Hash tables

- Performance of in-place solutions varies with how full the array is (called the load factor)
  - Small load factor (50% or less), can expect constant time for a search in most cases.
  - Load factor of 80-85% or more can lead to excessive collisions
    - Implementations may trigger a rebuilding of the hash table into a larger array in these cases.

```java
import java.util.Hashtable;
import java.util.Enumeration;

public class HashtableExample {

 public static void main(String[] args) {

   Enumeration names;
   String key;

   // Creating a Hashtable
   Hashtable<String, String> hashtable =
               new Hashtable<String, String>();

   // Adding Key and Value pairs to Hashtable
   hashtable.put("Key1","Chaitanya");
   hashtable.put("Key2","Ajeet");
   hashtable.put("Key3","Peter");
   hashtable.put("Key4","Ricky");
   hashtable.put("Key5","Mona");

   names = hashtable.keys();
   while(names.hasMoreElements()) {
      key = (String) names.nextElement();
      System.out.println("Key: " +key+ " & Value: " +
      hashtable.get(key));
   }
 }
}
```

Code from https://beginnersbook.com/2014/07/hashtable-in-java-with-example/, January 16, 2020

```java
import java.util.HashMap;
import java.util.Map;
import java.util.Iterator;
import java.util.Set;
public class Details {

   public static void main(String args[]) {

      /* This is how to declare HashMap */
      HashMap<Integer, String> hmap = new HashMap<Integer, String>();

      /*Adding elements to HashMap*/
      hmap.put(12, "Chaitanya");
      hmap.put(2, "Rahul");
      hmap.put(7, "Singh");
      hmap.put(49, "Ajeet");
      hmap.put(3, "Anuj");

      /* Display content using Iterator*/
      Set set = hmap.entrySet();
      Iterator iterator = set.iterator();
      while(iterator.hasNext()) {
         Map.Entry mentry = (Map.Entry)iterator.next();
         System.out.print("key is: "+ mentry.getKey() + " & Value is: ");
         System.out.println(mentry.getValue());
      }
```


Code from https://beginnersbook.com/2013/12/hashmap-in-java-with-example/ , January 16, 2020

Dynamically-growing structures

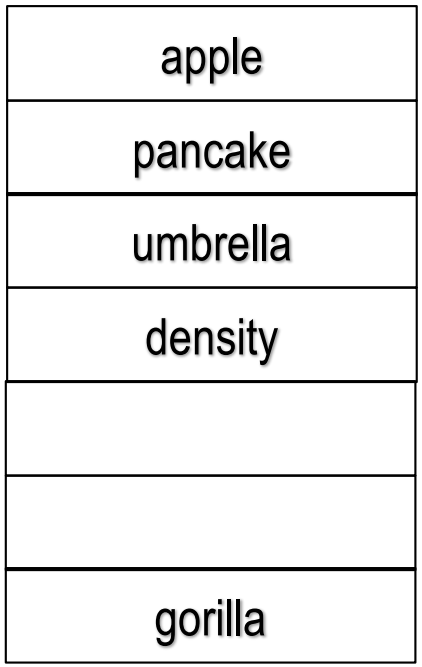
- Arrays and hash tables start with a fixed size.
- The structure becomes dynamic if you take the following steps when you try to add beyond the fixed size:
  - Allocate / create a new array or table that is bigger than the current one
  - Copy all items from the current array or table into the new one
    - This is the costly step
  - Add in the new data item

  - Common trick: when creating a new array or table, make it twice as big
    - Not reallocating the table all of the time to grow by 1 or 2
    - Have at most half of the space in the new array or table as unused

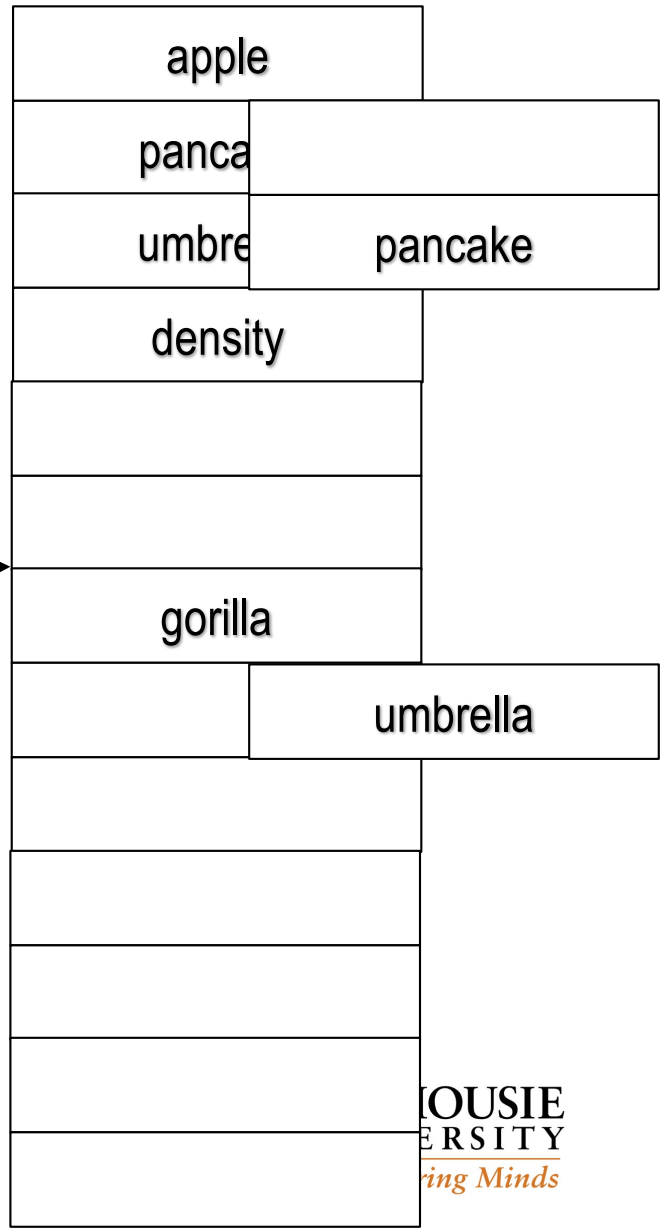
Dynamically-growing structures

- **Given a hash table, why don't we just copy over all the data directly?**

The hash function for the larger table may not put each value in the same place.



??



Data structures without a fixed size

Basics

- **Typically create a class to store a single item**

```
Class Node {
  Node Some_references_to_other_nodes;

  SomeClass value;
}
```

  - **Note that the class includes an attribute that refers back to the same class.**

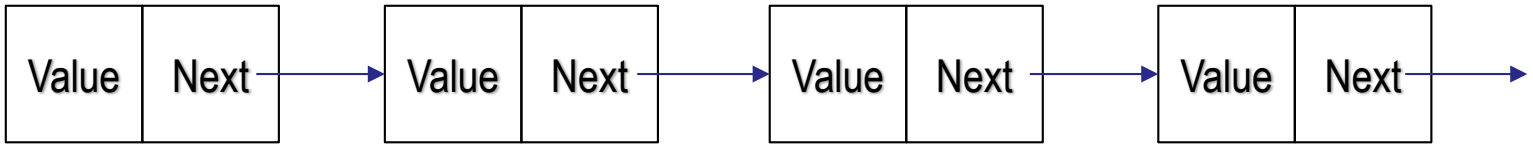
- **Different data structures arise in how we link these single items together**

Linked List

- Informally, it's a chain of data values
  - Explicit storage of the sequence of values
    - Each value points to the "next" value
- Variants
  - Sorted vs unsorted
  - Singly-connected – only forward pointers
  - Doubly-connected – forward and backward pointers
  - Circular –end point back to the front
    - Can be singly- or doubly- connected
- Linear progression through the list elements

Linked List

- Can grow arbitrarily large
- Has small additional cost for storing the order information
- Linear traversal incurs an efficiency loss for searching




```
Class Node {
  Node next;
  int value;

  public void add_after ( int new_value ) {
    Node new_node = new Node();
    new_node.value = new_value;
    new_node.next = next;
    next = new_node;
  }
}
```

Partial linked list node class, as a sample.

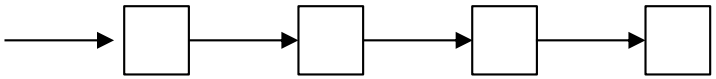
Doubly connected?

Linked List

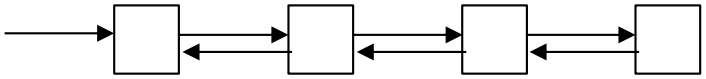
- A very common data structure, so learn how to implement them
  - Add
  - Remove
  - Search
  - Traverse

Variants of linked lists

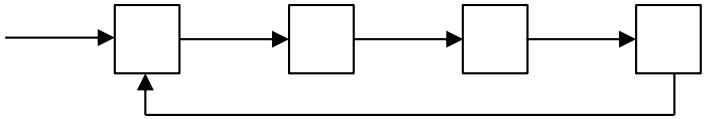
- **Design differences**
  - Singly-connected linked list (forward only)



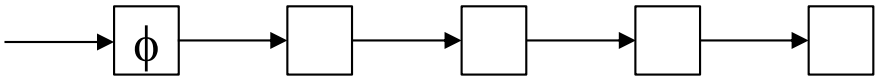
  - Doubly-connected linked list (forward and backward)



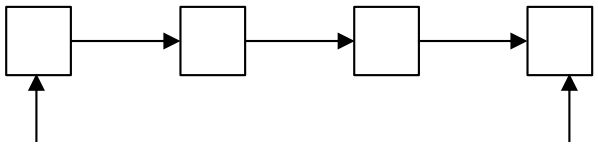
  - Circular linked list (can start anywhere)



- **Implementation differences**
  - Sentinel or dummy node as the start (list never empty)



  - Track both ends of the list (append is quick)

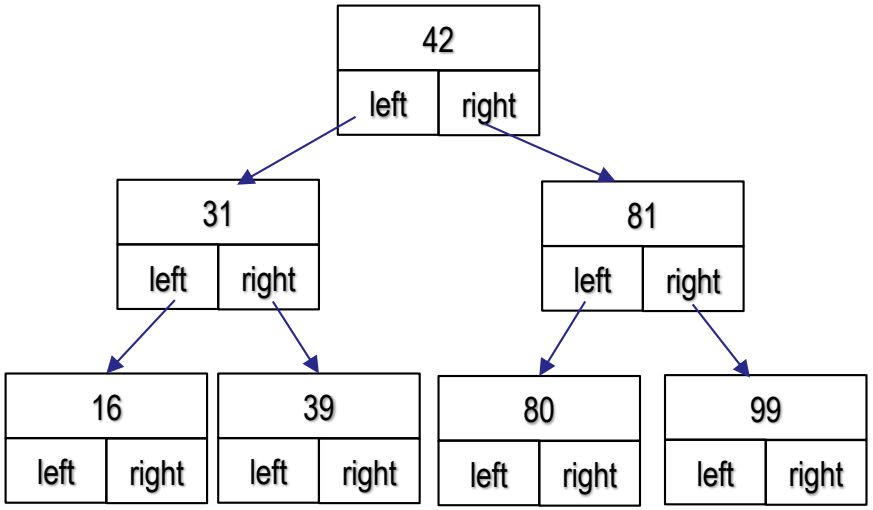


Binary search tree

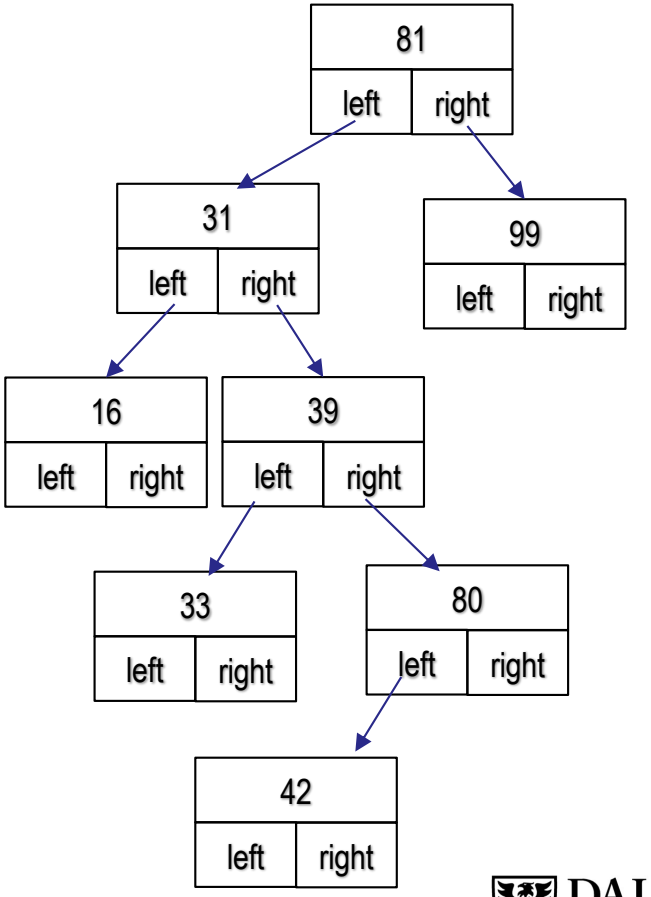
- A sorted organization of data values to make searching quick.
  - Parallels the checks of a binary search in a data structure.
- Stored as a set of nodes.
  - Simplistically, each node stores
    - A value
    - A reference to the sorted data before the value
    - A reference to the sorted data after the value
  - More precisely, each node u stores a reference to one node whose value precedes the value in u and a reference to one node whose value succeeds the value in u
    - Not necessarily the immediate predecessor or successor
    - You don't reference a node that any other part of the data structure references

Binary search tree

- **Balanced tree**



- **Unbalanced tree**



DALHOUSIE UNIVERSITY
Inspiring Minds

Binary search tree

- Locating an element in a binary search tree involves descending the levels
  - Great search time on balanced trees
  - Can be linear time on severely unbalanced trees
- Often use recursion when working with binary trees, but that's a convenience rather than a necessity.

Java type for a binary tree

```
public class BinaryTreeNode {
    private BinaryTreeNode left;
    private BinaryTreeNode right;
    private BinaryTreeNode parent;  // optional

    private SomeDataType  value;
}

public class BinaryTree {
    private BinaryTreeNode root;
}
```

Finding a value in a binary search tree

```
public boolean find ( int value ) {
    BinaryTreeNode current = root;  // root from the class

    // Walk a path from the root to where the node should be
    while ((current != null) && (current.value != value)) {
        if (value < current.value) {
            current = current.left;
        } else {
            current = current.right;
        }
    }
    if (current == null) {
        // Went off the end of the tree, so not found
        return false;
    } else {
        return true;
    }
}
```

Common variation to track where you were

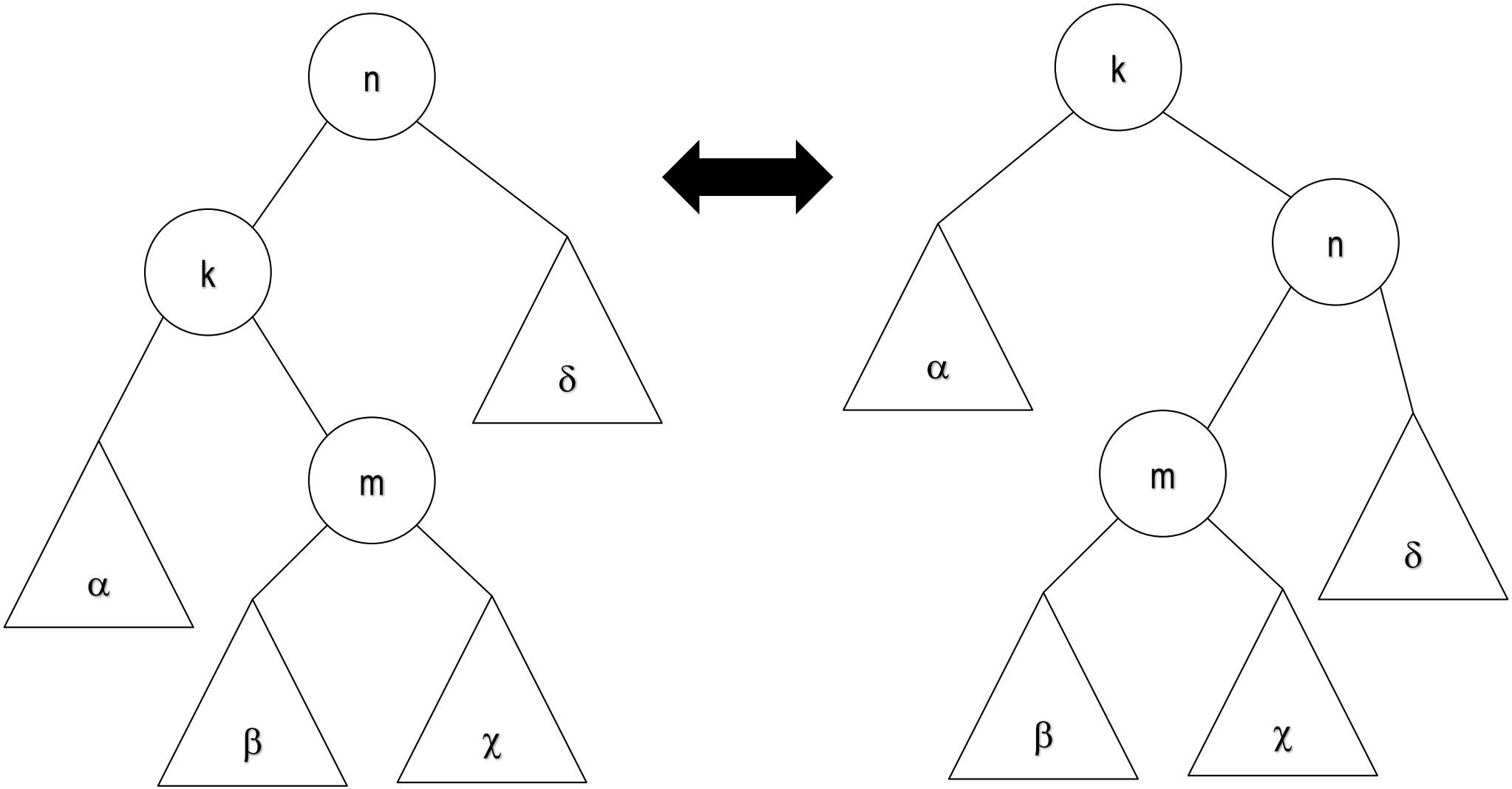
```
public boolean find ( int value ) {
    BinaryTreeNode current = root;  // root from the class
    BinaryTreeNode previous = null;

     // Walk a path from the root to where the node should be
    while ((current != null) && (current.value != value)) {
        previous = current;
        if (value < current.value) {
            current = current.left;
        } else {
            current = current.right;
        }
    }
    if (current == null) {
        // Went off the end of the tree, so not found
        return false;
    } else {
        return true;
    }
}
```

Binary search tree variants

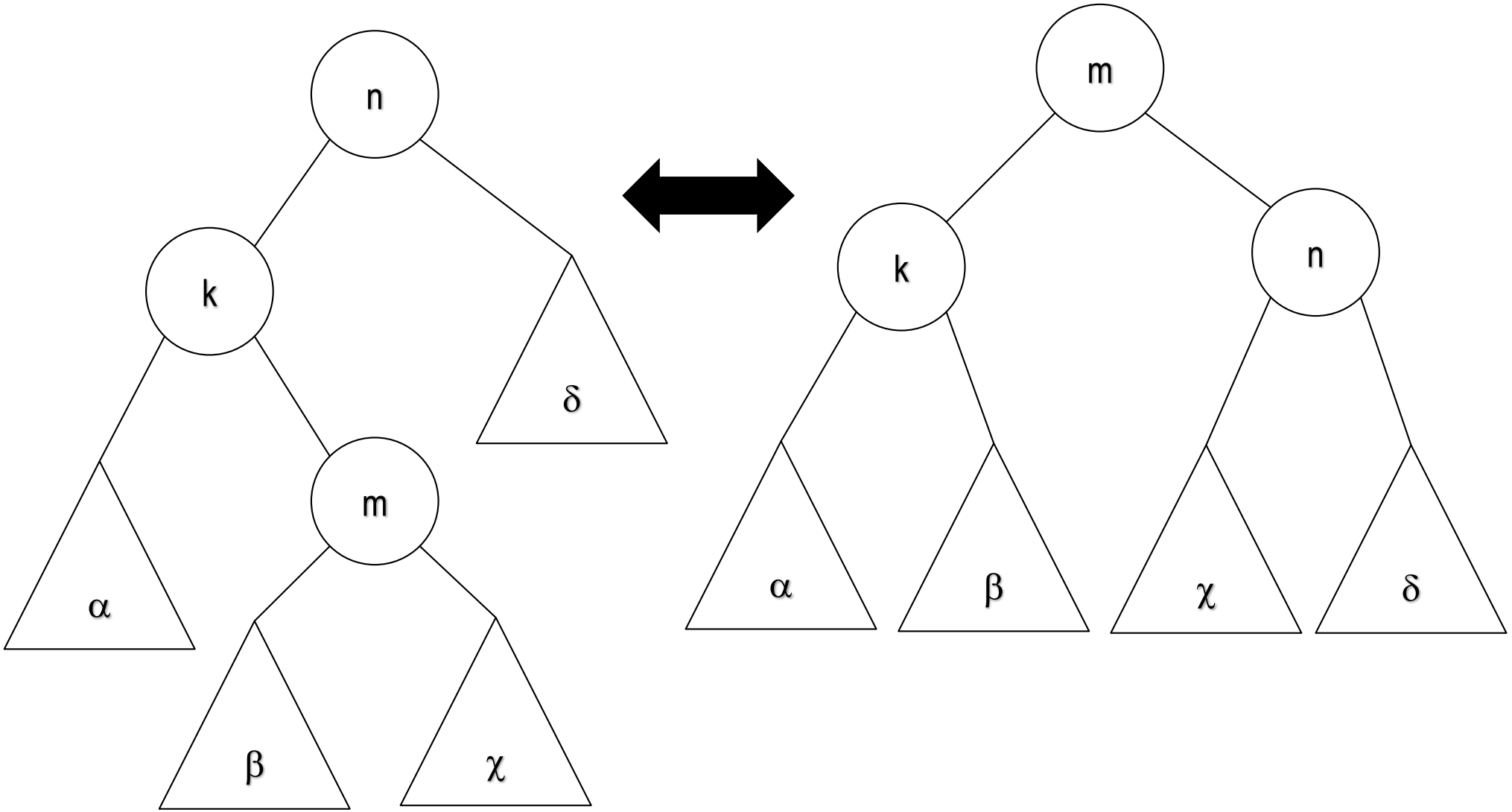
- Balanced binary tree
  - Always keep the height of the tree at its minimum
- Heuristics balancing
  - Maintain properties in the tree that keep the tree mostly balanced
    - Red-black trees
    - Weight balanced trees
    - Height balanced trees
    - AVL trees
  - Restructure the tree to optimize frequent searches – self-adjusting trees
    - When you search for an element in the tree and find it, restructure the tree to make this node quicker to find next time – move the node towards the root of the tree
  - Both rely on rotation operations

Sample Binary Search Tree Rotation



Tree level of k, n, α, and δ change

Sample Binary Search Tree Rotation



Tree level of m, n, β, and χ change