 1b



$z$ 's uncle  $y$  is **red** and  $z$  is a left child; recoloring is needed but no rotation; takes  $O(1)$  time; **continue fixing up** using the new  $z$  node

- `z.parent.color = BLACK; y.color = BLACK; grandParent(z).color = RED;`
- `z = grandParent(z); // continue fixing up using the new 'z'`



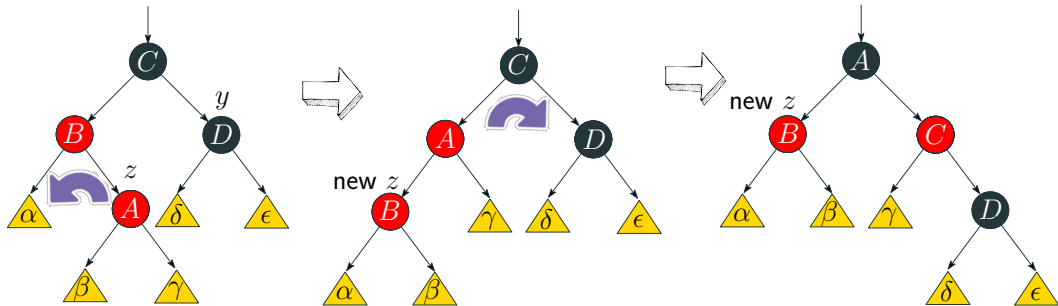
## Case 2



(Case 2a)  $z$  is a right child,  $z$ 's parent is a left child, and  $z$ 's uncle  $y$  is **BLACK**

(Case 2b)  $z$  is a left child,  $z$ 's parent is a right child, and  $z$ 's uncle  $y$  is **BLACK**

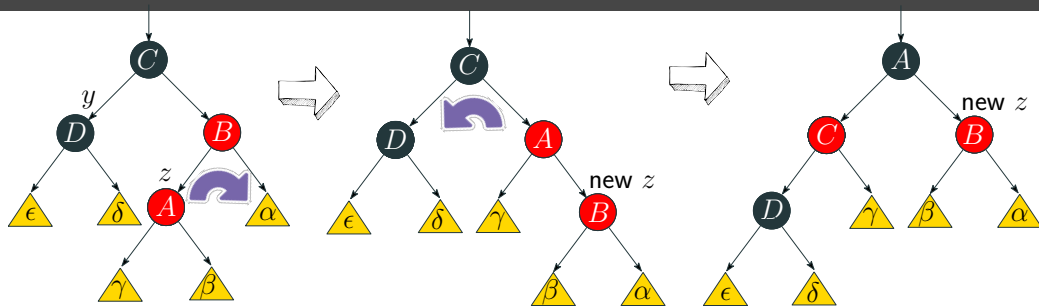
## Case 2a



$z$  is a right child,  $z$ 's parent is a left child, and  $z$ 's uncle  $y$  is **black**; recolorings + 2 rotations are needed; the fixing-up process **terminates** since  $z$ 's parent is **black** after the two rotations; takes  $O(1)$  time

- `z = z.parent; leftRotateAt(z);`
- `z.parent.color = BLACK; grandParent(z).color = RED; rightRotateAt(grandParent(z));`

## Case 2b



$z$  is a left child,  $z$ 's parent is a right child, and  $z$ 's uncle  $y$  is **black**; recolorings + 2 rotations are needed; the fixing-up process **terminates** since  $z$ 's parent is **black** after the two rotations; takes  $O(1)$  time

- `z = z.parent; rightRotateAt(z);`
- `z.parent.color = BLACK; grandParent(z).color = RED; leftRotateAt(grandParent(z));`

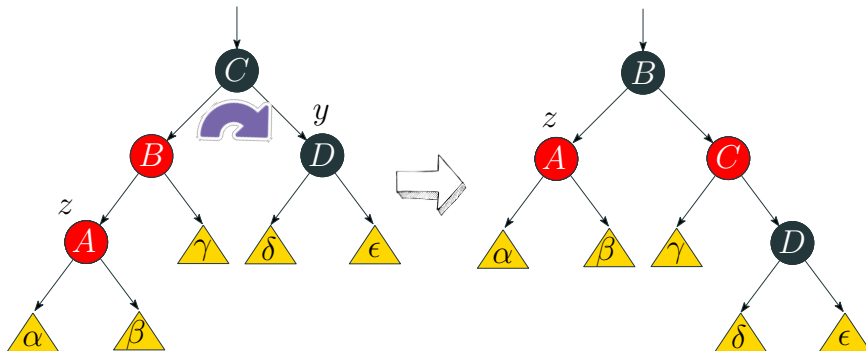
## Case 3



(Case 3a)  $z$  is a left child,  $z$ 's parent is left child, and  $z$ 's uncle  $y$  is **BLACK**

(Case 3b)  $z$  is a right child,  $z$ 's parent is a right child, and  $z$ 's uncle  $y$  is **BLACK**

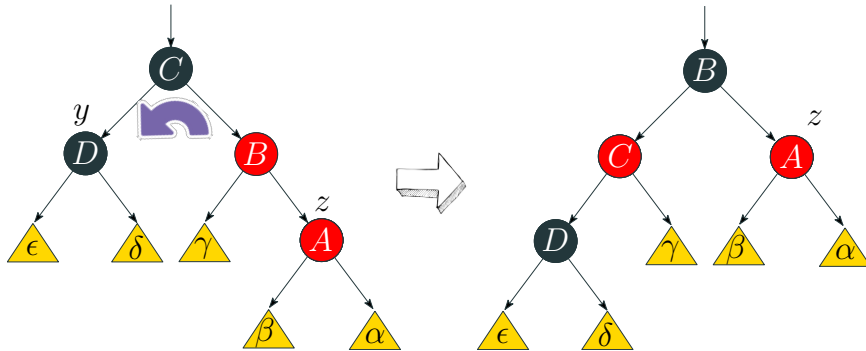
## Case 3a



$z$  is a left child,  $z$ 's parent is left child, and  $z$ 's uncle  $y$  is **black**; recolorings + 1 right rotation are needed; the fixing-up process **terminates** since  $z$ 's parent is **black** after the rotation; takes  $O(1)$  time

```
• parent(z).color = BLACK; grandParent(z).color = RED; rightRotateAt(grandParent(z));
```

## Case 3b



$z$  is a right child,  $z$ 's parent is a right child, and  $z$ 's uncle  $y$  is **black**; recolorings + 1 left rotation are needed; the fixing-up process **terminates** since  $z$ 's parent is **black** after the rotation; takes  $O(1)$  time

```
• parent(z).color = BLACK; grandParent(z).color = RED; leftRotateAt(grandParent(z));
```

The second rotation of Case 2a is exactly an execution of Case 3a  
Similarly, the second rotation of Case 2b is exactly an execution of Case 3b

See the class `TreeMapRBTree`

## In-browser visualization

41, 38, 31, 12, 19, 8

The value parts of the above records are ignored in this example

ITEM	ACTION
41	The only node, just color it <b>black</b>
38	Insert it to the left of 41; its parent is <b>black</b> , so, no action is needed
31	Case 3a; 31's uncle (a null reference) is <b>black</b> ; right rotate at 41
12	Case 1b; a simple recoloring is enough
19	Case 2a; two rotations are needed
8	Case 1b; a simple recoloring is enough

<https://www.cs.csubak.edu/~msarr/visualizations/RedBlack.html>



## Observations

- Time taken for inserting a new node is  $O(h) = O(\log n)$ ; after this fix-up may be needed
- During fix-up, we climb up the tree using Case 1, which only recolors but never rotates
- If we ever use Case 2 (uses 2 rotations) or 3 (uses 1 rotation), we are done (fix-up process terminates)!
- This means at every insertion of a new item, at most 2 rotations are needed
- At most  $h$  executions of Case 1 are needed plus 1 execution of Case 2/3, each taking  $O(1)$  time
- So, the total time taken for fix-up is  $(h + 1) \times O(1) = O(\log n) \times O(1) = O(\log n)$ , since, for RB-trees,  $h = O(\log n)$
- Total time taken for one insertion equals time taken for inserting a new node plus total time taken for fix-up  $= O(\log n) + O(\log n) = 2 \times O(\log n) = O(\log n)$
- Time complexity of searching is  $O(h) = O(\log n)$ , since, for RB-trees,  $h = O(\log n)$
- Deletion also takes  $O(h) = O(\log n)$  time but we are not going to discuss it

## Plain BSTs vs RB-Trees: tree creation time (using $n$ insertions)

$n$	RB-tree	Plain BST
10	1	1
100	1	1
1000	3	13
10000	7	214
100000	20	17249

When the input records were already **sorted** in ascending order, RB-trees could easily beat plain BSTs since the heights of the plain BSTs were exactly  $n - 1$  everywhere (much worse than the logarithmic heights of RB-trees).

$n$	RB-tree	Plain BST
10	1	1
100	1	1
1000	3	2
10000	7	8
100000	72	49

When the input records were in **random** order, plain BSTs performed quite as fast as RB-trees since the heights of plain BSTs were not  $n - 1$  or even close (in fact, they were almost logarithmic like RB-trees).

👉 Times are reported in milliseconds

## Are plain BSTs completely useless?

- Plain BSTs perform **terribly** when the inputs are sorted (or, almost sorted) in ascending or descending order
- But, BSTs are found to perform great on randomly ordered inputs
- In those cases,  $h$  is found to be much less than  $n - 1$  and is almost logarithmic
- Consequently, we get to see fast searching, insertion, and deletion times
- **Example.** when  $n = 5000$ , heights of plain BSTs are around 30 if the input is randomly ordered. Note that this is much less than 4999 (worst case height)

<https://opensa-server.cs.vt.edu/ODSA/Books/Everything/html/BST.html>