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**: $f(\mathrm{key}) = \texttt{value}$

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 $O(n)$ time for figuring out the correct spot for the incoming record; then $O(n)$ time for shifting items to the right to accommodate the new record; total time taken is $O(n) + O(n) = O(n)$
- **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 kill 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 kill the empty spot; total time taken $O(n) + O(n) = O(n)$

Our aim

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

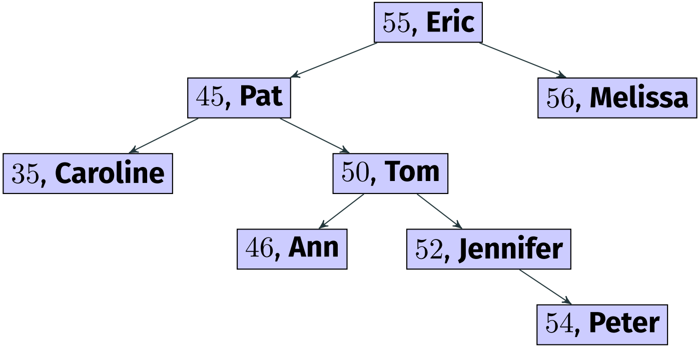
☞ **Balanced** binary search trees is the solution; stay tuned ...

The map ADT

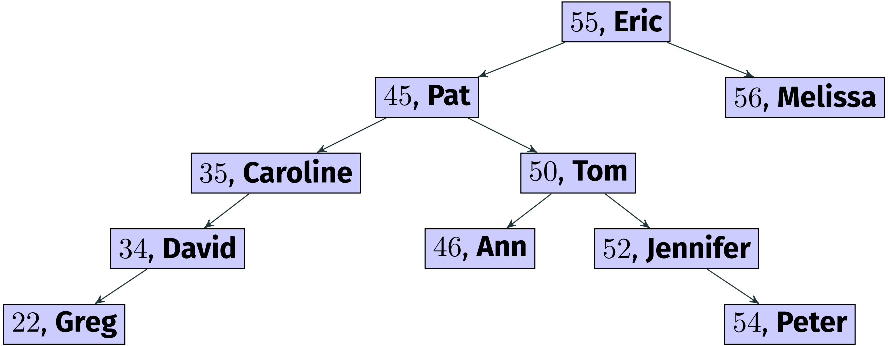
```java
public interface MapADT<K,V> {
   boolean put(K key, V value); // adds a new record with key 'key' and value 'value'
   V remove(K key); // removes the record having key 'key'
   V get(K key); // return the value part of the record whose key is 'key'
   V updateValue(K key, V newValue); // updates 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?

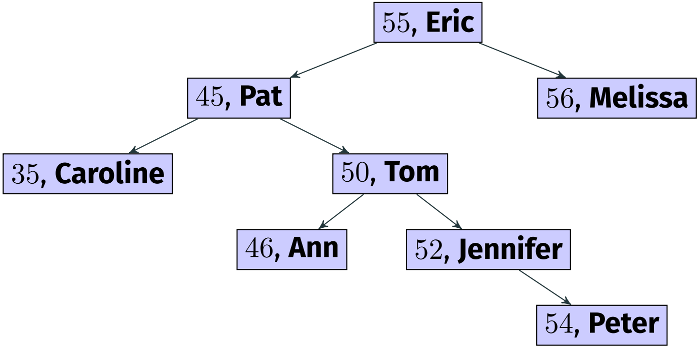


Use

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**

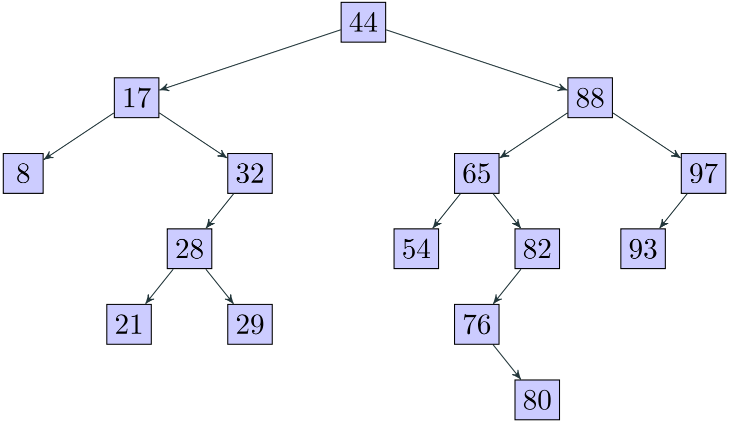
Let’s say you need to look for the record that has the key k; how will you do this?

Algorithm

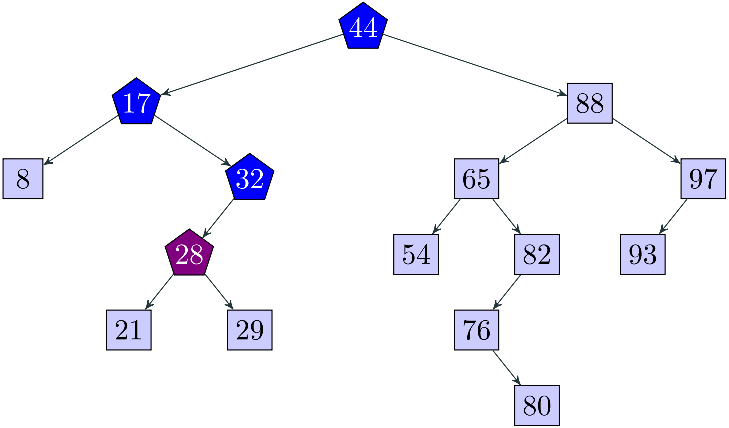
- Start at the root
- If the root’s key is k, then search is successful
- If $k <$ root’s key, search recursively (or iteratively) in the left subtree of the root
- Otherwise, search recursively (or iteratively) in the right subtree of the root
- If we have reached a `null` link, no record exists in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

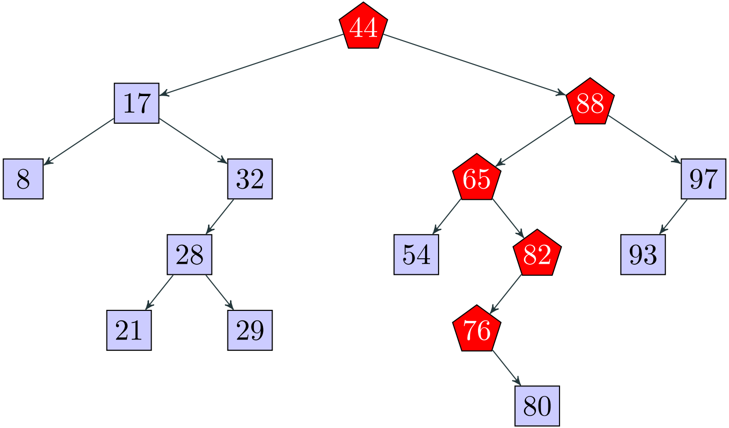


Example: search for 28



Found!

Example: search for 68



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

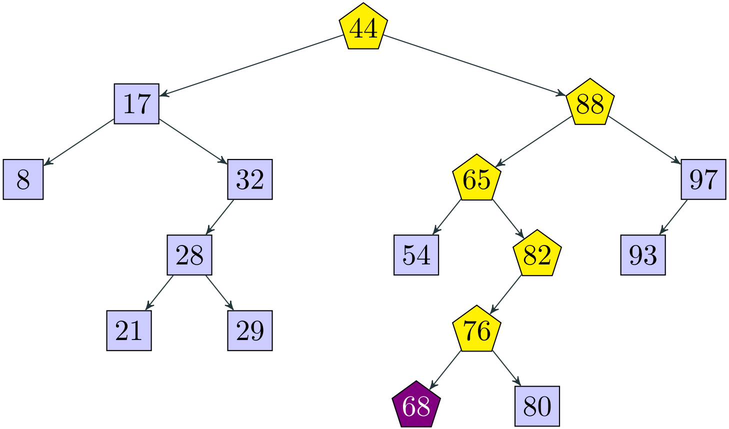
$O(h)$, where h is the height of the BST under consideration, since, for searching, we need to traverse a path whose length is h in the worst-case

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



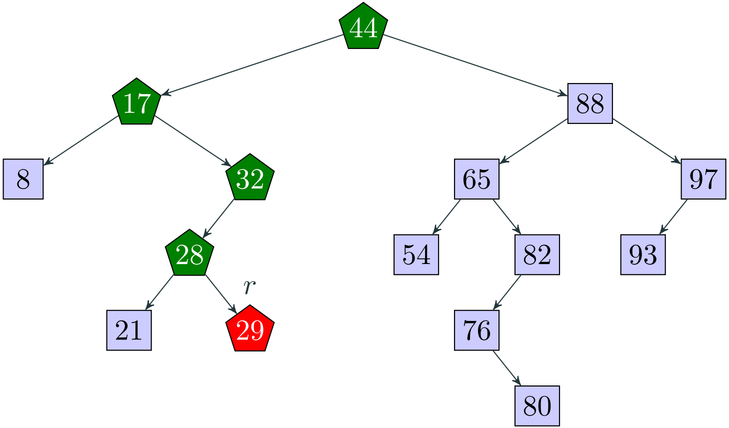
$O(h)$, where h is the height of the BST under consideration, since, for inserting a new record, we need to traverse a path whose length is h in the worst-case

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

Idea

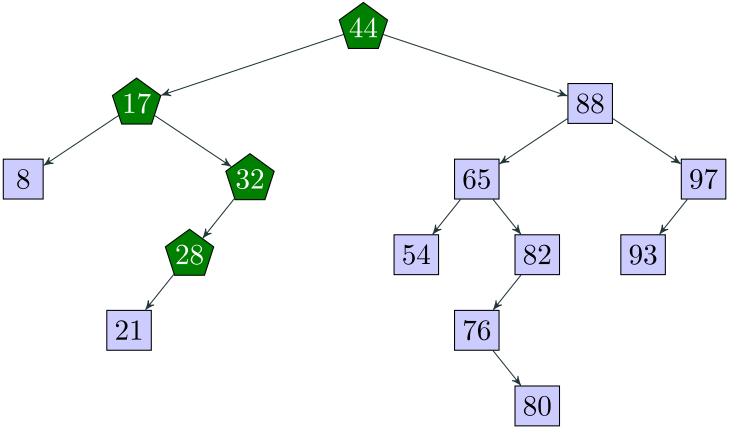
- Traverse through the tree to find the record having key k
- If the record cannot be found, no action is needed
- Now assume that we have found the record that has key k at node r
  1. r is a leaf node (no child)
  2. r has exactly one child (either left or right)
  3. r has two children (both left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

Case 1: r is a leaf node, delete 29

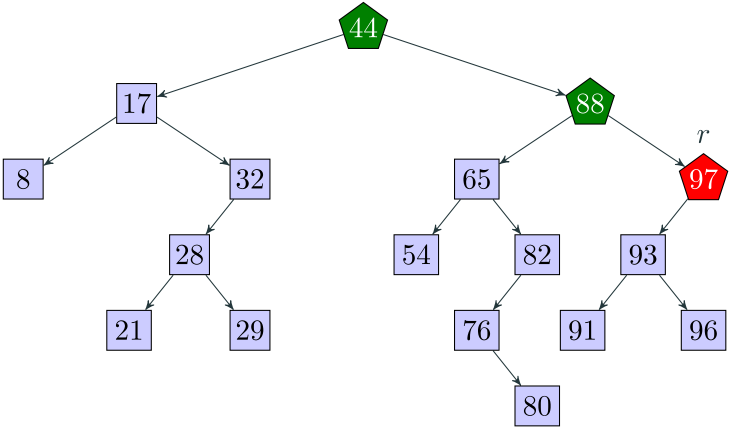


Easy! just delete it right away

Case 1: r is a leaf node, delete 29

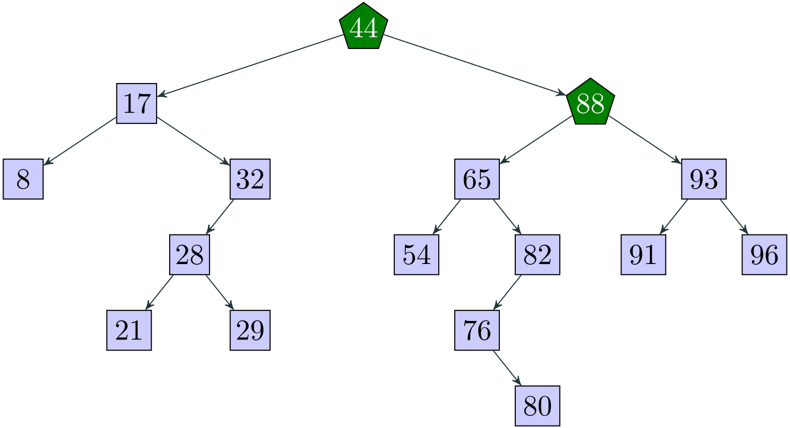


Case 2: r has one child, delete 97

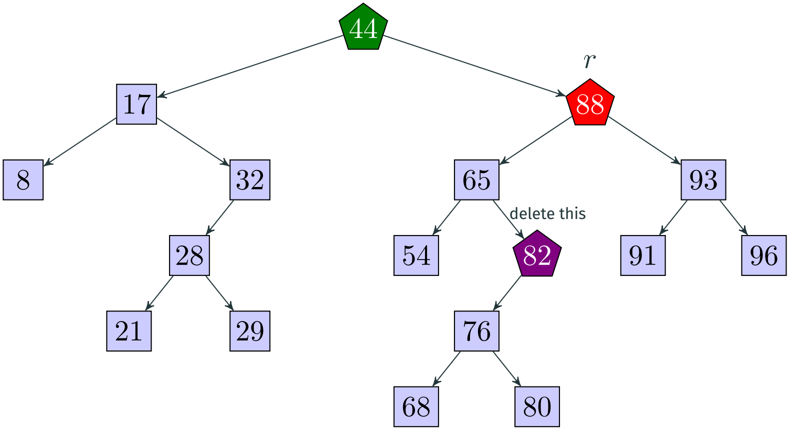


Make the parent of r point to the only child of r instead of r

Case 2: r has one child, delete 97

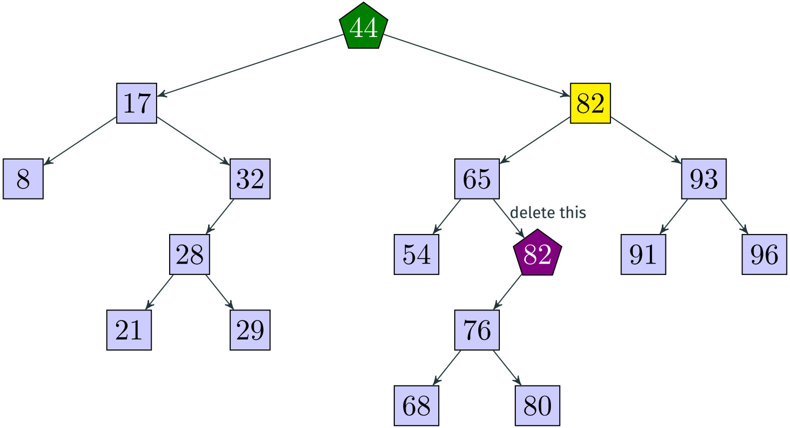


Case 3: r has two children, delete 88

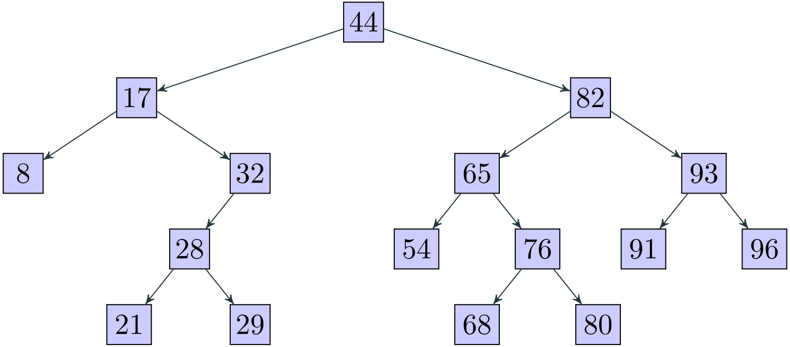


Replace the record at r with the record at the inorder predecessor p of n; then delete p using Case 1 or 2, depending on the number of children of p. Note that the inorder predecessor is either a leaf node or has a left child only (no right child).

Case 3: r has two children, delete 88

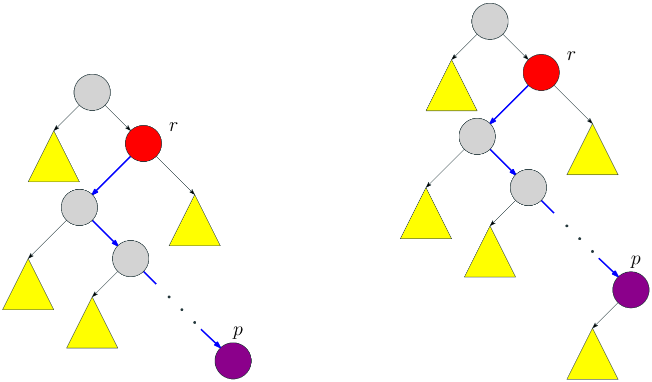


Replace the record at r with the record at the inorder predecessor p of n; then delete p using Case 1 or 2, depending on the number of children of p. Note that the inorder predecessor is either a leaf node or has a left child only (no right child).



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). 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).

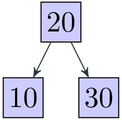
Time complexity of deletion

- 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

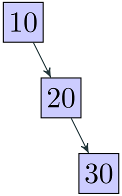
Code

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

Problem

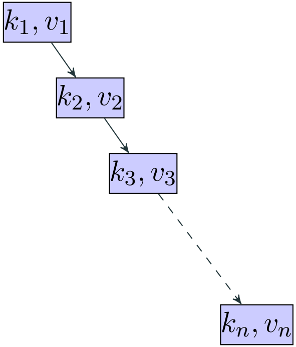
Given a text, find out the unique words in it along with their counts. We also need to output the distinct words with their count.

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, $k_1 < k_2 < k_3 < \ldots < k_n$

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

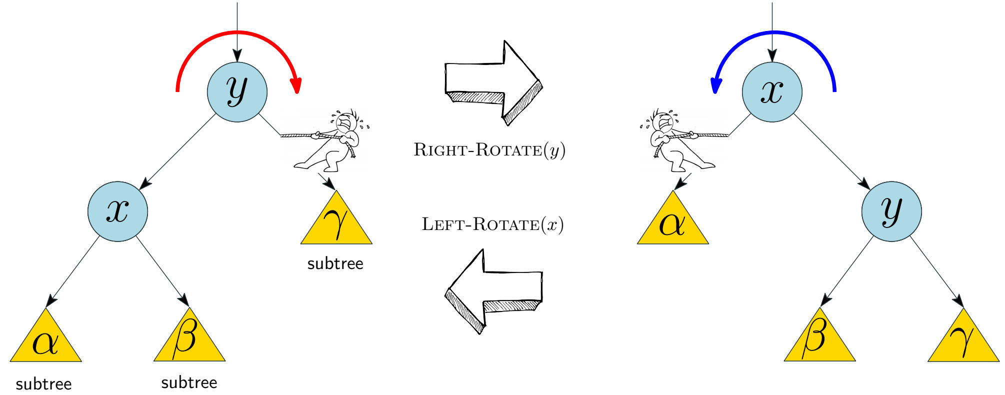
Therefore, searching, insertion, deletion take $O(h) = O(n)$ time each!

This is as bad as using singly linked-lists!

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

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

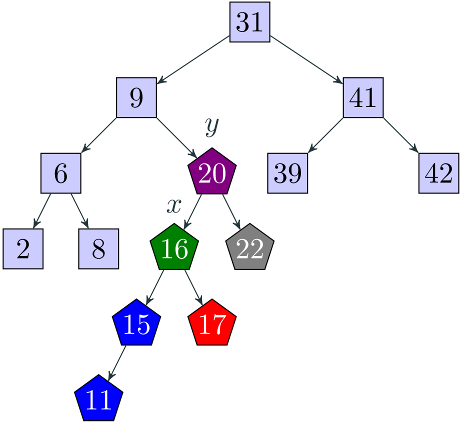
Red-Black trees are never skewed or close to being skewed unlike the plain binary search trees we just talked about 👍

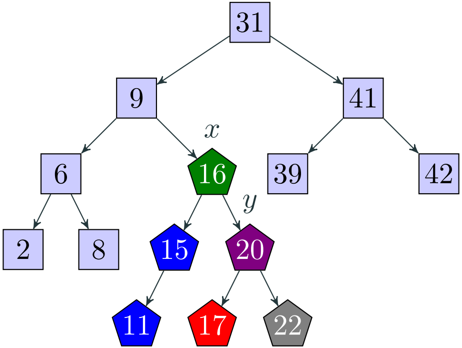


Rotations help in **reducing** height of BSTs; this means faster operations on BSTs

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

Can be applied to any type of **self-balancing** BST (red-black, AVL, etc.)





Height: 5

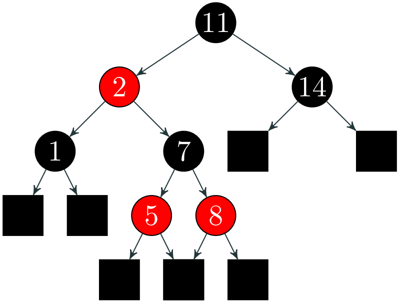
Height: 4

Note that the inorder traversal sequence remains the same!

Definition

A **red**-**black** tree is a self-balancing BST that maintains 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` references of the leaves are **black**)
4. The number of **black** nodes in any path from the root to a leaf is the same



It can be shown mathematically (out of scope) that for red-black trees,

$$h \leq 2\log(n+1) = O(\log n)$$

The three primary operations

1. **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!)
2. **Insertion**. we **will** discuss this; takes $O(\log n)$ time
3. **Deletion**. we **won't** discuss this; takes $O(\log n)$ time

☞ The `TreeSet` class in Java implements RB-Tree

`https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/TreeSet.html`

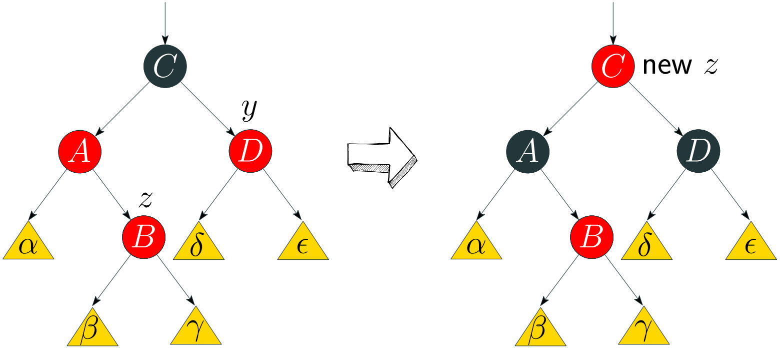
Inserting a new node z into a red-black tree

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

Uncle of a node

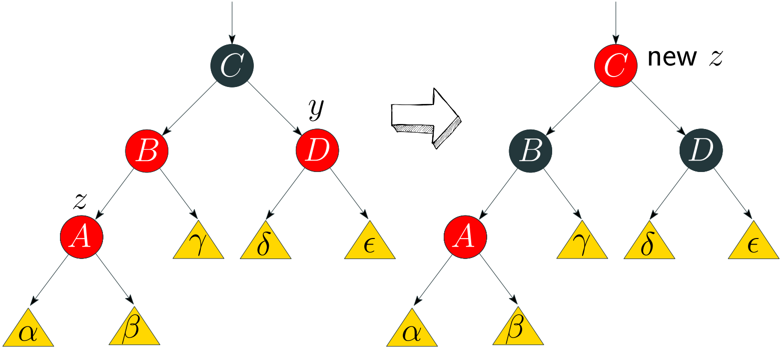
The uncle of a node is the sibling of its parent. In some cases, it could be a null link if there is no such sibling node.

Case 1



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

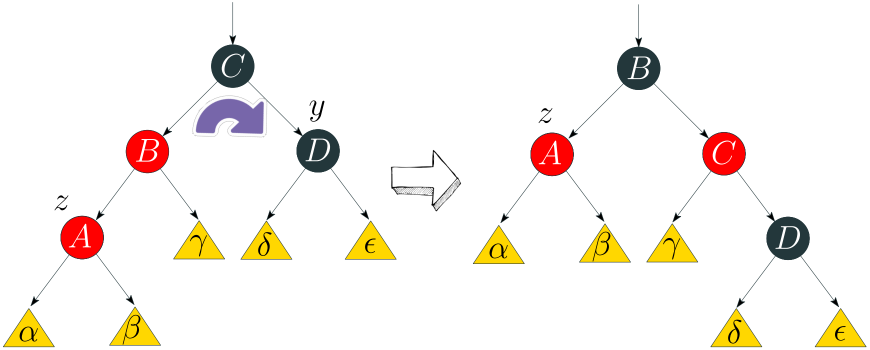
- `parent(`z`).color = `**`BLACK`**`; y.color = `**`BLACK`**`;`
- `parent(parent(`z`)) = `**`RED`**`;`
- `z = parent(parent(`z`));`



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

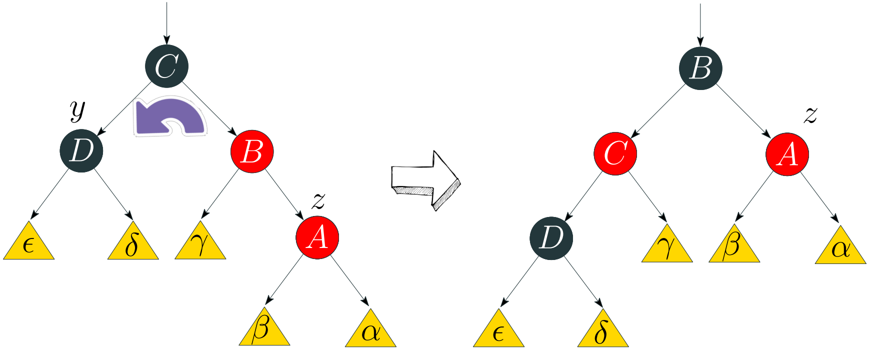
- `parent(`z`).color =` **BLACK**; `y.color =` **BLACK**;
- `parent(parent(`z`)) =` **RED**;
- `z = parent(parent(`z`));`

Case 2



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

- parent(z).color = **BLACK**;
- parent(parent(z)).color = **RED**;
- RIGHT-ROTATE(parent(parent(z)));

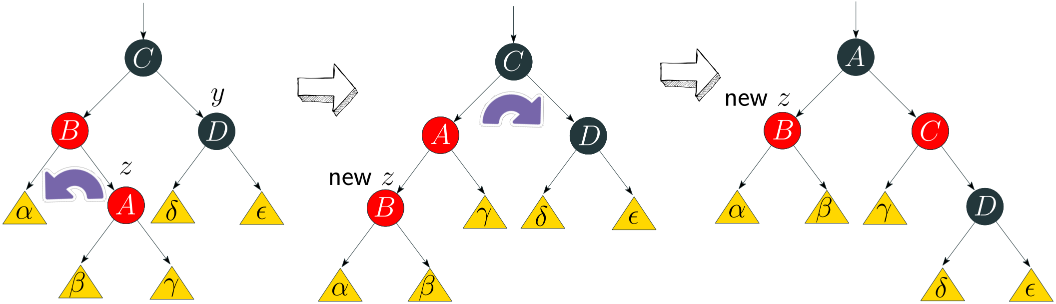


z is the right child and z's uncle y is **black**; recolorings + 1 left rotation are needed; the insertion process terminates since z's parent is **black**; takes $O(1)$ time

- `parent(`z`).color = ` **BLACK**;
- `parent(parent(`z`)).color = ` **RED**;
- Left-Rotate`(parent(parent(`z`)));`

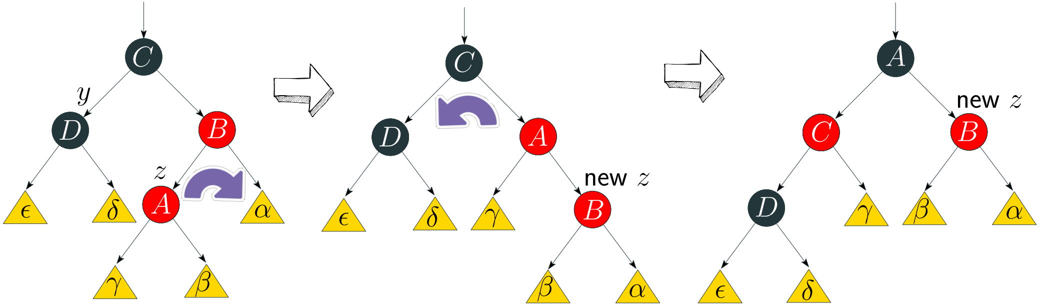
Case 3

Case 3a (mirror case of 3b)



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

Case 3b (mirror case of 3a)



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

$$41, 38, 31, 12, 19, 8$$

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

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

Observations

- During insertion, we climb up the tree using Case 1, which only recolors but never rotates
- If we ever use Case 2 or 3, we are done (insertion 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 time complexity of insertion is $h \times O(1) = O(\log n) \times O(1) = O(\log n)$
- For RB-trees, $h = O(\log n)$
- Similarly the time complexity of searching is $O(\log n)$
- Deletion also takes $O(\log n)$ time but we are not discussing it in this course

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 $n - 1$
- **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)

See the class `TestTreeMapBinarySearchTree` for a demonstration

Reading

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BST.html