Final Exam Review

CSE 017 | Prof Ziltz | Fall 22

Regular Expressions

Used in `String` Methods to replace and split

- Utilize regular expressions (regex)
  - general pattern in the string
- Used to describe a general pattern in a text
- Helpful in validating user/program input:
  - Phone number (ddd) ddd-dddd
  - Social Security Number ddd-dd-dddd
  - Lehigh email
- Very powerful tool for text analysis

```
+replaceFirst(String regex, String):String
+replaceAll(String regex, String):String
+split(String regex):String[]
+matches(String regex):boolean
```

Regular Expressions

`"Java.*"` `*` stands for any zero or more characters

`"\\d{3}-\\d{2}-\\d{4}"`

`\d` single digit

`{2}` number of digits

`"[$+#%]"` `[]` any one of the characters

Regular Expressions

Regex	Description	Regex	Description
`x`	Specific character x	`\s`	Whitespace character
`.`	Any single character	`\S`	Non whitespace character
`(ab|cd)`	ab or cd	`p*`	Zero or more occurrences of p
`[abc]`	a or b, or c	`p+`	One or more occurrences of p
`[^abc]`	Any character except a, b, or c	`p?`	Zero or one occurrence of p
`[a-z]`	a through z	`p{n}`	Exactly n occurrences of p
`[^a-z]`	Any character except a through z	`p{n,}`	At least n occurrences of p
`\d`	Single digit	`p{n,m}`	Between n and m occurrences of p (inclusive)
`\D`	Non digit		

Regular Expressions

```
"2+3-5".replaceFirst("[+-*/]","%");
// returns "2%3-5

"2+3-5".replaceAll("[+-*/], "%");
// returns "2%3%5
```

```
String[]items ="02/25/2021".split("/");
// returns items = {"02","25","2021"}

String[]tokens =
  "Java,C?C#,C++".split("[.,:;?]");
// returns tokens={"Java", "C", "C#", "C++"}
```

Regular Expressions

```
"2+3-5".matches("\\d[+-]\\d[+-]\\d");
// returns true

"2+3-5".equals("\\d[+-]\\d[+-]\\");
// returns false

"440-02-4534".matches("\\d{3}-\\d{2}-\\d{4}");
// returns true
```

Interfaces

- Include only:
  - `final static` data fields
  - `abstract`, `default` or `static` methods
- Abstract methods in an interface **may** have a `default` definition
  - When an interface is implemented, the default definition may be used or overridden
- **Example:**

```
default String howToEat() {
   return "Eat it the way you want";
}
```

  - In the classes `Chicken, Fruit, Apple, Orange,` the default definition can be used as is or can be overridden
- **Common Interfaces we'll use:**
  - `Comparable`
  - `Cloneable`

Comparable

- Defined in `java.lang`
- `interface` used to compare objects of any type/class
- Includes only one abstract method: `compareTo()`
  - Override in classes that `implement Comparable` to mimic the following:
    - returns 0 if the two arguments are equal
    - returns > 0 if the first argument comes after the second argument
    - returns < 0 if the first argument comes before the second argument

```
public interface Comparable<E> {
   int compareTo(E obj);
}
```

Testing Multiple Generic Types

```
public class Pair<E1, E2> {
   private E1 first;
   private E2 second;
   public Pair(E1 first, E2 second) {
       this.first = first;
       this.second = second;
   }
   public void setFirst(E1 first) {
       this.first = first;
   }
   public void setSecond(E2 second) {
       this.second = second;
   }
   public E1 getFirst() {
       return first;
   }

   public E2 getSecond() {
       return second;
   }
   public String toString() {
       return "(" + first.toString() + ", " +
              second.toString() + ")";
   }
   public boolean equals(Object obj) {
       Pair<E1, E2> p = (Pair<E1, E2>) obj;
       boolean eq1 = p.getFirst().equals(first);
       boolean eq2 = p.getSecond().equals(second);
       return eq1 && eq2;
   }
}
```

```
import java.util.ArrayList;
public class TestPair {
   public static void main(String[] args) {
       ArrayList<Pair<Integer, String>> list = new ArrayList<>();
       Pair<Integer, String> p;
       p = new Pair<Integer, String>(12345, "Lisa Bello");
       list.add(p);
       p = new Pair<Integer, String>(54321, "Karl Johnson");
       list.add(p);
       p = new Pair<Integer, String>(12543, "Jack Green");
       list.add(p);
       p = new Pair<Integer, String>(53241, "Emma Carlson");
       list.add(p);
       System.out.println(list.toString());

   }
}
```

Generic `printArray()`

```
public class GenericPrint{
   public static void main(String[] args) {
       Integer[] numbers = {11, 22, 33, 44, 55};
       String[] names = {"Kallie", "Brandon", "Amelia", "Doug"};
       printArray(numbers);
       printArray(names);
   }
   public static <E> void printArray(E[] list) {
       System.out.print("[ ");
       for (int i=0; i<list.length; i++)
           System.out.print(list[i] + " ");
       System.out.println("]");
   }

}
```

Generic `sortArray()`

- Sorting arrays of different types `java.util.Arrays.sort()`
- `sort()` needs to compare the elements (order them)
- Elements of the array need to be compared – must be comparable
  - Restrict the generic method to objects that can call compareTo()

```
public static <E extends Comparable<E>> void sort(E[] list) {
// Selection Sort
       int currentMinIndex;
         E currentMin;
         for (int i=0; i<list.length-1; i++) {
             currentMinIndex = i;
             currentMin = list[i];
             for(int j=i+1; j<list.length; j++) {
                 if(currentMin.compareTo(list[j]) > 0) {
                     currentMin = list[j];
                     currentMinIndex = j;
                 }
             }
             if (currentMinIndex != i) {
                 list[currentMinIndex] = list[i];
                 list[i] = currentMin;}
         }
  }
```

Recursion

[From Review Questions] Write a recursive method to find occurrences of a character in a string.

```
public static int occ(String s, char c) {
     return occ(s, c, 0); // start from index 0
    }

public static int occ(String s, char c, int index){

     if (index >= s.length())
         return 0;

     else if(s.charAt(index) == c)
          return 1 + occ(s, c, index+1);
     else
          return occ(s, c, index+1);

}
```

Algorithm Analysis

- Big-O notation is a mathematical function for measuring algorithm complexity based on the input size
  - **Time complexity:** Execution time as a function of the input size
  - **Space complexity:** Amount of memory space as a function of the input size

Using a Stack

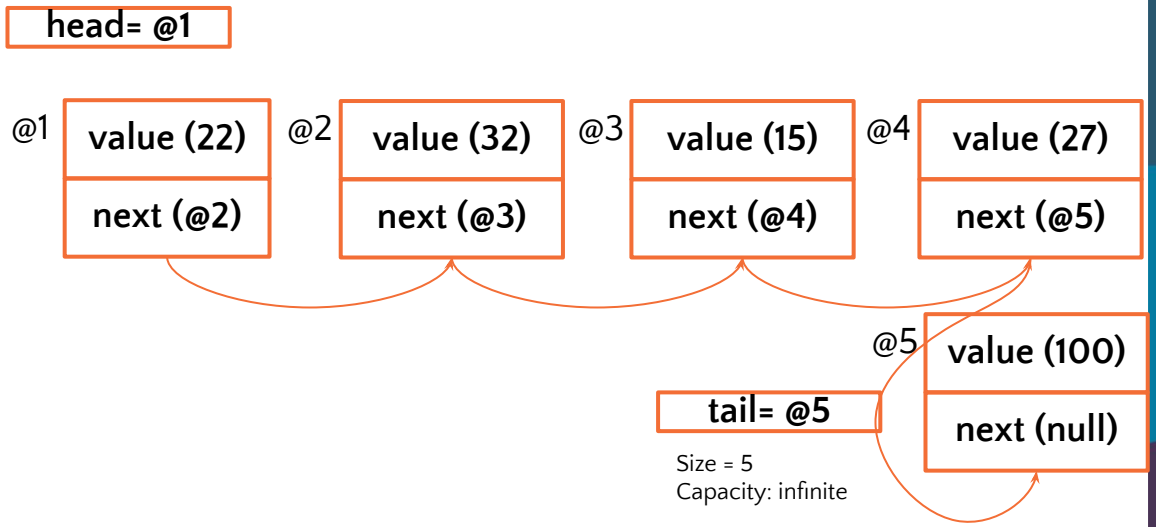
Evaluate the following postfix expression using a stack. Show all the steps.

```
20 6 * 6 5 * 31 + 1 - / 9 - *
```

LinkedLists: Alt to ArrayList

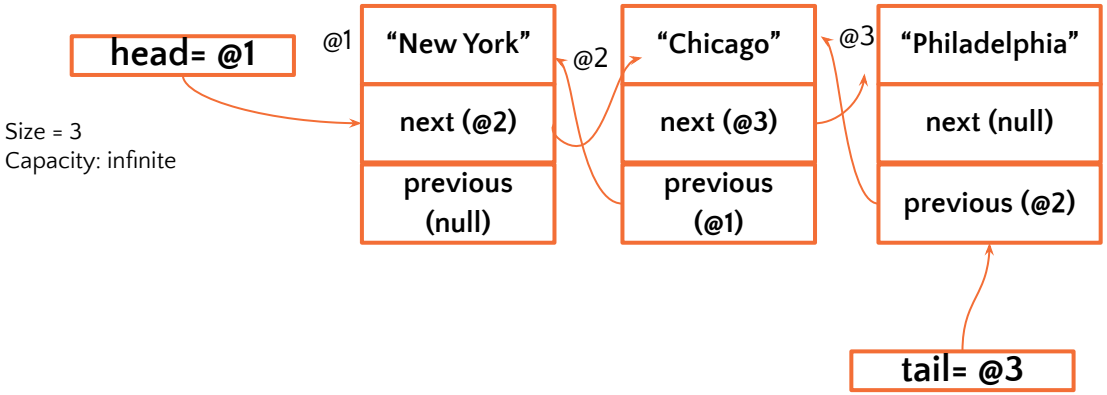
Node:

value	Value of the node (data stored @ that spot in the list)
next	Reference (pointer) to the next node



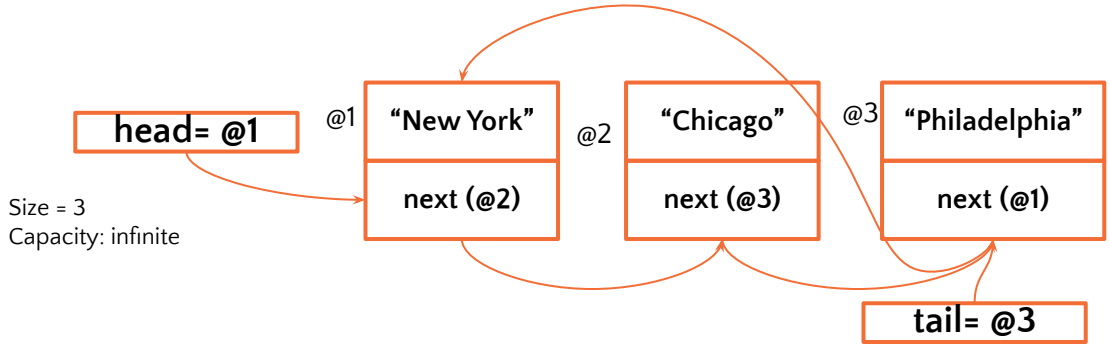
Variations of LinkedLists

- **Doubly Linked List**
  - Every node is linked to the next and the previous elements
  - Improves the performance of removeLast (from O(n) to O(1))



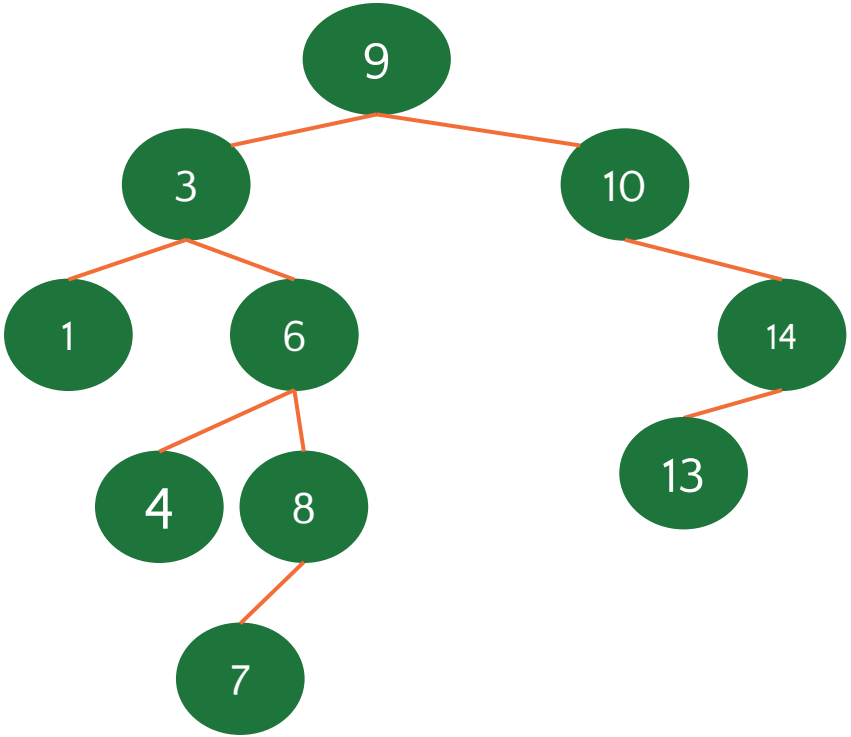
Variations of LinkedLists

- **Circular Linked List**
  - Last element is linked back to the first element



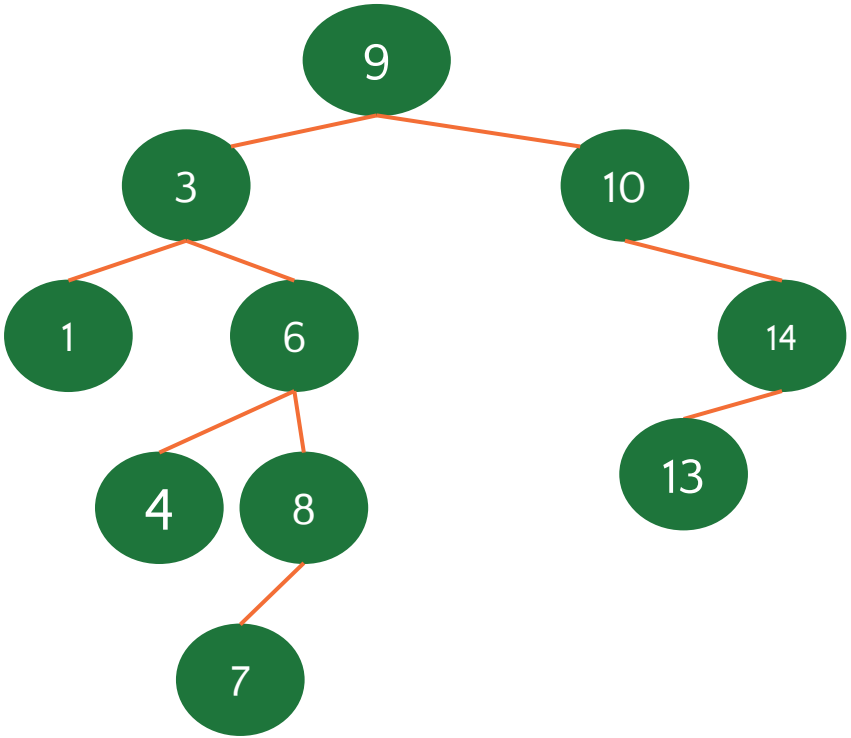
Binary Search Tree

- Special type of binary tree
- BST has a root, a left subtree (L) and a right subtree (R)
- The value of the root is greater than the value of every node in L
- The value of the root is less than the value of every node in R
- L and R are also BSTs
- Used for efficient search in large data sets



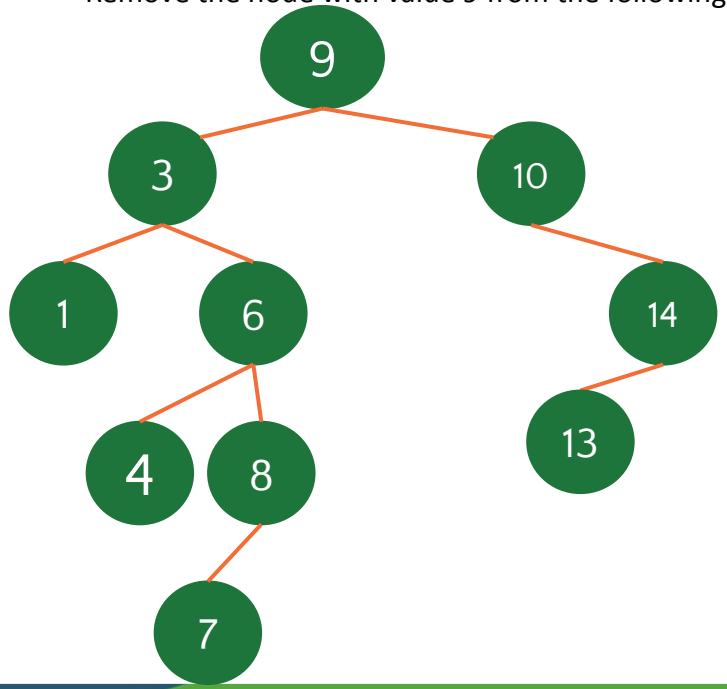
Traversing a BST

- Preorder: Visit-Left-Right
- Postorder: Left-Right-Visit
- Inorder: Left-Visit-Right



Binary Trees (#7)

Remove the node with value 9 from the following binary search tree. Write the in-order traversal of the BST after the deletion.



Binary Trees (#6)

Pre-order: A B C D E F G H I

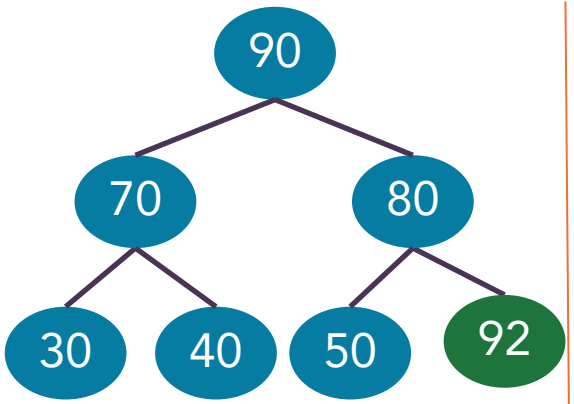
In-order: D C B A F E G I H

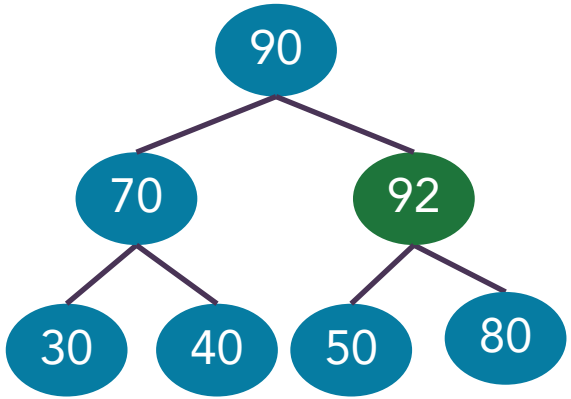
Draw the binary tree and write the post-order traversal of the tree.

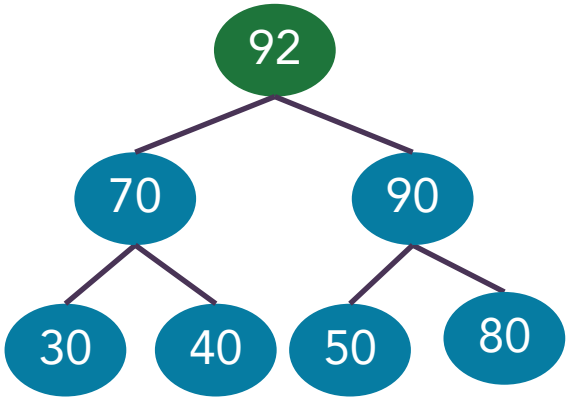
Heap

- **What is it?** Special binary tree
- **Properties:**
  - Complete binary tree - All the levels are filled except the last level
    - All leaves on the last level are placed leftmost
  - Every node is greater than or equal to any of its children (Max Heap)
    - [Min Heap: less than or equal]
- **Used for efficient sorting**

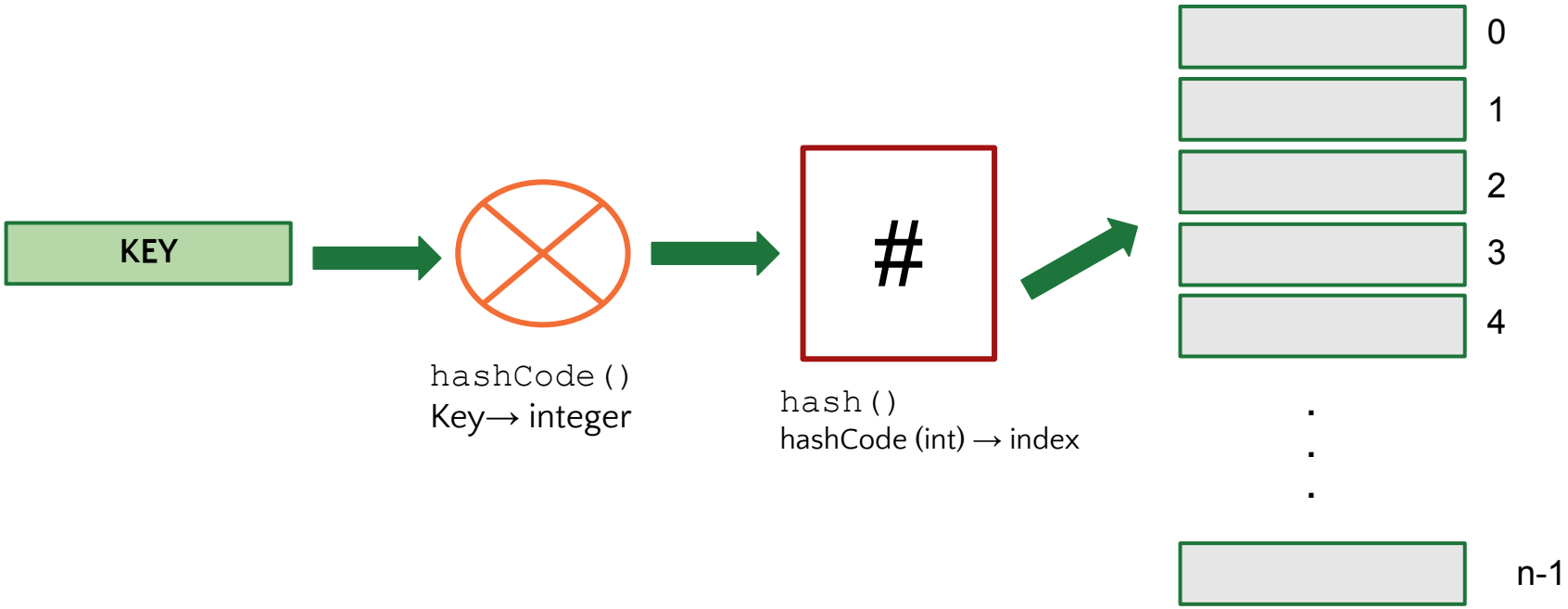
Adding a new node to the heap (92)







Hash Tables



Hash Tables

- If the size of **HT** is N, then

  `0 ≤ hash() ≤ N-1` **(valid index)**

- Searching for a value v is performed using one comparison with

  `HT[hash(hashCode(v))]`

- How data is added/found in **HT** using **`hash()`**?
- How **`hashCode()`** and **`hash()`** are defined?

HashTable Examples

- Values `{34, 29, 53, 44, 120, 39, 45}` to store in a `HT` of size 4
- Each value v is stored at an index i calculated by

```
hash(v)= v%(size of HT)
[i = v % 4 to start]
```

- Load Factor: 0.5
- Collision Handling: Linear Probing

HashTable Examples

- Values `{34, 29, 53, 44, 120, 39, 45}` to store in a `HT` of size 4
- Each value v is stored at an index i calculated by

```
hash(v)= v%(size of HT)
[i = v % 4 to start]
```

- Load Factor: 0.9
- Collision Handling: Separate Chaining

QuickSort

Assume the following list of values to be sorted using **quicksort**. Use a pivot as the first element of the list. Show all the steps to sort the list.

```
{16, 80, 22, 55, 64, 95, 25}
```