Binary Search Trees

- Implementing Balancing Operations
  – AVL Trees
  – Red/Black Trees
- Hash Tables
- Reading: 11.5-11.6, Appendix E

Implementing Balancing Operations

- Knowing rotations, we must know how to detect that the tree needs to be rebalanced and which way
- There are 2 ways for tree to become unbalanced
  - By insertion of a node
  - By deletion of a node
- There are two mechanisms for detecting if a rotation is needed and which rotation to perform:
  - AVL Trees
  - Red/Black Trees
- It is best for both to have a parent reference in each child node to backtrack up the tree easily

Implementing Balancing Operations

- AVL trees (after Adel'son-Vel'ski and Landis) keep a balance factor attribute in each node that equals the height of the right sub-tree minus the height of the left sub-tree
- Each time an insertion or deletion occurs:
  - The balance factors must be updated
  - The balance of the tree must be checked from the point of change up to and including the root
- If a node's balance factor is > +1 or < -1, the sub-tree with that node as root must be rebalanced

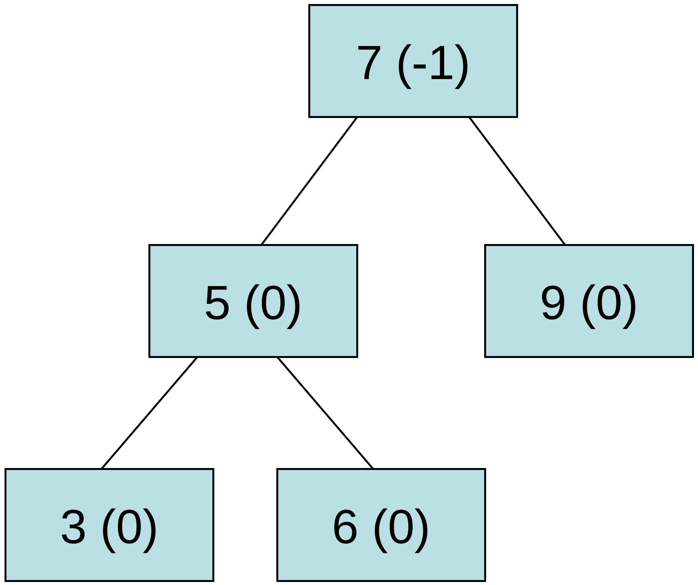
Implementing Balancing Operations

- If the balance factor of a node is -2
  - If the balance factor of its left child is -1 or zero*
    - Perform a right rotation
  - If the balance factor of its left child is +1
    - Perform a leftright rotation
- If the balance factor of a node is +2
  - If the balance factor of its right child is +1 or zero*
    - Perform a left rotation
  - If the balance factor of its right child is -1
    - Perform a rightleft rotation

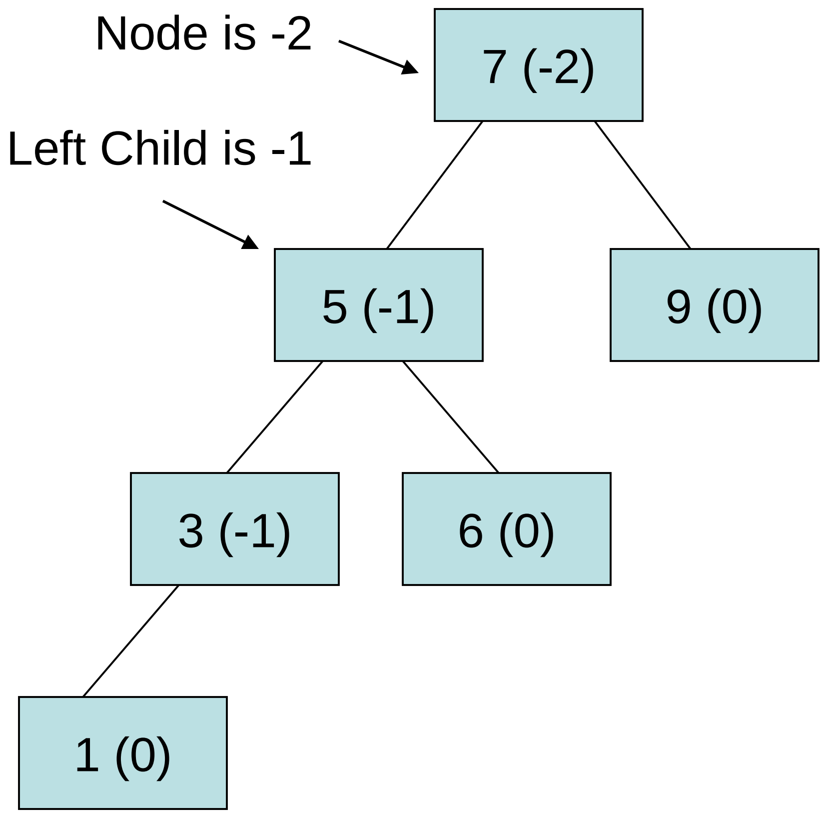
Note: Error in L&C on page 315 → zero is not mentioned,
but zero can happen in some cases of remove operations

AVL Tree Right Rotation

Initial Tree

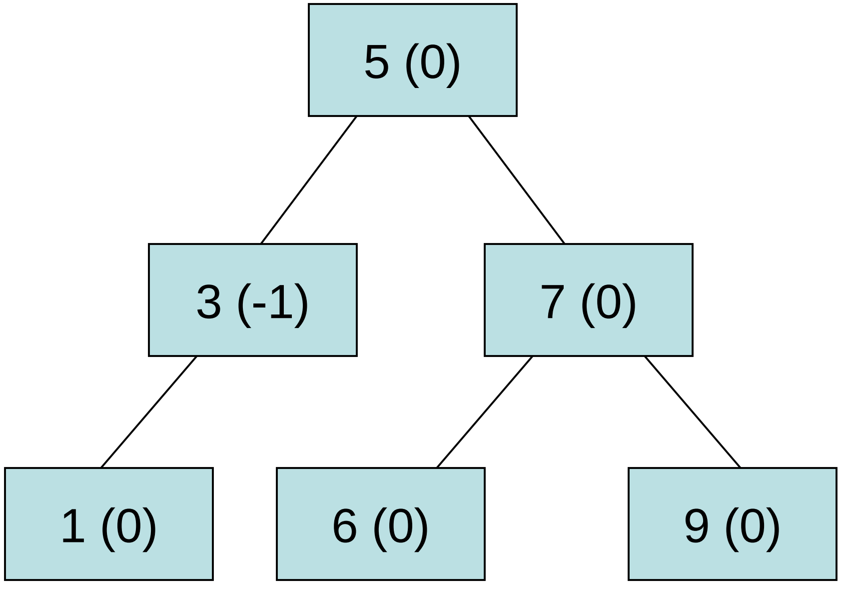


After Add 1

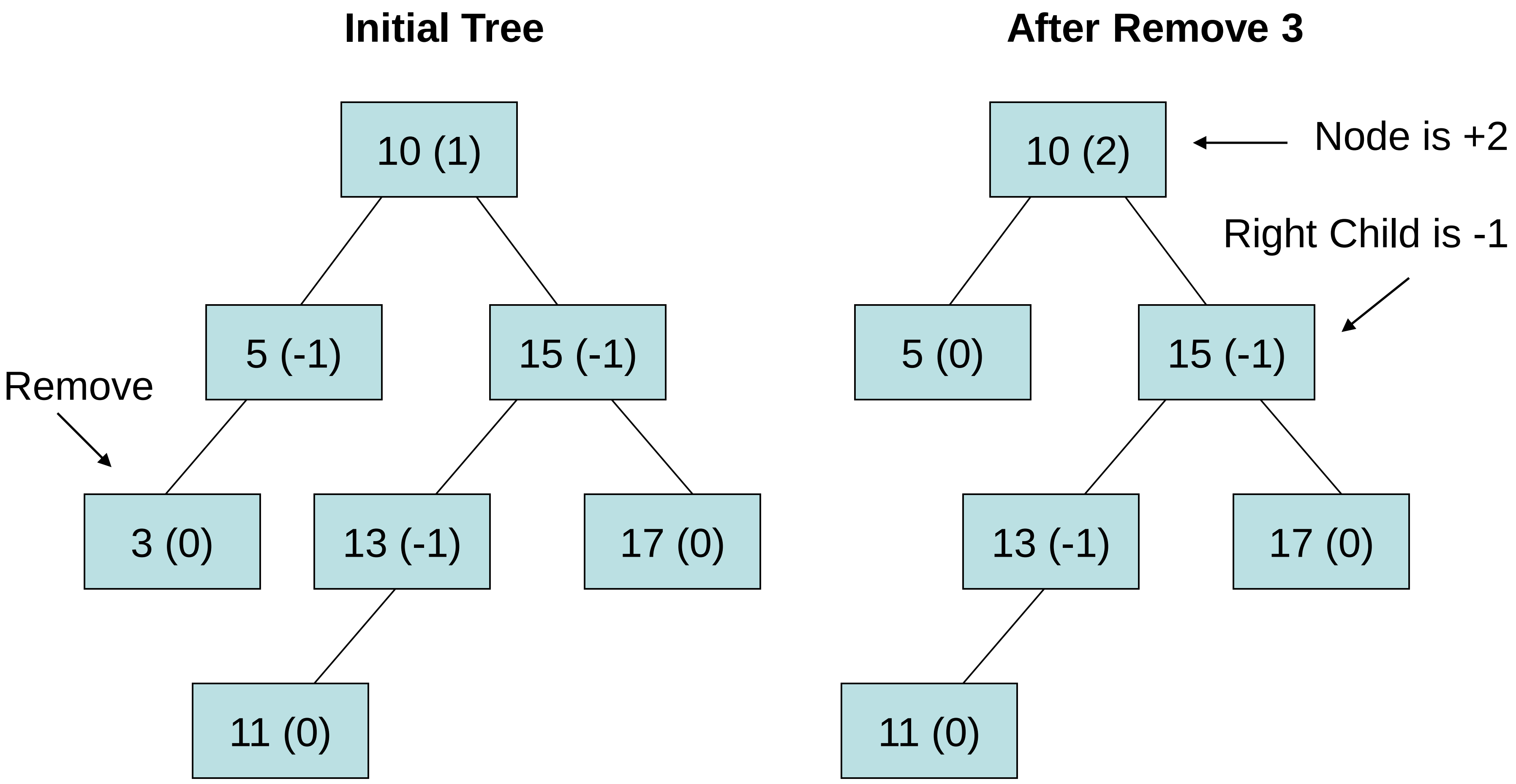


AVL Tree Right Rotation

After Right Rotation

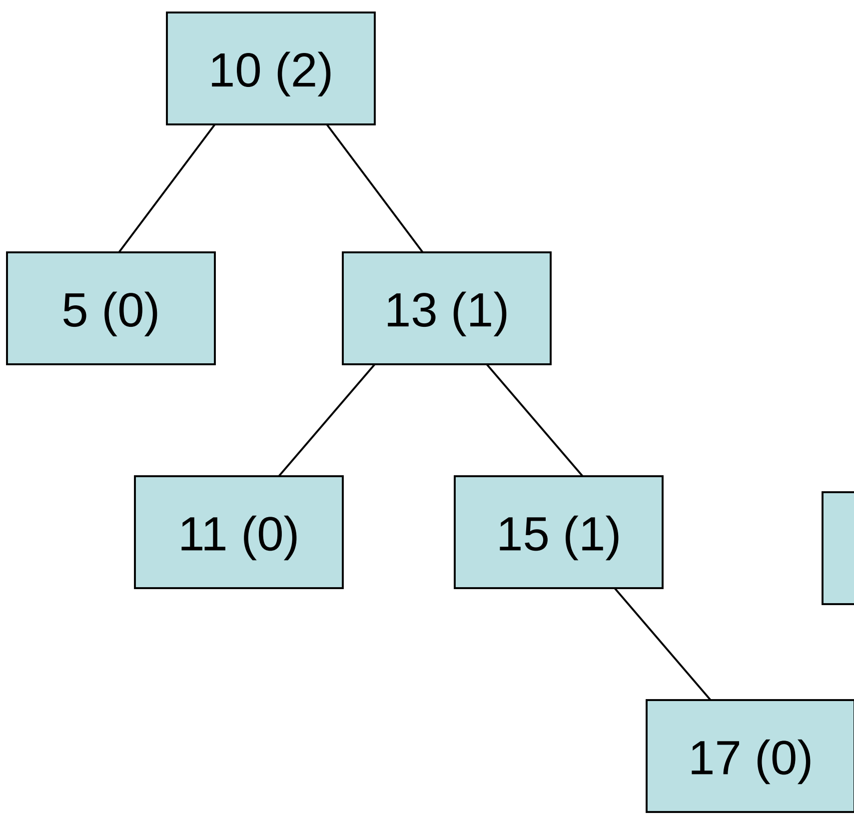


AVL Tree Rightleft Rotation

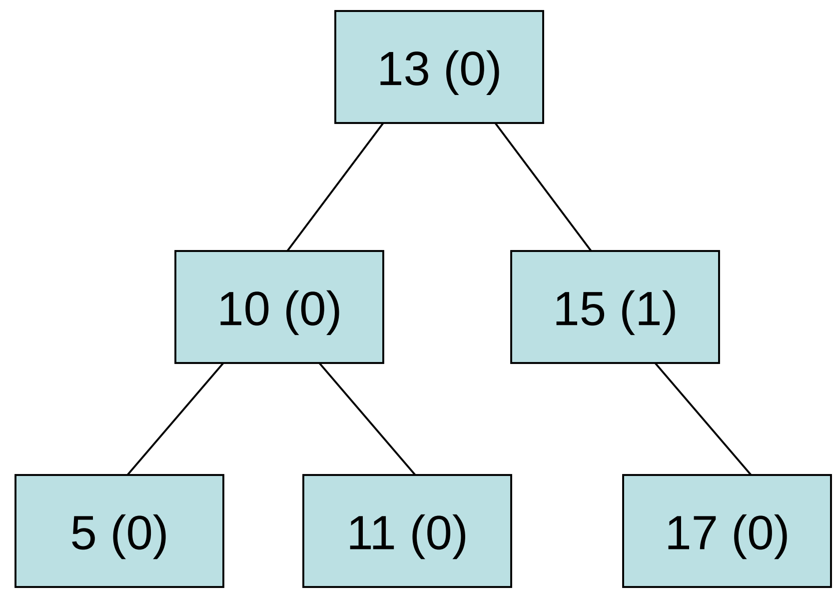


AVL Tree Rightleft Rotation

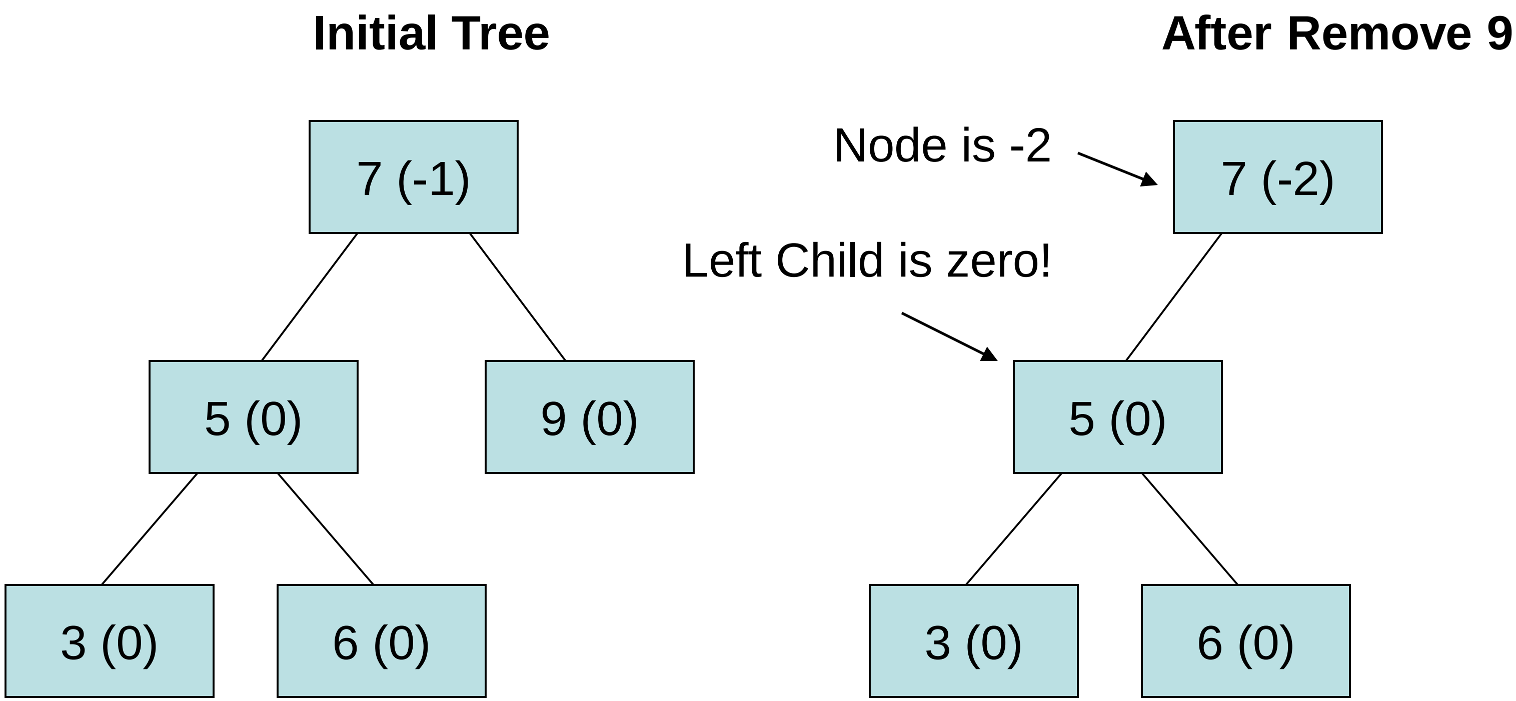
After Right Rotation



After Left Rotation



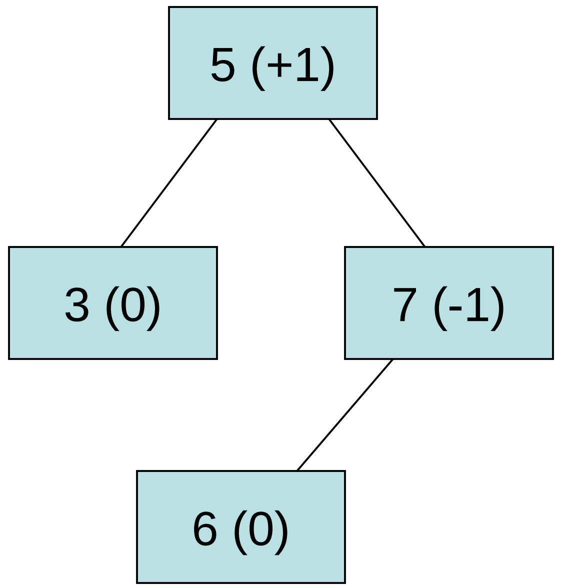
AVL Tree Right Rotation



Note: This is the case that the L&C text misses in its discussion and examples

AVL Tree Right Rotation

After Right Rotation



Note: This is the case that the L&C text misses in its discussion and examples

Red/Black Trees

- Red/Black Trees (developed by Bayer and extended by Guibas and Sedgewick) keep a "color" red or black for each node in the tree
- This approach is used in the Java class library binary search tree classes
- The maximum height of a Red/Black tree is roughly 2*log n (not as well controlled as an AVL tree), but the height is still O(log n)
- The rules for insert, delete, and rebalance are more complicated than our textbook section covers, so we won't study this technique

Hash Tables

- Binary search trees can be used for both sorting in O(N log N) time and searching in O(log N) time
- However, if you have an application that requires only searching but not sorting, a hash table is a data structure that provides O(1) performance for searching
- A hash table optimizes time performance at the expense of using more memory

Hash Tables

- Some programming languages include direct language support for hash tables - such as PERL (hashes) and Python (dictionaries)
- Java does not - hash tables are implemented in library classes such as HashMap<K,V> or you can write a class yourself if necessary
- You should understand how to implement a hash table in any language such as C where there is no language or library support

Hash Tables

- The base of a hash table can be an array
- A hash function operates on the search key to produce an integer index into the array
- All data elements whose hash of their keys result in the same index are stored starting at that position in the array
- Essentially, a hash table is structured like a comb with a long “spine” and short “teeth”
- Only a few elements start at the same index

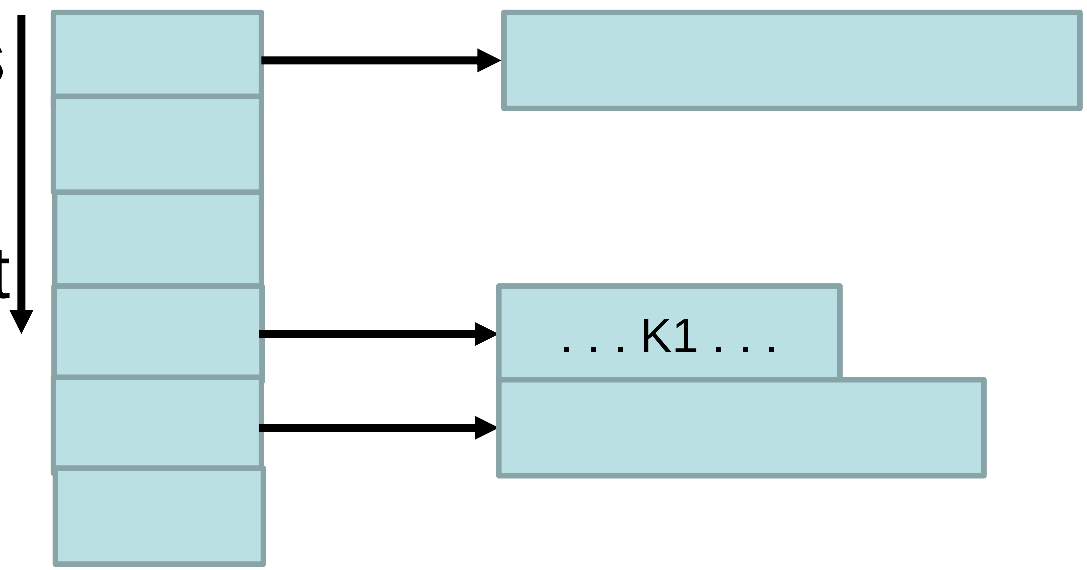
Hash Tables

- In Java, one reasonable way to implement a hash table is with a parameterized class that encapsulates an array of ArrayList<K> objects

ArrayList<K> hashTable = new ArrayList<K>();

Hash Value hashTable ArrayList(s)

for key K1 as
an index to
one ArrayList



Hash Tables

- To add an element, the add method:
  - Hashes the key to get an index into the table
  - Instantiates an ArrayList (if needed)
  - Adds the key object to that ArrayList
- To find an element, the find method:
  - Hashes the key to get an index into the table
  - If there is no ArrayList there, it returns not found
  - It searches the ArrayList to find the Key
  - If the Key is not there, it returns not found

Hash Tables

- The hash table lookup is faster because a search quickly finds the correct ArrayList which is short for searching relative to N
- If as you add elements to the Hash Table the ArrayLists get too large, you need to expand the capacity of the array as we did previously for stacks and queues and then rehash every key into the new larger array with new ArrayLists

Hash Functions

- The hashing function is critical for random distribution of the key elements
- For example, if your keys are telephone numbers in Massachusetts, using the first 3 digits (the area codes) would be a poor choice because there are only a few area codes in use in the state (“clustering”)
- The last 4 digits would be a better choice, but is still not ideal because the array size must be 10,000 (not a prime number)

Hash Functions

- The mathematics don’t work well unless the array size is a prime number (P)
- A better choice would be to select a prime number for the size of the array and use the whole phone number modulo the array size to get an array index from 0 to P-1
- If you need to expand the array capacity, you can’t just double the size of the array because that would not be a prime number

Hash Functions

- Implementing a good hash function is not easy
- There is a hashCode() method in the Object class that is the parent class of all classes
- Although it is not considered to generate the best hash codes, you should probably use it if you don’t know how to generate a good one yourself