AVL Trees

CPT 204 - Advanced OO Programming

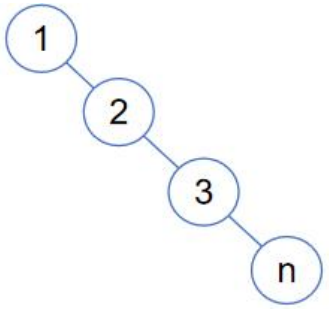
AY 24/25

Objectives

- To know what an ***AVL tree*** is
- To understand how to ***rebalance*** a tree using the ***LL rotation***, ***LR rotation***, ***RR rotation***, and ***RL rotation***
- To know how to design the **`AVLTree`** class
  - To ***insert*** elements into an AVL tree
  - To implement node ***rebalancing***
  - To ***delete*** elements from an AVL tree
  - To implement and test the **`AVLTree`** class
- To analyze the ***complexity*** of search, insert, and delete operations in AVL trees

Why AVL Trees?

The search, insertion, and deletion time for a binary search tree is dependent on the **height of the tree.** In the worst case, the height is **O(n)**, so worse time complexity is **O(n)**



Can we maintain a **perfectly balanced tree (i.e., complete binary tree)**, so that

Level 1 (the root) holds at most 1 node (2^0).

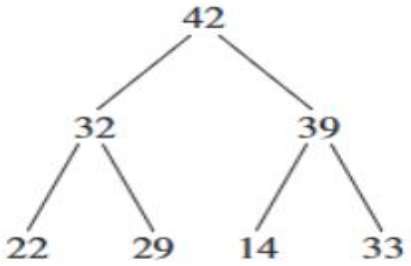
Level 2 holds at most 2 nodes (2^1).

Level 3 holds at most 4 nodes (2^2).

… …

Level h holds at most 2^{h-1} nodes

$n = 2^0+2^1+2^2+...+2^{h-1} \Rightarrow h = \log n$



Yes, but it would be very **costly,** as it requires rebuilding and nodes shifting very often during insertions/deletions

Alternatively, we can maintain a ***well-balanced tree*— the heights of two subtrees for every node are about the same —** so that $h \approx \log n$, while less rebalancing efforts

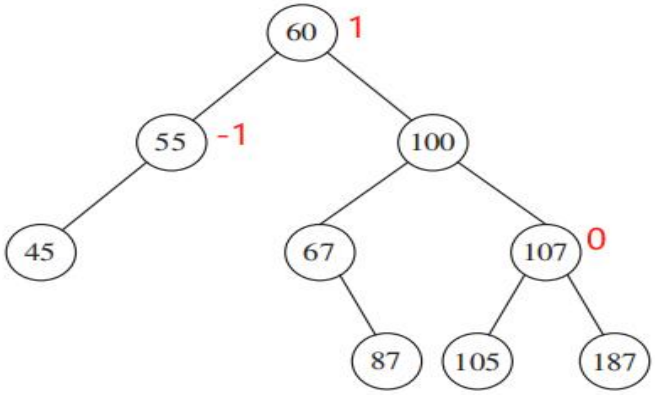
What are AVL Trees?

AVL trees are **well-balanced**. The difference between the heights of every node's two subtrees will not be more than 1.

| height(right subtree) - height(left subtree) | ≤ 1

The ***balance factor*** of a node is the height of its right subtree minus the height of its left subtree

- A node is said to be ***balanced*** if its balance factor is -1, 0, or 1
- Although still balanced, a node is said to be ***left-heavy*** if its balance factor is -1
- Although still balanced, a node is said to be ***right-heavy*** if its balance factor is +1

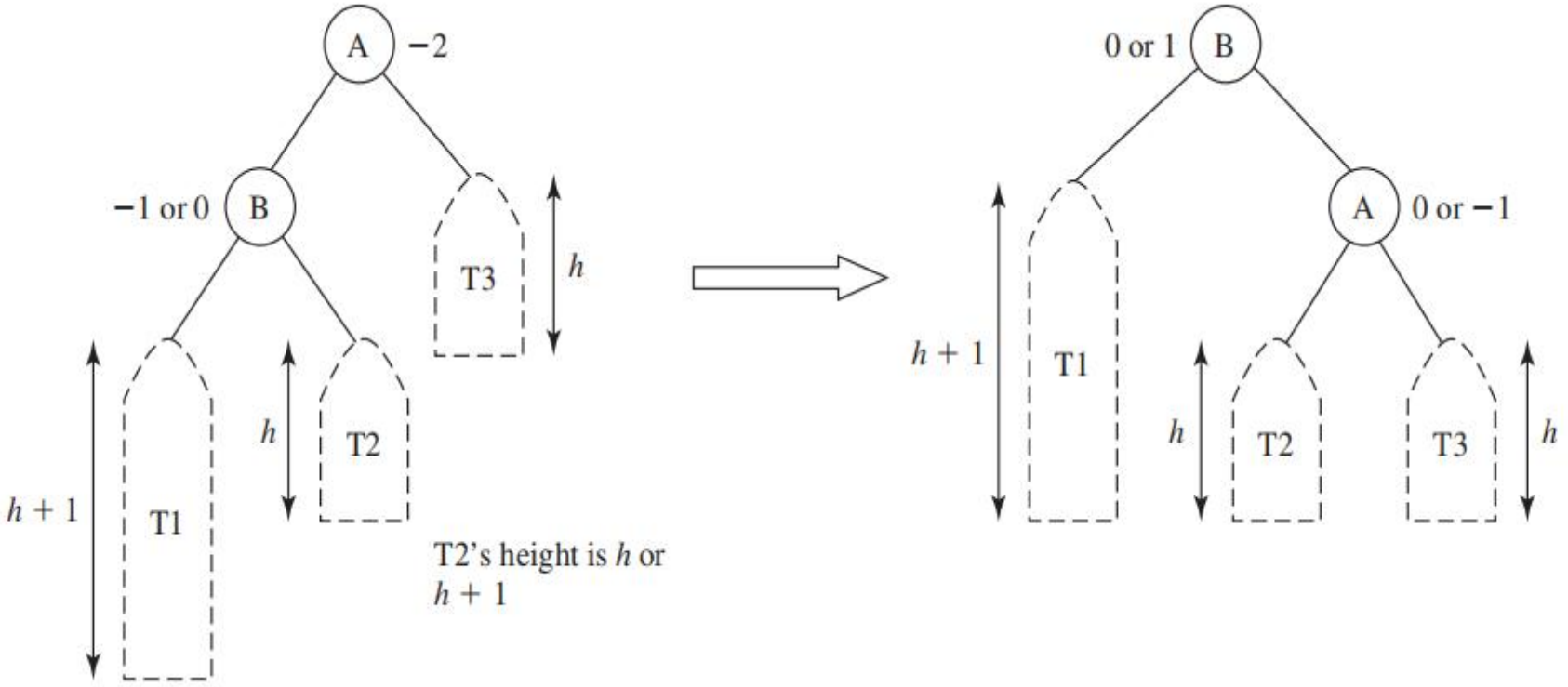


Balancing Trees

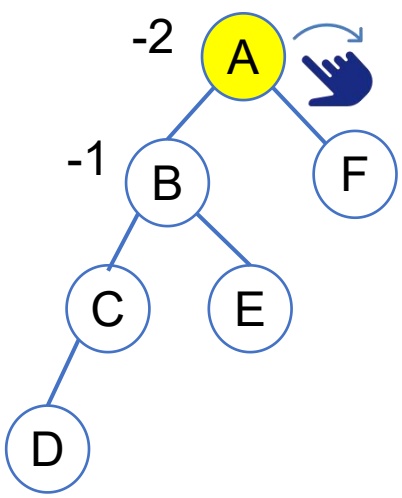
- If a node is **not balanced** (i.e., its balance factor is not -1, 0, or 1) after an insertion or deletion operation, you need to rebalance it:
- The process of rebalancing a node is called a ***rotation***
- There are four possible rotations:
  - ***LL rotation*** (***left-heavy left-heavy rotation***)
  - ***RR rotation*** (***right-heavy right-heavy rotation***)
  - ***LR rotation*** (***left-heavy right-heavy rotation***)
  - ***RL rotation*** (***right-heavy left-heavy rotation***)

LL imbalance and LL rotation

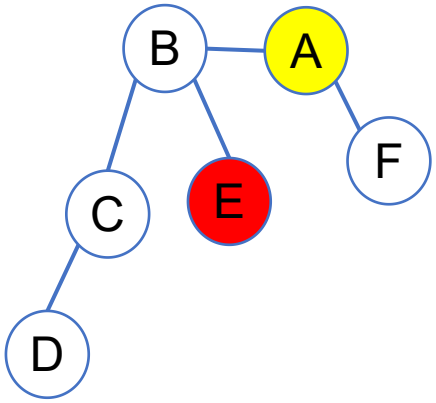
- An ***left-heavy, left-heavy imbalance*** occurs at a node **A** if **A** has a balance factor **-2 (left-heavy)** and its left child **B** has a balance factor **-1 (left-heavy)** or **0**
  - ***LL Rotation (a single right rotation at node A)***:



LL imbalance and LL rotation

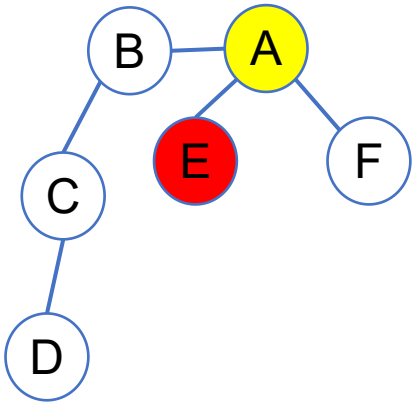


Right-rotating node A, the unaffected node (F) stays in its original place still as the right child of A

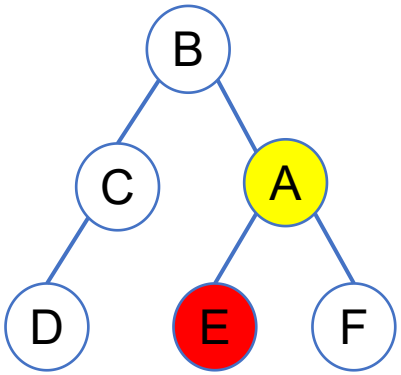


When rotating node A to the right, B replaces A's place (to be the root in this case), while A will move down to become B's right child.

However, B already has a right child (E).



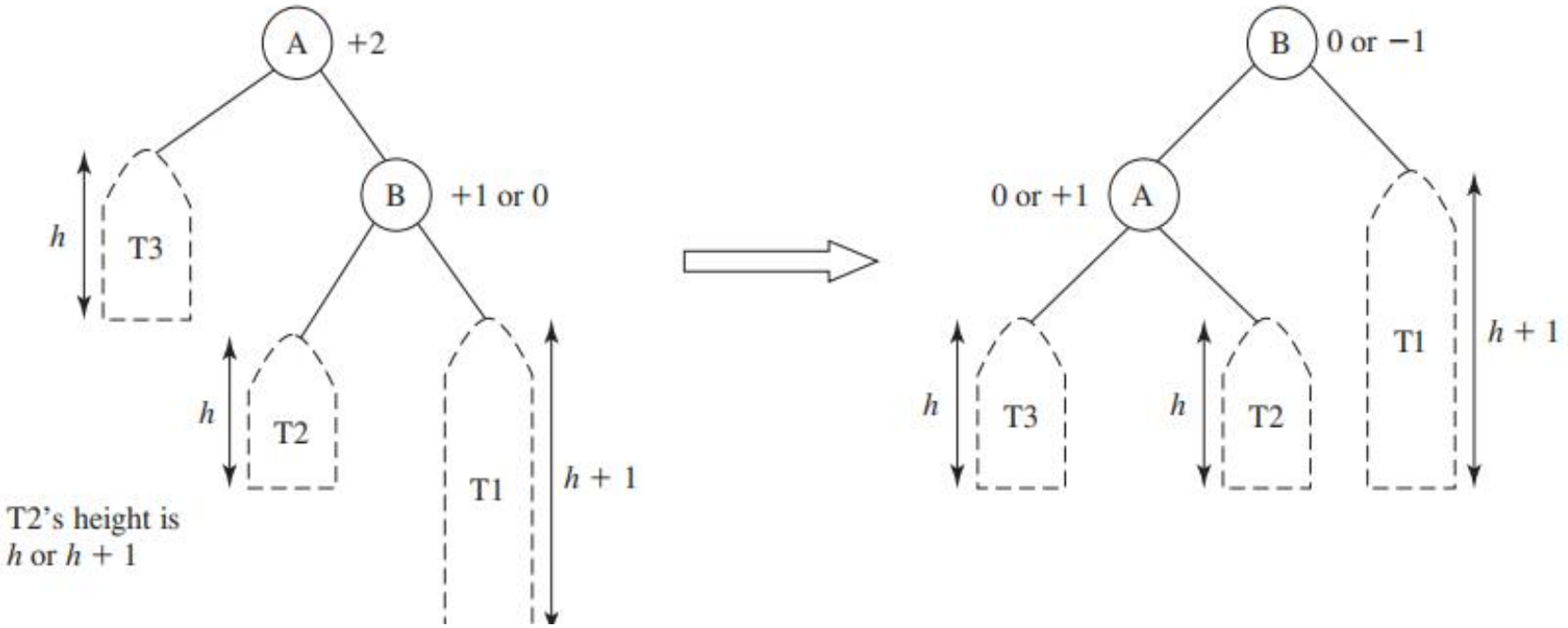
To avoid conflict, E will be attched to A as A's left child during the rotation.



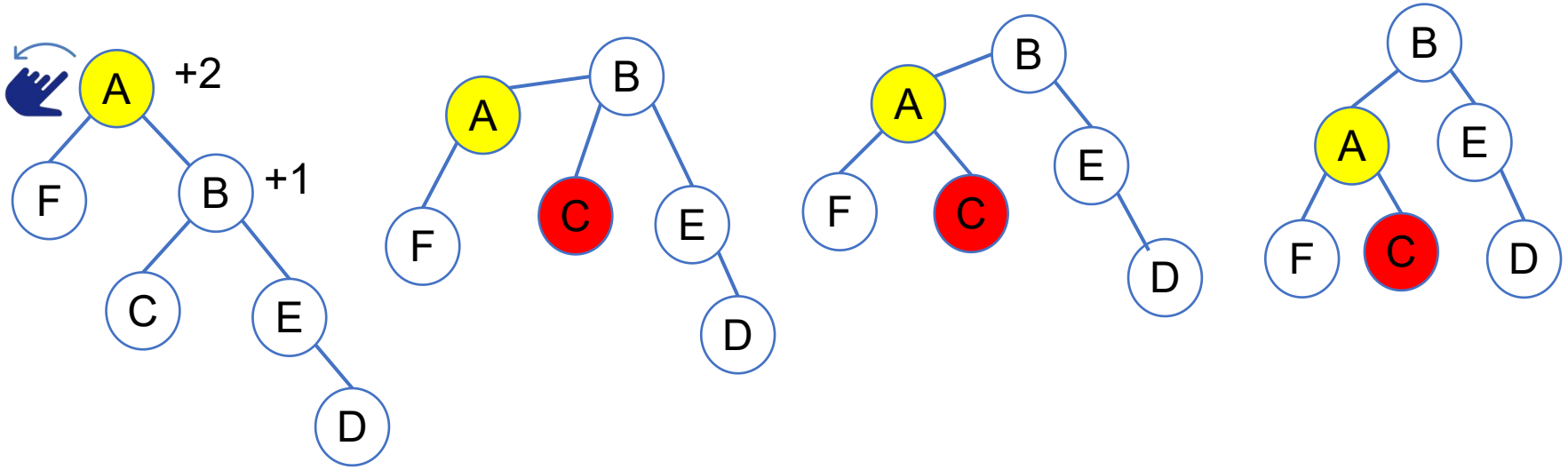
Now, we finish the LL rotation

RR imbalance and RR rotation

- An ***RR imbalance*** occurs at anode **A** if **A** has a balance factor **+2 (right-heavy)** and a right child **B** has a balance factor **+1 (right-heavy)** or **0**
  - ***RR Rotation (a single left rotation o node A)***:



RR imbalance and RR rotation



Left-rotating node A, the unaffected node (F) stays in its original place as the left child of A

When rotating node A to the left, B replaces A's place (root in this case), while A will move down to become B's left child.

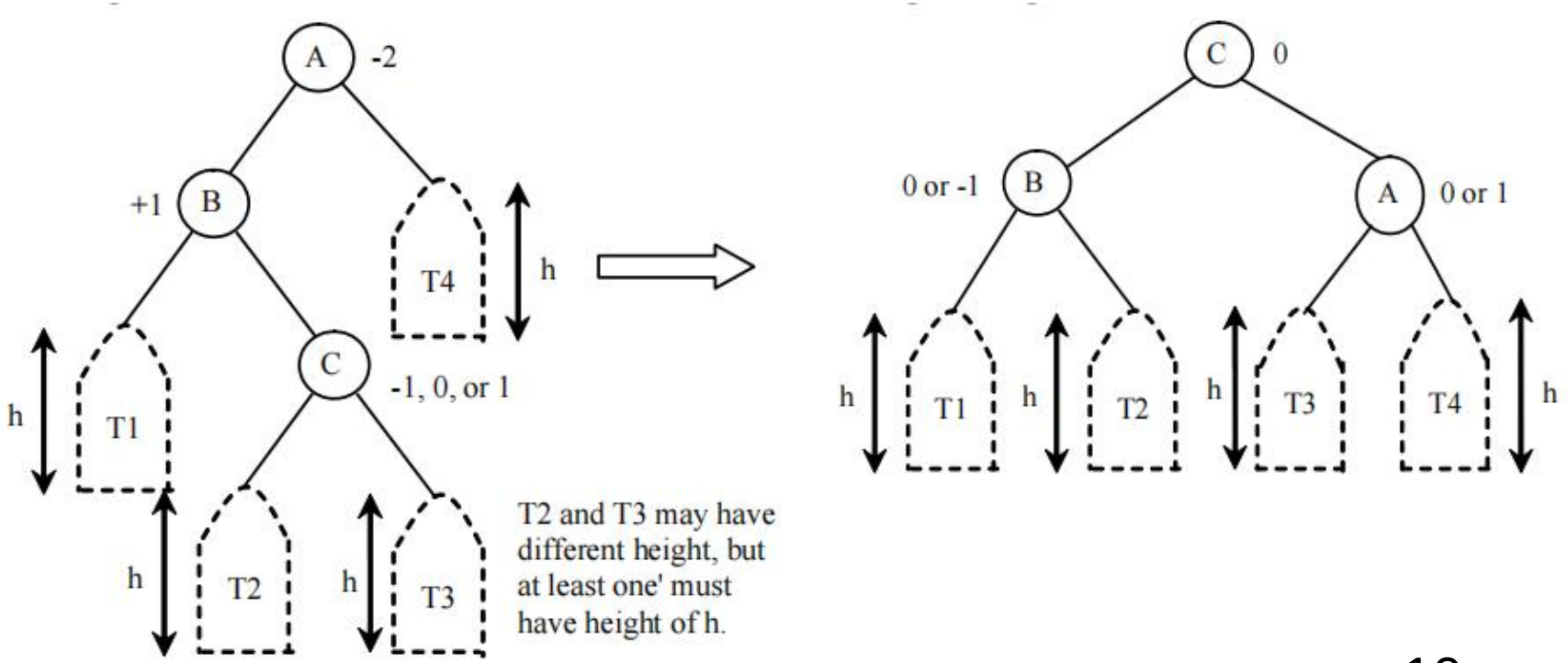
However, B already has a left child (C).

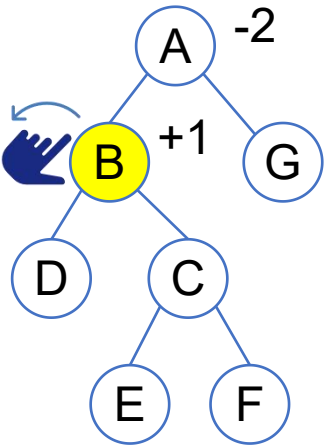
To avoid conflict, C will be attached to A as A's right child during the rotation.

Now, we finish the RR rotation

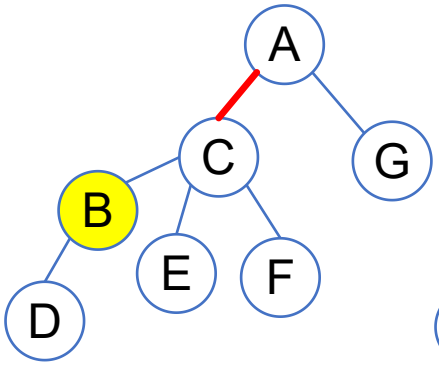
LR imbalance and rotation

- ***LR Rotation***: An ***LR imbalance*** occurs at a node **A** if **A** has a balance factor **-2 (left-heavy)** and a left child **B** has a balance factor **+1 (right-heavy)**
  - This type of imbalance can be fixed by performing a double rotation: first a single left rotation at **B** and then a single right rotation at **A**

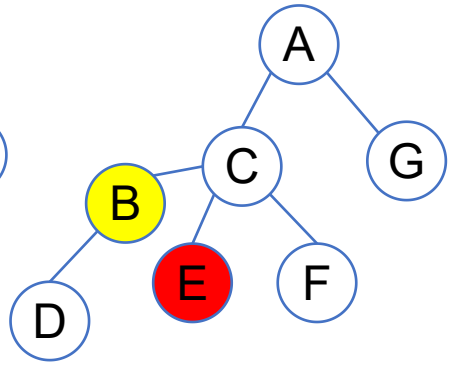




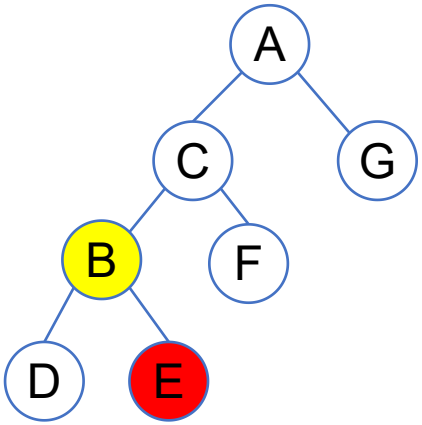
Left-rotating A's left child B



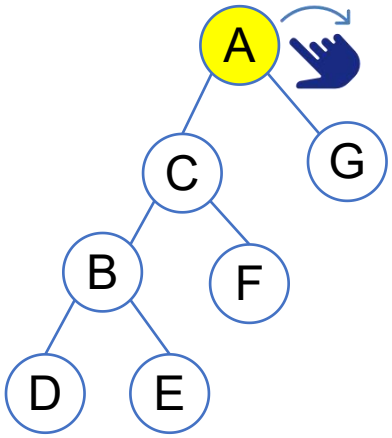
B disconnects with A, while C replaces B's place, to be A's left child.



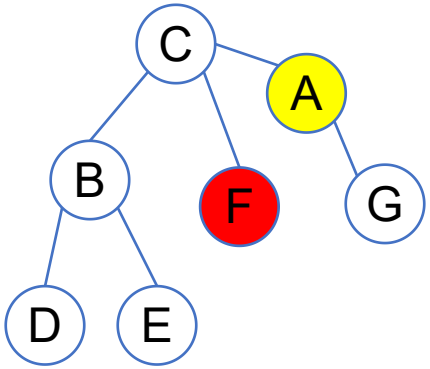
C already has a left child E



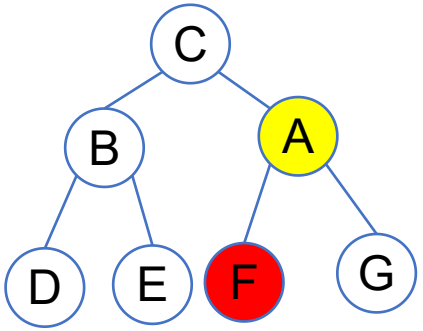
Attach E to be B's right child to avoid conflict and finish rotation at B



Right-rotating A



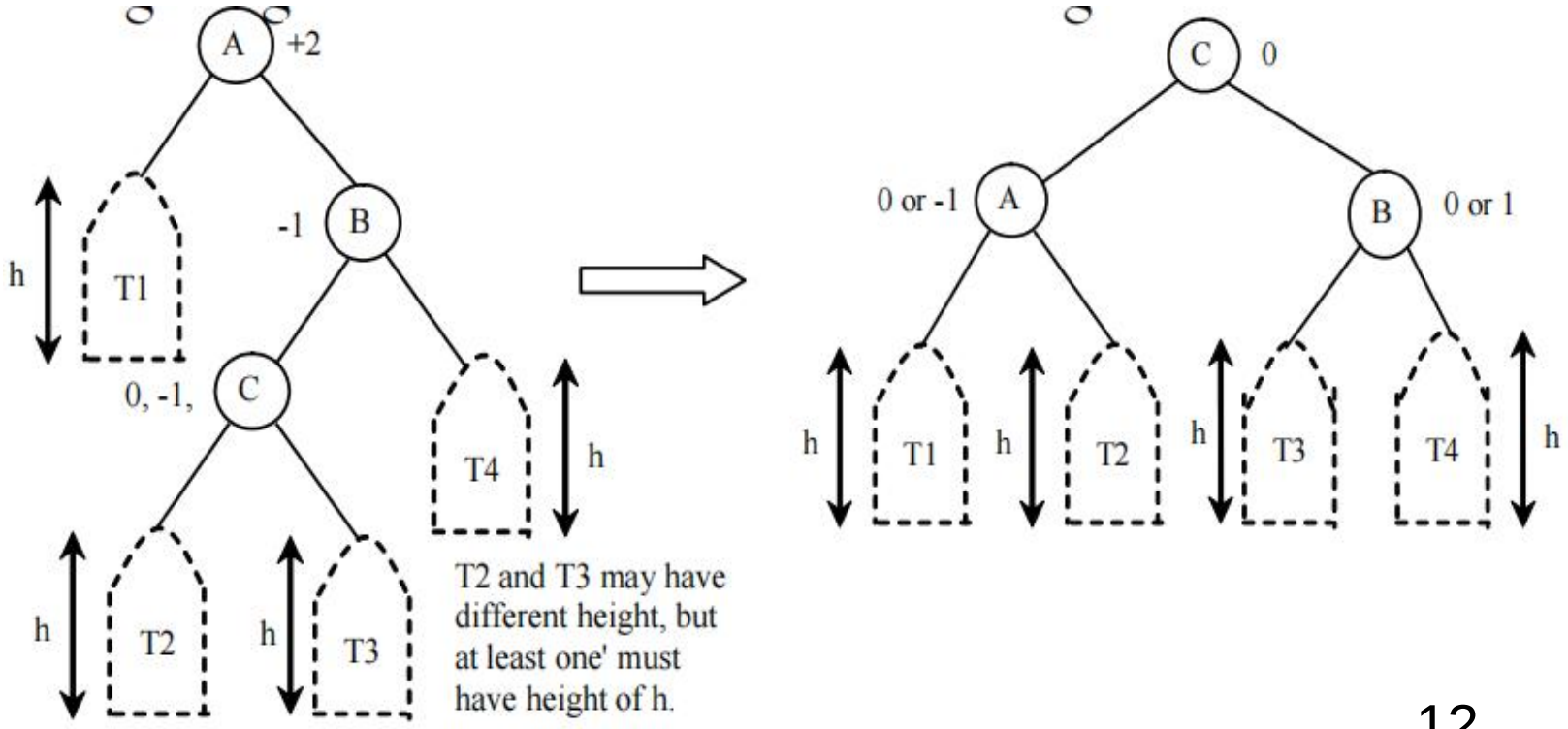
A conflicts with F

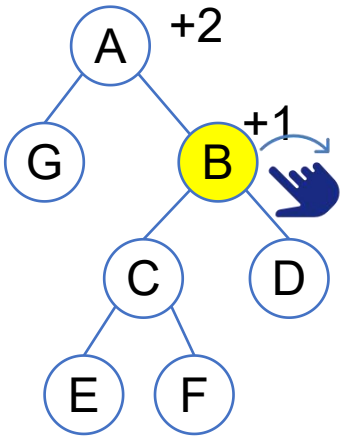


Attach F to be A's left child and finish rotation

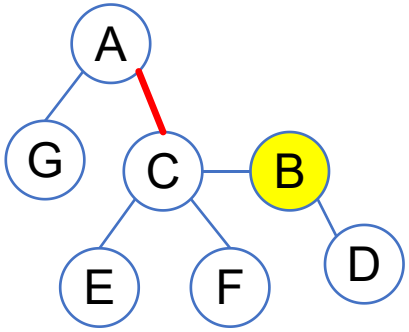
RL imbalance and RL rotation

- ***RL Rotation***: An ***RL imbalance*** occurs at anode **A** if **A** has a balance factor **+2 (right-heavy)** and a right child **B** has a balance factor **-1 (left-heavy)**
  - This type of imbalance can be fixed by performing a double rotation: first a single right rotation at **B** and then a single left rotation at **A**

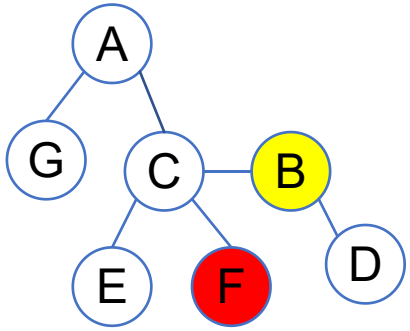




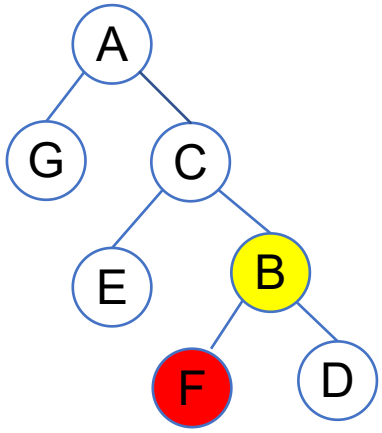
Right-rotating right child B



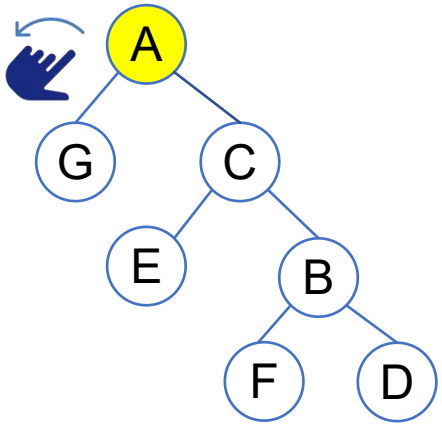
B disconnects with A, while C replaces B's place, to be A's right child.



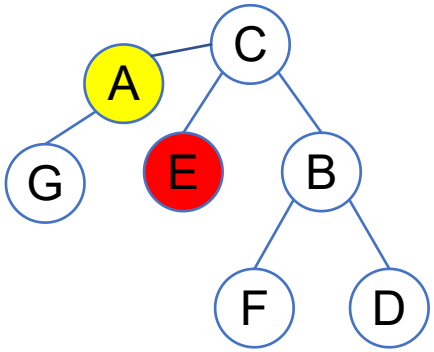
C already has a left child F



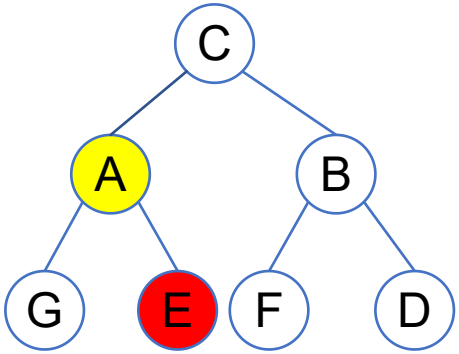
Attach F to be B's left child to avoid conflict and finish rotation at B



Left-rotating A

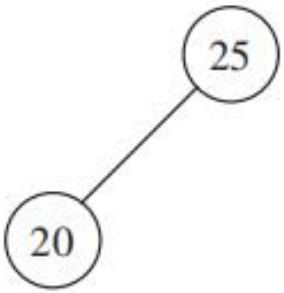


A conflicts with E

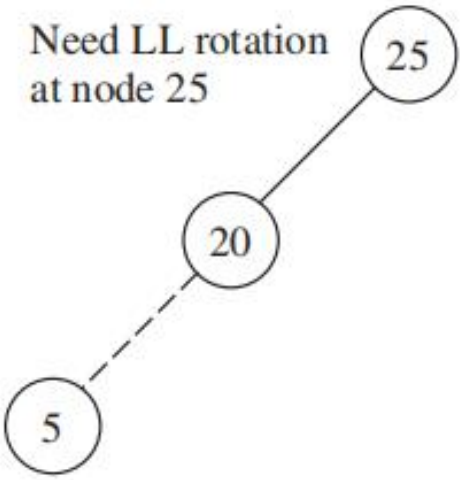


Attach E to be A's right child and finish rotation

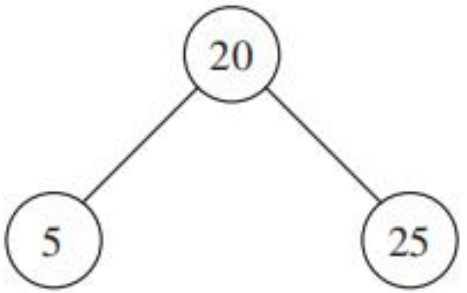
Inserting 25,20,5,34,50,30,10



(a) Insert 25, 20

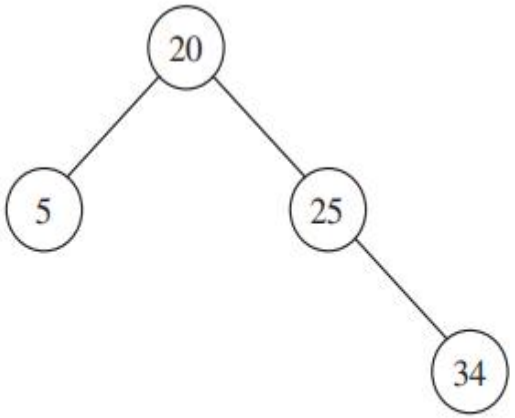


(b) Insert 5

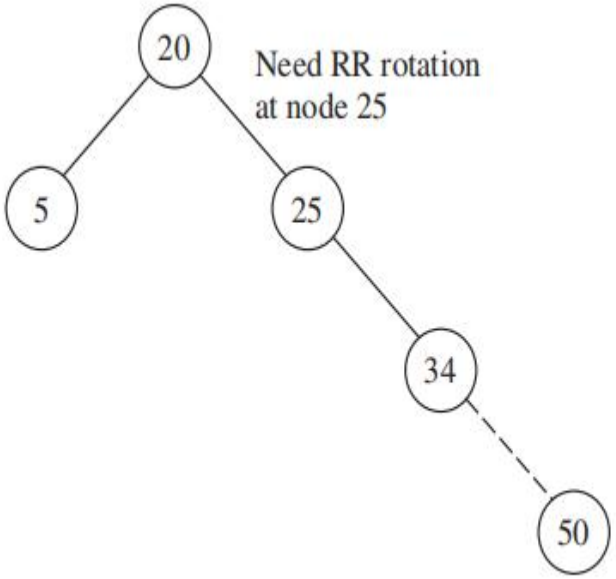


(c) Balanced

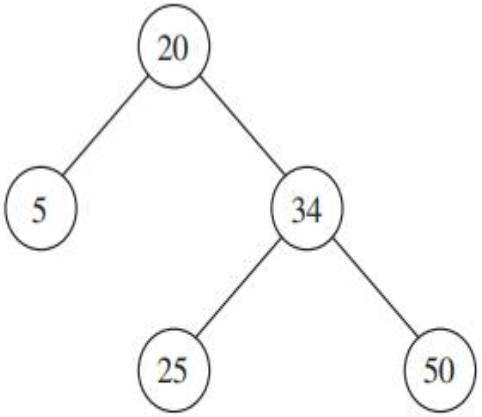
Inserting 25,20,5,34,50,30,10



(d) Insert 34

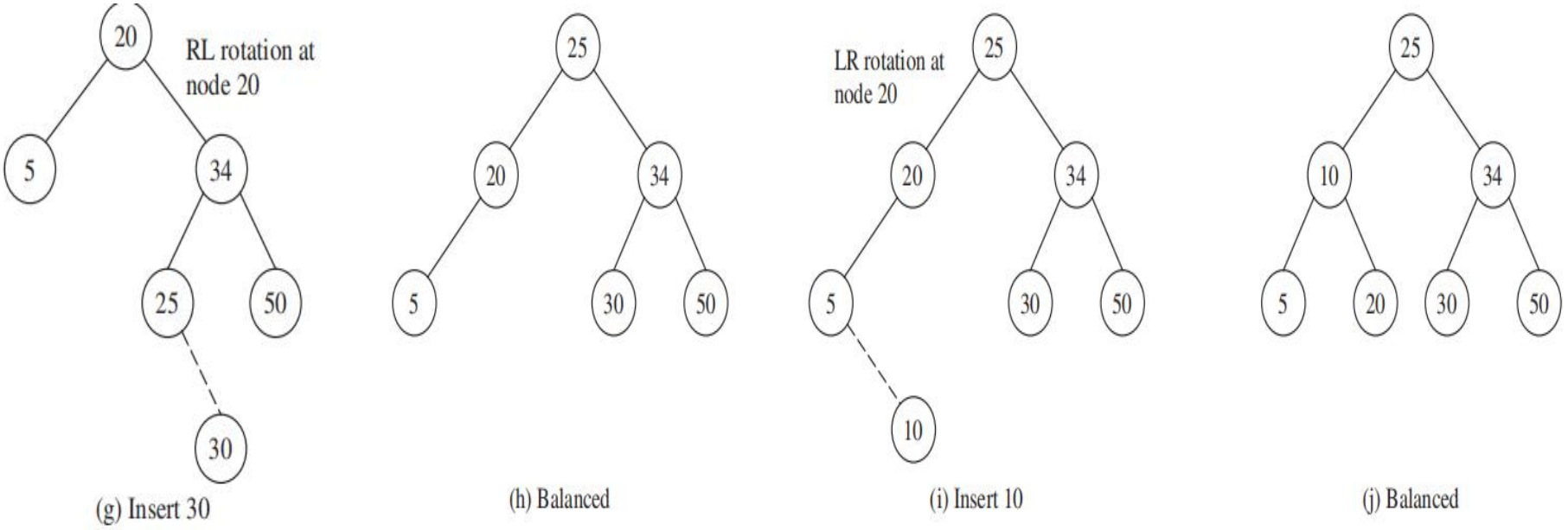


(e) Insert 50

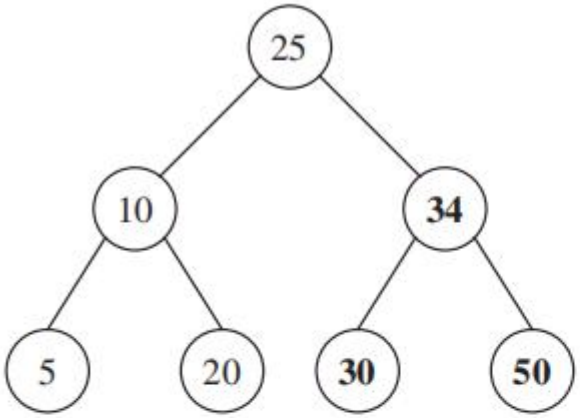


(f) Balanced

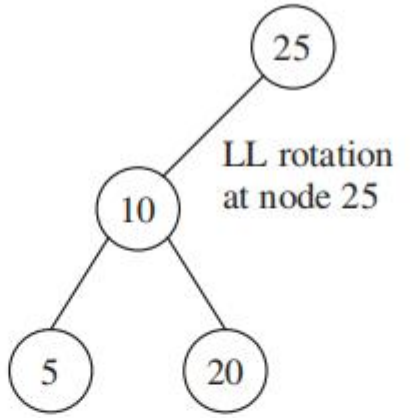
Inserting 25,20,5,34,50,30,10



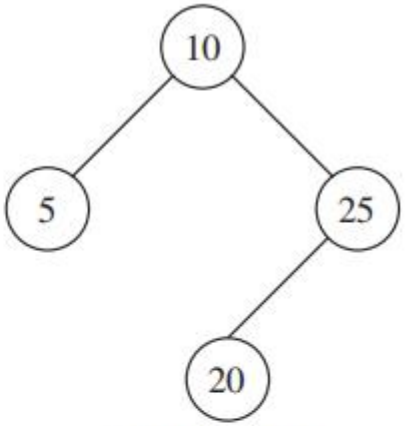
Deleting 34,30,50,5



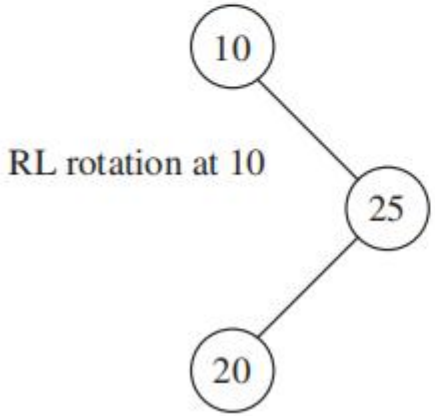
(a) Delete 34, 30, 50



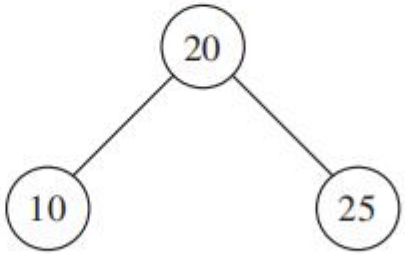
(b) After 34, 30, 50 are deleted



(c) Balanced



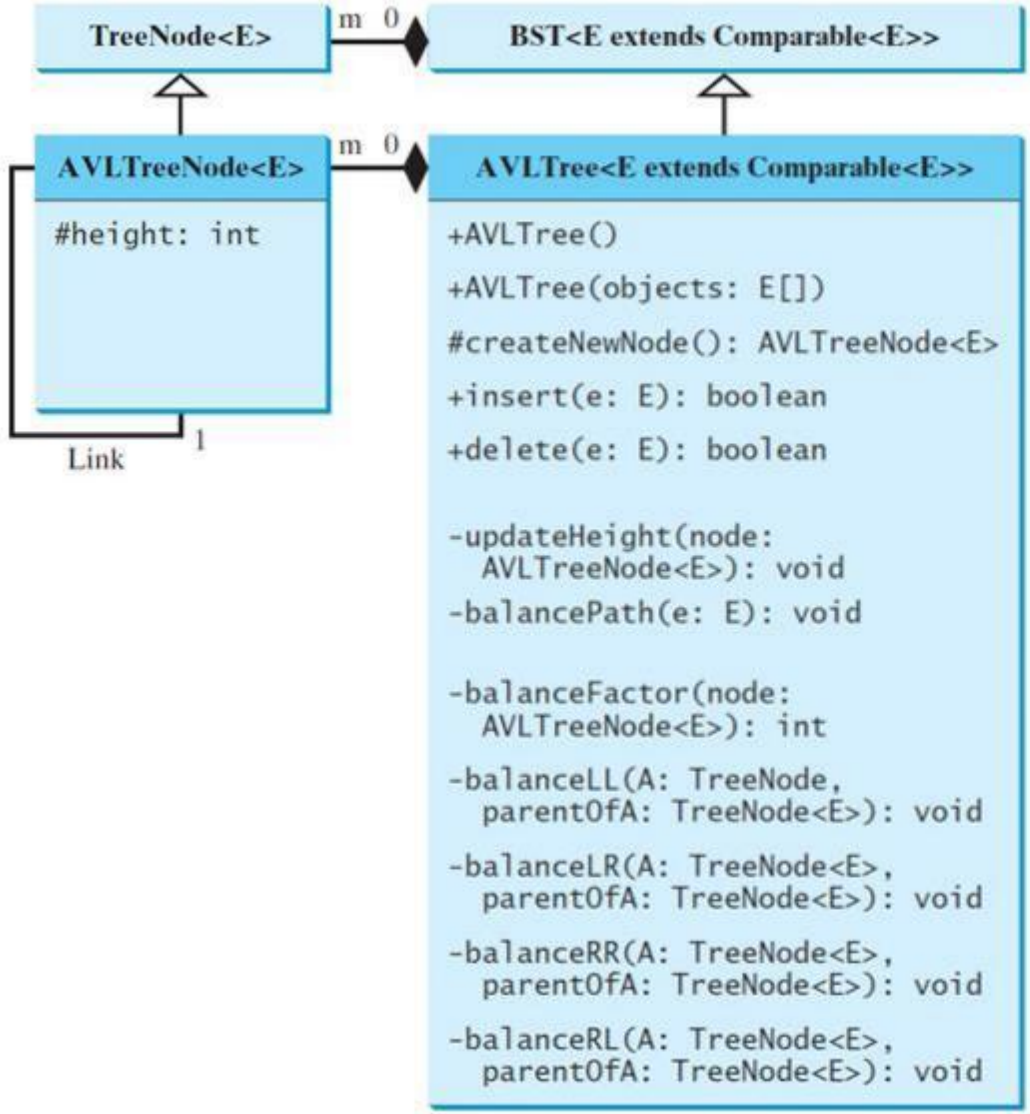
(d) After 5 is deleted



(e) Balanced

Designing Classes for AVL Trees

- An AVL is a binary search tree, so we can define the **`AVLTree`** class to extend the **`BST`** class



```
public class AVLTree<E extends Comparable<E>> extends BST<E> {

  /** AVLTreeNode is TreeNode plus height */
  protected static class AVLTreeNode<E extends Comparable<E>>
      extends BST.TreeNode<E> {
    protected int height = 0; // New data field

    public AVLTreeNode(E o) {
      super(o);
    }
  }

  @Override /** Override createNewNode to create an AVLTreeNode */
  protected AVLTreeNode<E> createNewNode(E e) {
    //(in BST)return new TreeNode<>(e);
    return new AVLTreeNode<E>(e);
  }

  /** Create a default AVL tree */
  public AVLTree() {
  }

  /** Create an AVL tree from an array of objects */
  public AVLTree(E[] objects) {
    super(objects);
  }
```

Need the height (a new data field) for the balance factor calculation

Overriding **createNewNode** method in BST, ensuring the created nodes are AVL tree nodes (with heights).

Also provides 2 types of constructors for AVL trees

```java
@Override /** Insert an element and rebalance if necessary */
public boolean insert(E e) {
  boolean successful = super.insert(e);
  if (!successful)
    return false; // e is already in the tree
  else {
    balancePath(e);
  }
  return true; // e is inserted
}
```

Overriding **insert** method in BST.

Calls the orginal BST insert method (super.insert(e)).

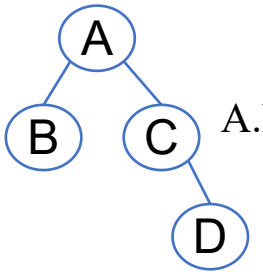
If the insertion is successful, calls the balancePath() for rebalance if needed.

```
function balancePath(e):

    Get the path from the root to the inserted node e
    For each node A in the path (from the inserted node up to the root):
        1. Update the height of A
        2. Find A's parent (null if A is root)
        3. Check the balance factor of A
            case -2 (A is left heavy):
                If LL imbalance:
                    Perform LL rotation
                Else (LR imbalance):
                    Perform LR rotation at A with parentOfA
            case +2 (A is right heavy):
                If RR imbalance:
                    Perform RR rotation
                Else (RL imbalance):
                    Perform RL rotation at A with parentOfA
```

(Detailed code see next page)

```
/** Balance the nodes in the path from the specified node to the root if necessary */
private void balancePath(E e) {

 // Get the path from the root to the inserted node 'e'.
 java.util.ArrayList<TreeNode<E>> path = path(e);
 // For every node on the path (starting from the last one)
 for (int i = path.size() - 1; i >= 0; i--) {
    // Get the current node A (which we are checking for imbalance)
   AVLTreeNode<E> A = (AVLTreeNode<E>)(path.get(i));
   // Update the height of A, because insertion/deletion may change the height
   updateHeight(A);
   // Get the parent of A,which is null when A is the root,otherwise,the previous node
   AVLTreeNode<E> parentOfA = (A == root) ? null :
       (AVLTreeNode<E>)(path.get(i - 1));
   // Calculate the balance factor of A to check if it's unbalanced
   switch(balanceFactor(A)) {
      case -2: // When A is left heavy
         if (balanceFactor((AVLTreeNode<E>)A.left) <= 0) { // LL situation
           balanceLL(A, parentOfA); // Perform LL rotation
         } else { // LR situation
           balanceLR(A, parentOfA); // Perform LR rotation
           }
        break;
      case +2: // When A is right heavy
         if (balanceFactor((AVLTreeNode<E>)A.right) >= 0) { // RR situation
           balanceRR(A, parentOfA); // Perform RR rotation
         } else { // RL situation
           balanceRL(A, parentOfA); // Perform RL rotation
           }
   }
 }
```

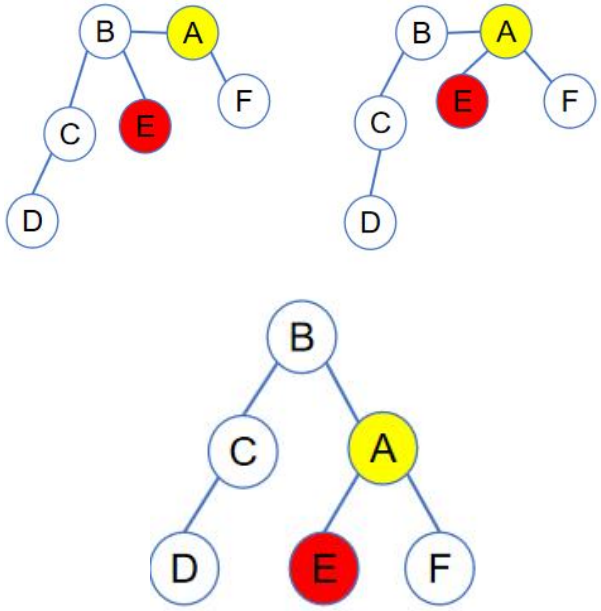
```
/** Update the height of a specified node */
private void updateHeight(AVLTreeNode<E> node) {
  // node is a leaf, its height = 0
  if (node.left == null && node.right == null)
    node.height = 0;
  // node has no left subtree, its height = 1 + the height of its right subtree
  else if (node.left == null)
    node.height = 1 + ((AVLTreeNode<E>)(node.right)).height;
  // node has no right subtree, its height = 1 + the height of its left subtree
  else if (node.right == null)
    node.height = 1 + ((AVLTreeNode<E>)(node.left)).height;
  // node has both left and right subtree, its height = 1 + the height of the taller one
  else
    node.height = 1 + Math.max(
        ((AVLTreeNode<E>)(node.right)).height,
        ((AVLTreeNode<E>)(node.left)).height);
}
```



A.height = 1+2 (C-D>B)

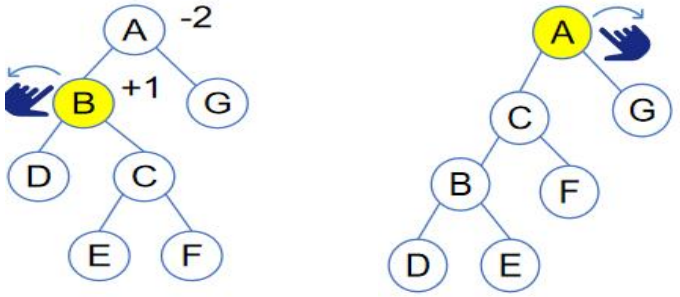
```
/** Return the balance factor of the node */
private int balanceFactor(AVLTreeNode<E> node) {
  // node has no right subtree, balance factor is -node.height
  if (node.right == null)
    return -node.height;
  // node has no left subtree, balance factor is +node.height
  else if (node.left == null)
    return +node.height;
  // node has both left and right subtree, balance factor is right - left
  else
    return ((AVLTreeNode<E>)node.right).height - ((AVLTreeNode<E>)node.left).height;
}
```

```
/** Balance LL */
private void balanceLL(TreeNode<E> A, TreeNode<E> parentOfA) {
  // Define A's left child to be B to facilitate the rotation
  TreeNode<E> B = A.left;
  // B replaces A's position in the tree
  if (A == root) {
    root = B;
  } else { // link B to A's parent (If A has one)
    if (parentOfA.left == A) {
      parentOfA.left = B;
    } else {
      parentOfA.right = B;
    }
  }
  // Attach B's right subtree
  // to A's left subtree
  A.left = B.right;
  // Attach A to be B's right child
  B.right = A;
  // Update height of A and B
  updateHeight((AVLTreeNode<E>)A);
  updateHeight((AVLTreeNode<E>)B)

}
```

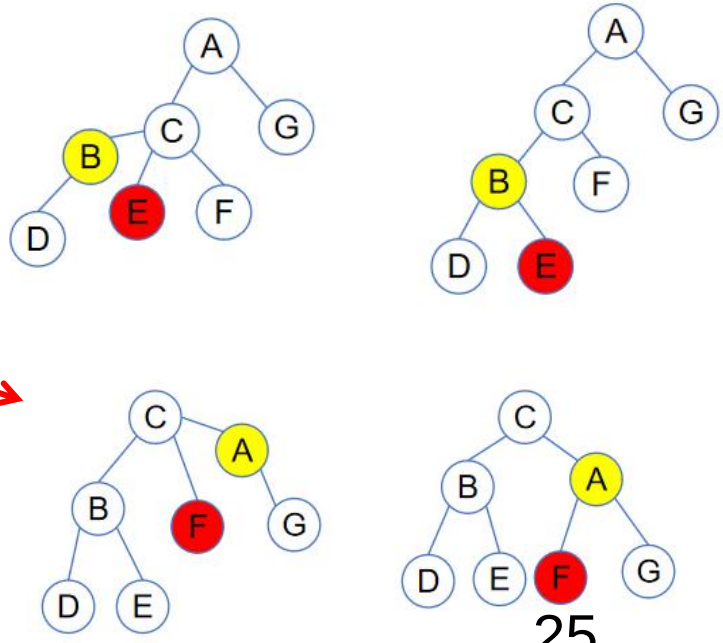


```
/** Balance RR */
private void balanceRR(TreeNode<E> A, TreeNode<E> parentOfA) {
  TreeNode<E> B = A.right; // A is right-heavy and B is right-heavy
  if (A == root) {
    root = B;
  } else {
    if (parentOfA.left == A) {
      parentOfA.left = B;
    } else {
      parentOfA.right = B;
    }
  }
  A.right = B.left;
  B.left = A;
  updateHeight((AVLTreeNode<E>)A);
  updateHeight((AVLTreeNode<E>)B);
}
```

```
/** Balance LR */
private void balanceLR(TreeNode<E> A, TreeNode<E> parentOfA) {
// Define A's left child to be B, and B's right child to be C
TreeNode<E> B = A.left;
TreeNode<E> C = B.right;
// C replaces A's position directly
if (A == root) {
  root = C;
} else { // Link C to A's parent (if A has one)
  if (parentOfA.left == A) {
    parentOfA.left = C;
  } else {
    parentOfA.right = C;
  }
}
// Rearrange the connections
B.right = C.left;
A.left = C.right;
C.left = B;
C.right = A;
// Adjust heights
updateHeight((AVLTreeNode<E>)A);
updateHeight((AVLTreeNode<E>)B);
updateHeight((AVLTreeNode<E>)C);
```



Note that the code directly replaces A with C for efficiency. This achieves the **same final result** as performing two separate rotations — first on B and then on A — as illustrated step-by-step on page 12



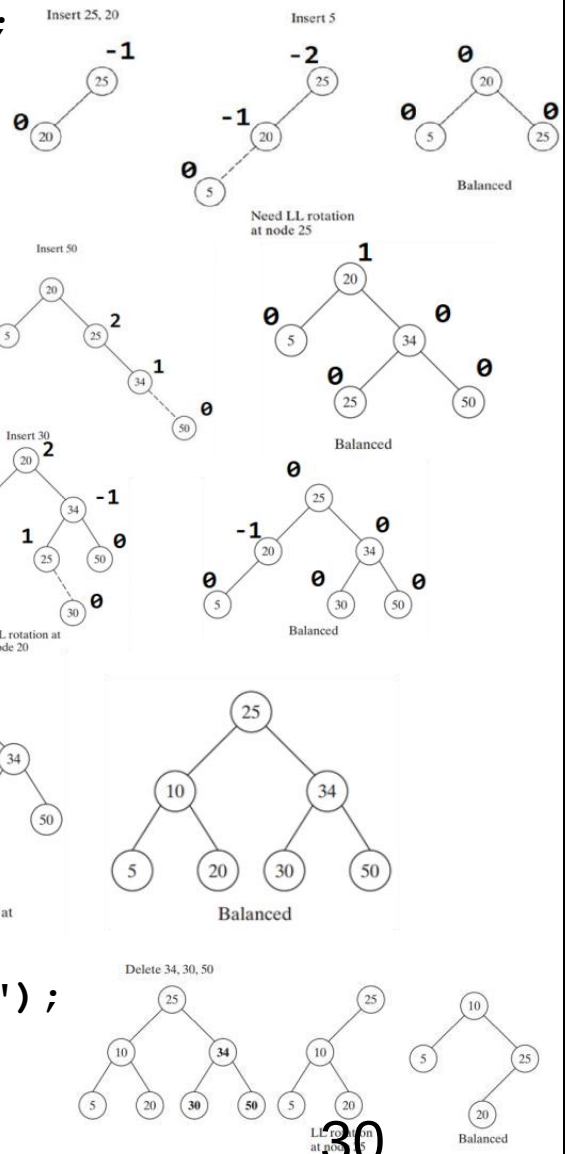
```
/** Balance RL */
private void balanceRL(TreeNode<E> A, TreeNode<E> parentOfA) {
  TreeNode<E> B = A.right;
  TreeNode<E> C = B.left;
  if (A == root) {
    root = C;
  } else {
    if (parentOfA.left == A) {
      parentOfA.left = C;
    } else {
      parentOfA.right = C;
    }
  }
  A.right = C.left;
  B.left = C.right;
  C.left = A;
  C.right = B;

  updateHeight((AVLTreeNode<E>)A);
  updateHeight((AVLTreeNode<E>)B);
  updateHeight((AVLTreeNode<E>)C);
}
```

```
@Override /** Delete an element from the binary tree.
 * Return true if the element is deleted successfully
 * Return false if the element is not in the tree */
public boolean delete(E element) {
  if (root == null)
    return false; // Element is not in the tree
  // Locate the node to be deleted and also locate its parent node
  TreeNode<E> parent = null;
  TreeNode<E> current = root;
  while (current != null) {
    if (element.compareTo(current.element) < 0) {
      parent = current;
      current = current.left;
    } else if (element.compareTo(current.element) > 0) {
      parent = current;
      current = current.right;
    } else
      break; // Element is in the tree pointed by current
  }
  if (current == null)
    return false; // Element is not in the tree
  // Case 1: current has no left children
  if (current.left == null) {
    // Connect the parent with the right child of the current node
    if (parent == null) {
      root = current.right;
    }
```

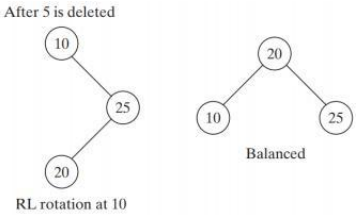
```
    else {
      if (element.compareTo(parent.element) < 0)
        parent.left = current.right;
      else
        parent.right = current.right;
      // Balance the tree if necessary
      balancePath(parent.element);
    }
  } else {
    // Case 2: The current node has a left child
    // Locate the rightmost node in the left subtree of
    // the current node and also its parent
    TreeNode<E> parentOfRightMost = current;
    TreeNode<E> rightMost = current.left;
    while (rightMost.right != null) {
      parentOfRightMost = rightMost;
      rightMost = rightMost.right; // Keep going to the right
    }
    // Replace the element in current by the element in rightMost
    current.element = rightMost.element;
    // Eliminate rightmost node
    if (parentOfRightMost.right == rightMost)
      parentOfRightMost.right = rightMost.left;
    else
      // Special case: parentOfRightMost is current
      parentOfRightMost.left = rightMost.left;
```

```
      // Balance the tree if necessary
      balancePath(parentOfRightMost.element);
    }
    size--;
    return true; // Element deleted
  }
}
```

```
public class TestAVLTree {
  public static void main(String[] args) {
    // Create an AVL tree
    AVLTree<Integer> tree = new AVLTree<>(new Integer[]{25, 20, 5});
    System.out.print("After inserting 25, 20, 5:");
    printTree (tree);

    tree.insert(34);
    tree.insert(50);
    System.out.print("\nAfter inserting 34, 50:");
    printTree (tree);

    tree.insert(30);
    System.out.print("\nAfter inserting 30");
    printTree (tree);

    tree.insert(10);
    System.out.print("\nAfter inserting 10")
    printTree (tree);

    tree.delete (34);
    tree.delete (30);
    tree.delete (50);
    System.out.print("\nAfter removing 34, 30, 50:");
    printTree (tree);
```



```
    tree.delete (5);
    System.out.print("\nAfter removing 5:");
    printTree (tree);

    System.out.print("\nTraverse the elements in the tree: ");
    for (int e: tree) { // inorder: 10 20 25
      System.out.print(e + " ");
    }
  }

  public static void printTree (BST tree) {
    // Traverse tree
    System.out.print("\nPreorder: ");
    tree.preorder();
    System.out.print("\nInorder (sorted): ");
    tree.inorder();
    System.out.print("\nPostorder: ");
    tree.postorder();
    System.out.print("\nThe number of nodes is " + tree.getSize());
    System.out.println();
  }
}
```



AVL Tree Time Complexity Analysis

- In the balancePath(), we have:
  - *for (int i = path.size () - 1; i >= 0; i--)*
- Given the height of the AVL tree is **log n**, the number of nodes on the path (from the newly inserted/deleted one to the root) is **log n**
- The operations, such as updating heights and calculating balance factors (which apply to single nodes), and rotations (which involve a small constant number of nodes), do not depend on the size of the tree => **O(1)**
- So, the time complexity of AVL tree is

$$\mathbf{O(\log n) \times O(1) = O(\log n)}$$

Hashing

Objectives

- To understand what ***hashing*** is and what hashing is used for
- To obtain the hash code for an object and design the hash function to map a key to an index
- To **handle collisions (conflicts) using open addressing**
  - To know the differences among ***linear probing***, ***quadratic probing***, and ***double hashing***
- To **handle collisions using separate chaining**
- To understand the **load factor** and the need for **rehashing**
- To implement **`MyHashMap`** using hashing

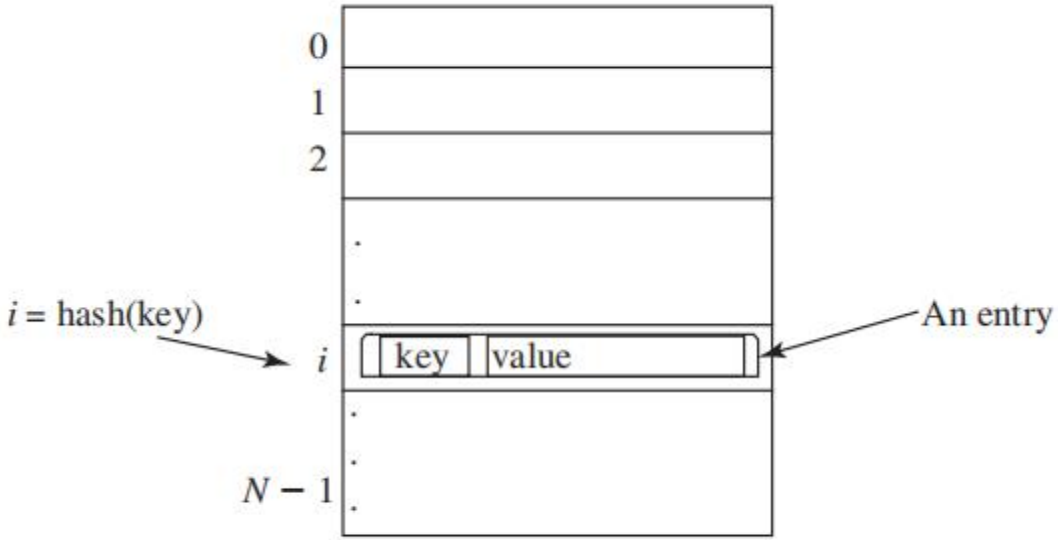
Why hashing?

- Hashing is used for sets and maps:
  - A ***set*** is a container that contains unique elements (no order considered)
  - A ***map*** (***dictionary***, ***hash table***, or ***associative array***) is a data structure that stores data in pairs: **a key and a value.**
  - **Without hashing:** checking whether an element exists in a set or retrieving a value from a map would require searching through the entire collection
  - **With hashing:** A hash function allow us to map each element or key to a specific index, so that we can jump directly to the location where the element or key should be

- **The main advantage of hashing is speed.**
  - We can search, insert, and delete elements in O(1) time on average (no need to search and compare)

How is Hashing implemented?

- Remember **Arrays**: If you know the index of an element in the array, you can retrieve/update the element using the **index** in **O(1)** time
- Follow this idea, we have:
- The array that stores the values is called a ***hash table***
- The function that maps a key to an index in the hash table is called a ***hash function***



Hash function

Hash Functions and Hash Codes

- To eventually get the index, we first need to use the ***hash function*** to convert a search key to an **integer** value called a ***hash code,*** then compresses the hash code into an index to the hash table.

Hash fucntion : Search Key → Hash Code (as integer)
Compressing: Hash Code (as integer) → Index

- Example:
  - Java's root class **`Object`** has a **`hashCode`** method, which returns an integer hash code

```
Object o = new Object();
System.out.println(o.hashCode());
366712642 (decimal representation)
```

  - And then, do the compressing on the hashcode for the index *(because we do not usually have a hash table containing over 300000000 places for storage)*

Hash Codes for Primitive Types

For search keys of the types **byte**, **short**, **char** and **int** are simply coverted into an **int**

E.g., char a = 'A'; int charHash = (int) a;

For a **Float type**, use *Float.floatToIntBits(f)* to convert the float value into its raw integer bit representation, which is then used as the hash code.

System.out.println(Float.floatToIntBits(1.23f)) //1067282596

For a **long type** (64 bits), we generate the hash code by combining the higher 32 bits and lower 32 bits using XOR (exclusive-or operation), so that the result fits into an int

Problem: Direct convert to int will discard (ignore) the higher 32 bits and only keep the lower 32 bits

long value1 = 0xFFFFFFFFABCDEF00L;

long value2 = 0x12345678ABCDEF00L;

(int)value1 → ABCDEF00L

(int)value2 → ABCDEF00L

Solution:

int hashCode = (int)(key ^ (key >> 32));

```
value    : FFFFFFFF ABCDEF00
xorWith  : 00000000 FFFFFFFF
------------------------------
result   : FFFFFFFF 543210FF
```

```
value    : 12345678 ABCDEF00
xorWith  : 00000000 12345678
------------------------------
result   : 12345678 B9F9B978
```

Hash Codes for Primitive Types

- For **double type** (64 bits as well), first convert it to a long value using the Double. doubleToLongBits method, then perform the same XOR.

```
long bits = Double.doubleToLongBits(value);
int hashCode = (int)(bits ^ (bits >> 32));
```

- String's hashCode is calculated by starting from 0 and, for each character in the string, multiplying the current hash value by a fixed prime number (e.g., 31, 33,41, etc) and adding the character's integer value. This process accumulates the effect of each character, making the hash sensitive to both the characters and their order.

```
int hashCode = 0;
for (int i = 0; i < length; i++)
{ hashCode = 31 * hashCode + charAt(i); }
```

Compressing Hash Codes

Hash fucntion : Search Key → Hash Code (as integer)

Compressing: Hash Code (as integer) → Index

- The hash code for a key can be a large integer that is out of the range for the hash table index
  - E.g., a table of size 11 (index 0-10), the hashcode may be a large integer like 366712642
  - We need to scale it down to fit in the index's range
- Assume the index for a hash table is between **0** and **N-1** -- the most common way to scale an integer is

```
h(hashCode)=hashCode % N
```

% is called **modulus operator**, it gets the remainder. E.g., 7%3 =1 => 7/3=2...**1**

N is the size (capacity) of the hash table, which defines how many slots are available for storing elements.Usually choose a **prime number for N (e.g., 11, 13, 31, etc)** to ensure the indices are spread evenly

Handling Collisions

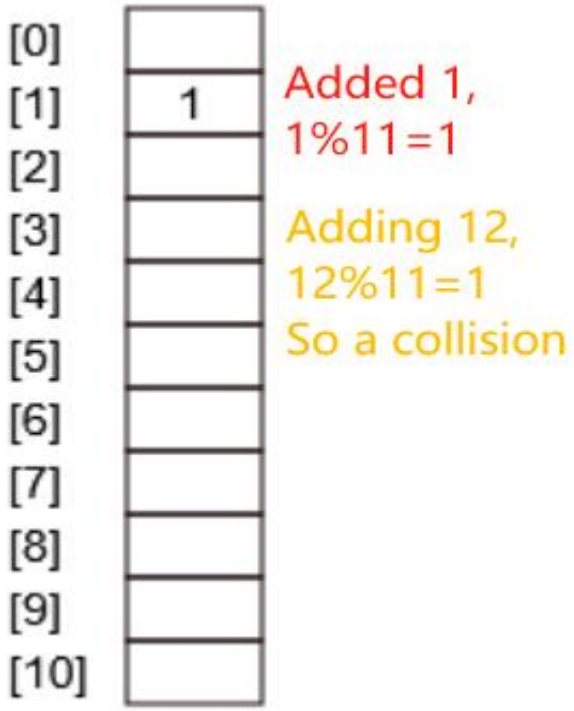
- A ***hashing collision*** (or *collision*) occurs when two different keys are mapped to the same index in a hash table

There are two ways for handling collisions: ***open addressing*** and ***separate chaining***

Open addressing is the process of finding an open location in the hash table in the event of a collision

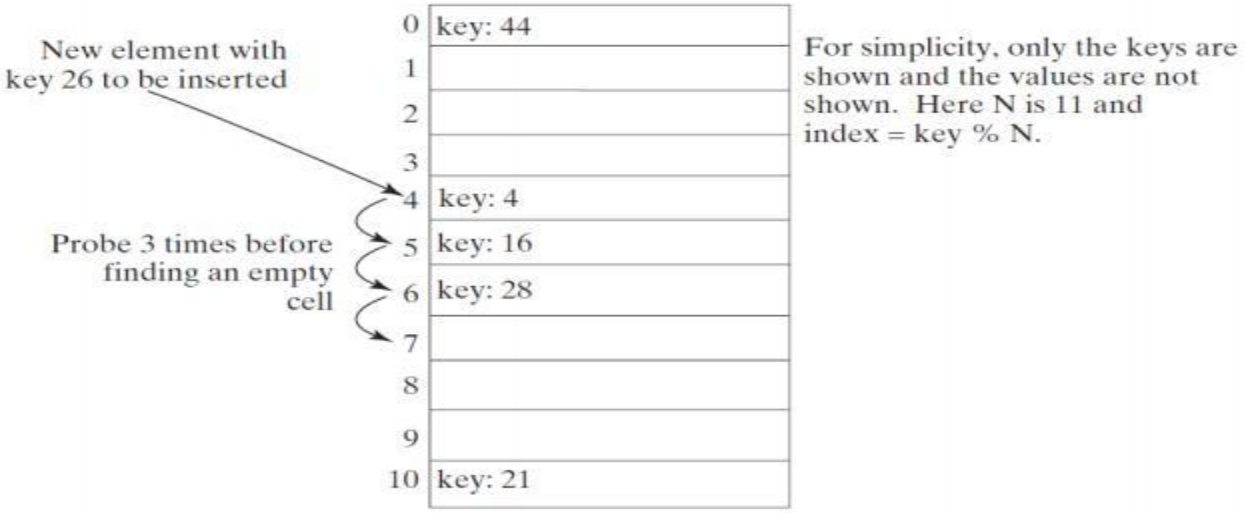
- Open addressing has several variations: ***linear probing***, ***quadratic probing*** and ***double hashing***

Separate chaining places all entries with the same hash index into the same location in a list



Linear Probing

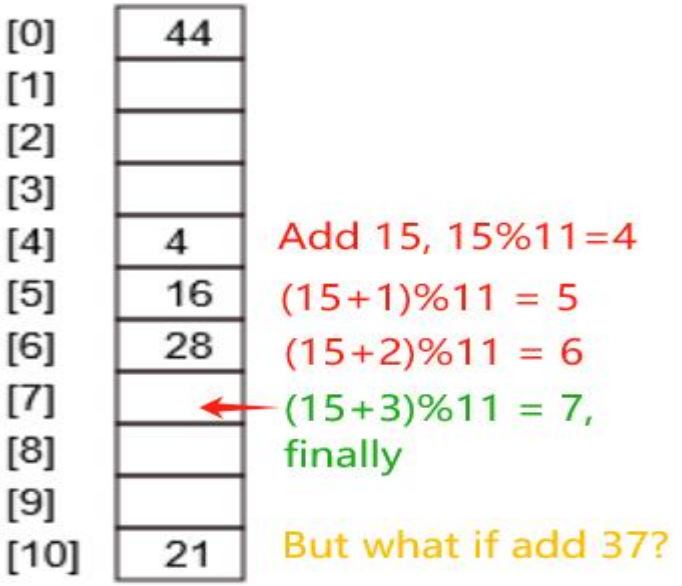
- When a collision occurs during the **insertion** of an entry to a hash table, linear probing finds the next available location sequentially
  - If a collision occurs at **`hashTable[key % N]`**, check whether **`hashTable[(key+1) % N]`** is available
  - If not, check **`hashTable[(key+2) % N]`** and soon, until an available cell is found



Linear Probing

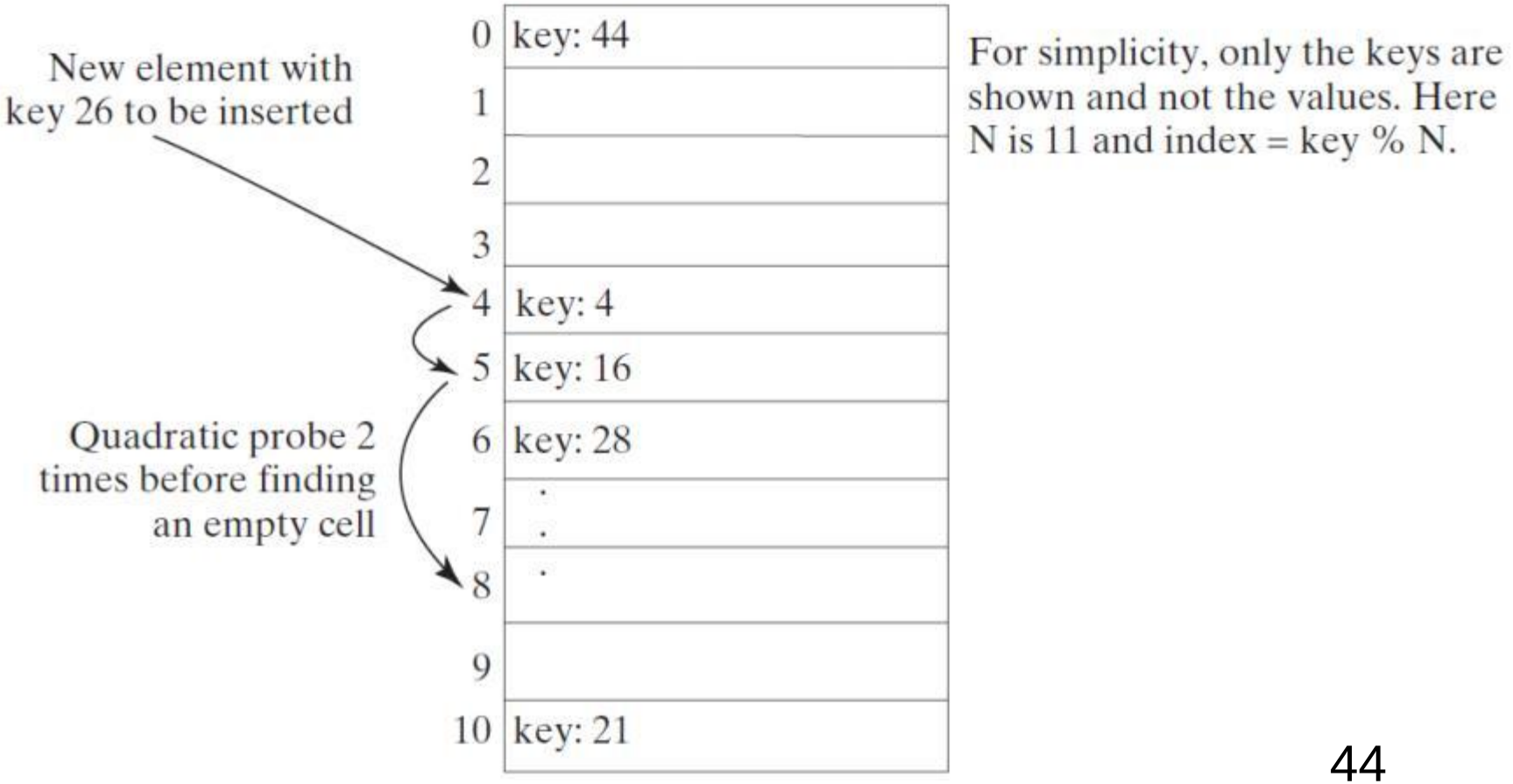
- To **remove** an entry from the hash table, search the entry that matches the key
  - If the entry is found, place a special marker ***marked*** to denote that the entry is deleted, but available for insertion of other values
    - Each cell in the hash table has three possible states: ***occupied***, ***marked***, or ***empty***
      - a marked cell is also available for insertion
- Linear probing tends to cause groups of consecutive cells in the hash table to be occupied -- each group is called a ***cluster***

  - Linear probing causes clustering because when collisions occur, **new elements are always placed into nearby empty slots (k%N, (k+1)%N, (k+2)%N)...)**, causing occupied cells to accumulate and form clusters
  - Each cluster is actually a probe sequence that you must search when retrieving, adding, or removing an entry. This is a big disadvantage of linear probing



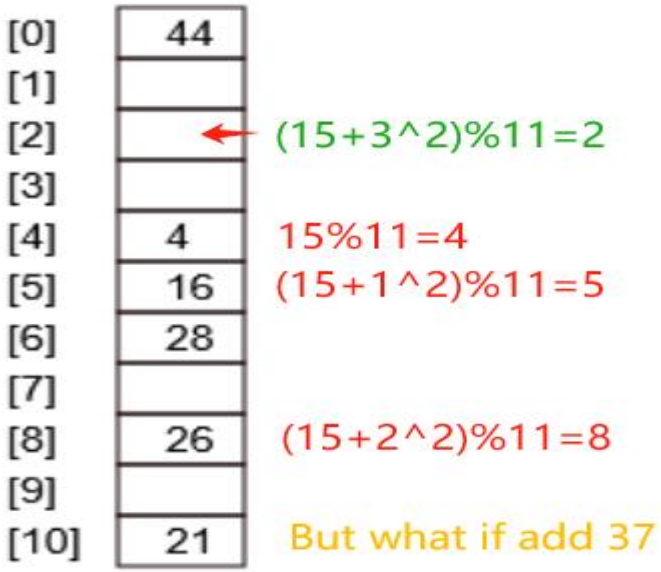
Quadratic Probing

- Quadratic probing looks at the cells at indices **`(key+j²)%N`** for **`j>=0`**, that is, **`key%N`**, **`(key+1)%N`**, **`(key+4)%N`**, **`(key+9)%N`**, **`(key+16)%N`**, and so on



Quadratic Probing

- Quadratic probing works in the same way as linear probing for retrieval, insertion and deletion, except for the change in the search sequence (the 'step' it takes for finding an open location)
- ***Quadratic probing*** can avoid the clustering problem in linear probing **for consecutive keys**
  - It still has its own clustering problem, called ***secondary clustering*** for **keys that collide with the occupied entry** using the same probe sequence



Double Hashing

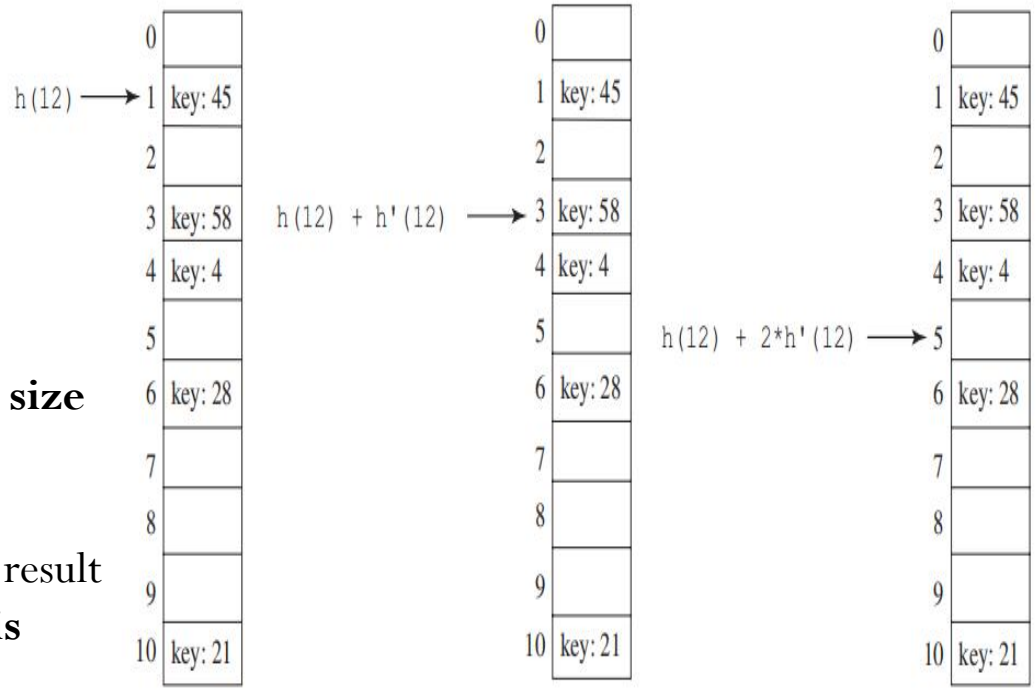
- ***Double hashing*** uses a secondary hash function **`h'(key)`** on the keys to **<u>determine the increments to avoid the clustering</u>** problem
- Double hashing looks at the cells at indices **`(h(key)+j*h'(key))%N`** for **`j>=0`**, that is, **`(h(key)+0*h'(key))%N`**, **`(h(key)+1*h'(key))%N`**, **`(h(key)+2* h'(key))%N`**, **`(h(key)+3*h'(key))%N`**, and so on

h(key) = key % 11;
h'(key) = 7 – key % 7;

h(key) = key % 11 determines the **initial position** in the hash table.

h'(key) = 7 - key % 7 determines the **step size for probing** in case of a collision.

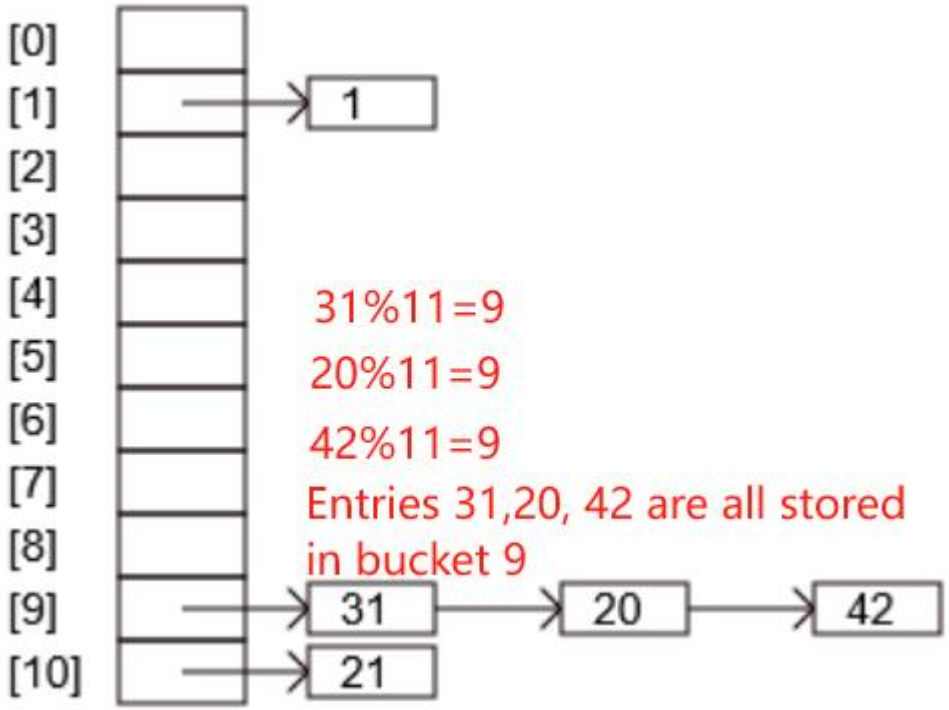
h'(key) can be other formulas as long as its result ensures the **step size is never zero** and **is relatively prime to the table size (N)**
E.g., 6 - key % 6, step sizes can be 1,2,3,4,5,6
And 1,2,3,4,5,6 is relatively prime to 11



Why h'(key) has to be relatively prime to N?
Given h'(key) = 7 – key % 7;
Insert 7,14,28,35 when N is 11 (relatively prime)
Insert 7,14,28,35 when N is 14 (NOT relatively prime)

Handling Collisions Using Separate Chaining

- The ***separate chaining scheme*** places all entries with the same hash index into the same location, rather than finding new locations
  - Each location in the separate chaining scheme is called a ***bucket,*** which can be implemented using an ***array, ArrayList, or LinkedList*** (in the example below)
  - A ***bucket*** is a container that holds multiple entries:

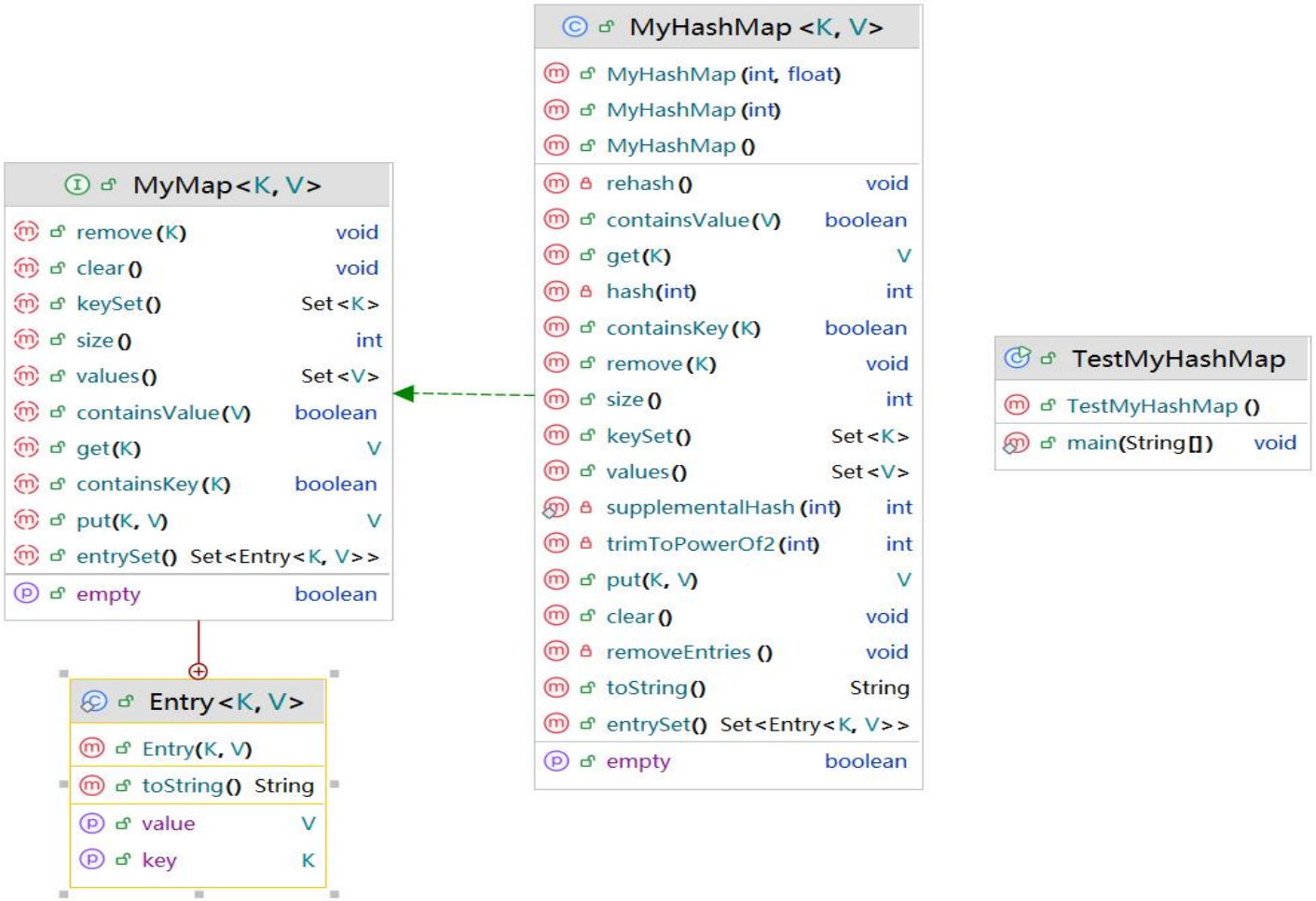


Load Factor and Rehashing

- Load factor λ (lambda) is the ratio of the number of elements to the size of the hash table
  - $\lambda = n/N$
- where **n** denotes the number of elements and **N** the number of locations in the hash table
  - For the open addressing scheme, λ is between **0** and **1**
    - If the hash table is empty, then $\lambda = 0$
    - If the hash table is full, then $\lambda = 1$
  - For the separate chaining scheme, λ can be any value

Load Factor and Rehashing

- As λ increases, the probability of a collision increases
  - Because the ***load factor*** measures how full a hash table is
  - You should maintain the load factor under `0.5` for the open addressing schemes and under `0.9` for the separate chaining scheme
    - In the implementation of the **`java.util.HashMap`** class, the threshold is usually set to be `0.75`
    - If the load factor is **exceeded**, we **increase** the hash table size and **reload** the entries into a new larger hash table -> this is called ***rehashing***
      - We need to change the hash functions, since the hash-table size (N) has been changed
      - To reduce the likelihood of rehashing, since it is costly, you should at least double the hash-table size



- **MyMap<K,V>** is the Interface defining what a map can do, such as put(), get(), containsKey(), etc.
- **MyHashMap<K,V>** implements MyMap, providing the concrete implementation of the abstract methods (e.g., put(), etc) while also implementing the hashing-related operations (e.g., hash(int), rehash(int), etc)
- **Entry<K,V>** is an inner class in MyMap, serving as the data container that is used by the hashing-related operations
- **TestMyHashMap** contains the main method for testing

```java
package org.example;

public interface MyMap<K, V> {  6 usages  1 implementation
  /** Remove all of the entries from this map */
  public void clear();  1 usage  1 implementation

  /** Return true if the specified key is in the map */
  public boolean containsKey(K key);  1 usage  1 implementation

  /** Return true if this map contains the specified value */
  public boolean containsValue(V value);  1 usage  1 implementation

  /** Return a set of entries in the map */
  public java.util.Set<Entry<K, V>> entrySet();  1 usage  1 implementation

  /** Return the first value that matches the specified key */
  public V get(K key);  3 usages  1 implementation

  /** Return true if this map contains no entries */
  public boolean isEmpty();  no usages  1 implementation

  /** Return a set consisting of the keys in this map */
  public java.util.Set<K> keySet();  no usages  1 implementation

  /** Add an entry (key, value) into the map */
  public V put(K key, V value);  6 usages  1 implementation

  /** Remove the entries for the specified key */
  public void remove(K key);  1 usage  1 implementation

```

MyMap Interface defines the common behaviors a map can do

```java
/** Define inner class for Entry */
public static class Entry<K, V> {  23 usages
  K key;  3 usages
  V value;  4 usages

  public Entry(K key, V value) {  1 usage
    this.key = key;
    this.value = value;
  }

  public K getKey() { return key; }

  public V getValue() { return value; }

  @Override
  public String toString() { return "[" + key + ", " + value + "]"; }
}
}
```

MyMap Interface also has the Entry<K,V> inner class, which serves as the data conatiner used by the hashing-related implementations

```java
public class MyHashMap<K, V> implements MyMap<K, V> {  1 usage
  // Define the default hash table size. Must be a power of 2
  private static int DEFAULT_INITIAL_CAPACITY = 4;  1 usage

  // Define the maximum hash table size. 1 << 30 is same as 2^30
  private static int MAXIMUM_CAPACITY = 1 << 30;   3 usages

  // Current hash table capacity. Capacity is a power of 2
  private int capacity;  14 usages

  // Define default load factor
  private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;   2 usages

  // Specify a load factor used in the hash table
  private float loadFactorThreshold;   2 usages

  // The number of entries in the map
  private int size = 0;   7 usages
```

MyHashMap implements MyMap Interface, first defining the core attributes for the hashing operations, such as the default hash table size, max size, etc

```java
@Override /** Return the value that matches the specified key */ 3 usages
public V get(K key) {
  int bucketIndex = hash(key.hashCode());
  if (table[bucketIndex] != null) {
    LinkedList<Entry<K, V>> bucket = table[bucketIndex];
    for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key))
        return entry.getValue();
  }

  return null;
}

@Override /** Return true if this map contains no entries */ no usages
public boolean isEmpty() { return size == 0; }

@Override /** Return a set consisting of the keys in this map */ no usages
public java.util.Set<K> keySet() {
  java.util.Set<K> set = new java.util.HashSet<K>();

  for (int i = 0; i < capacity; i++) {
    if (table[i] != null) {
      LinkedList<Entry<K, V>> bucket = table[i];
      for (Entry<K, V> entry: bucket)
        set.add(entry.getKey());
    }
  }

  return set;
}

@Override /** Add an entry (key, value) into the map */ 6 usages
public V put(K key, V value) {
  if (get(key) != null) { // The key is already in the map
    int bucketIndex = hash(key.hashCode());
    LinkedList<Entry<K, V>> bucket = table[bucketIndex];
    for (Entry<K, V> entry: bucket)
      if (entry.getKey().equals(key)) {
        V oldValue = entry.getValue();
```

MyHashMap implements the methods defined in the MyMap Interface.

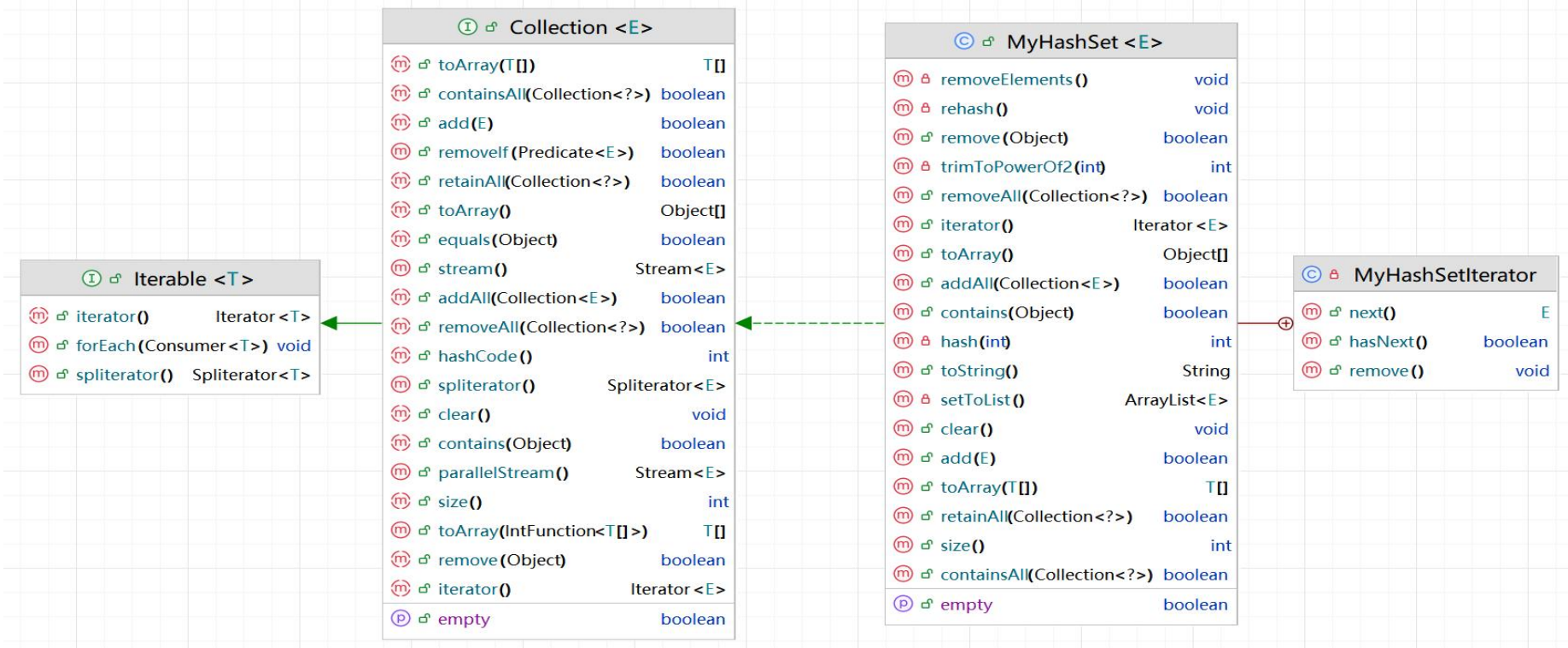
```
201     /** Hash function */
202     private int hash(int hashCode) { return supplementalHash(hashCode) & (capacity - 1); }
205
206     /** Ensure the hashing is evenly distributed */
207     private static int supplementalHash(int h) {  1 usage
208       h ^= (h >>> 20) ^ (h >>> 12);
209       return h ^ (h >>> 7) ^ (h >>> 4);
210     }
```

The **supplementalHash()** attempts to avoid collision by shifting high bits to lower positions (e.g., h>>>20) and mixing them using XOR (^)

```
[ High bits ] [ Middle bits ] [ Low bits ]    ← Original h

            ↓ right shift

[ Zero padding ] [ High bits ] [ Middle bits ]    ← Shifted h

            ↓ XOR with original h

[ Mixed bits ] [ Mixed bits ] [ Mixed bits ]    ← Final result
```

Afterwards, the **hash()** will take the calculated hashcode with the *&(capacity-1)* operation to ensure the final index is always within the valid range of the table size

MyHashMap v.s. MyHashSet



The Major difference:

MyHashSet implements Collection<E> Interface, which indrectly extends the Iterable<E>. This requires MyHashSet to provide an iterator() method for elements traversal.

However, MyHashMap implements the MyMap Interface which is a customized Interface. It does not a direct iterator but provides traversal capabilities indirectly through methods like entrySet(), keySet(), and values(), each returning a set that by default supports iteration for traversal