CSE 142

Searching & Sorting

Introduction

- **Review:**
  - **Implementing collection classes – StringList**
- **Today:**
  - **Linear & binary search**
  - **Maintaining a sorted list**

Review – class StringList

- **Operations**

```
class StringList {                  // a list of strings
    StringList(int capacity);       // create new StringList with given capacity
    boolean isEmpty( );             // = "this StringList is empty"
    boolean isFull( );              // = "this StringList is full"
    int size( );                    // = # of Strings in this StringList
    boolean add(String str);        // add str to this StringList, result true if success
    int contains(String str);       // = location of str in list, or –1 if not present
    String get(int pos);            // return String at given position
    String remove(int pos);         // return String at given position and remove
                                    //   it from this StringList
```


StringList Representation

- **Underlying Representation is an array of Strings plus a "numStrings" field to keep track of how much of the array is in use**

```
class StringList {                  // a list of strings
    // instance variables
    private String[ ] strings;      // Strings in this StringList are stored in
    private int       numStrings;   //   strings[0] through strings[numStrings-1]
    ...
}
```


Linear Search

- **Locate a string in the list**

```
/** Return location of str in the list, or –1 if not present */
public int contains(String str) {




}
```


Can we do better?

- **How much work does linear search do?**
- **Can we do it faster?**
  - **No, if we don't know anything about the order of elements in the list**
  - **Yes, if the list is sorted**

Binary Search – Informal

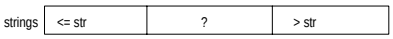
- Idea
  - Look in the middle of the list
  - If we haven't found what we're looking for, we can ignore half of the list and look at the other half
- Precondition: The list must be sorted for this to work
  - We'll assume strings[0] <= strings[1] <= ... <= strings[numStrings-1]
  - (To save a bit of writing, we'll write strings[...] instead of this.strings[...] – works just fine)

Binary Search – Goal

- Goal (more formally)
  - Want to find the midpoint of the list such that everything to the left
    is <= the string we're searching for and everything to the right is >.
- Picture:

Binary Search – Strategy

- On a typical iteration, we have



- Idea:
  - Let mid = (L+R)/2
  - If strings[mid] <= str, move L
  - If strings[mid] > str, move R

String Comparisons

- We need to compare Strings to determine ordering, not just equality
- Can't use <, <=, etc. on objects
- Solution: method compareTo in class String

  s.compareTo(t)

  returns

  negative integer if s < t

  zero if s == t

  positive integer if s > t

Binary Search – Code

```
/** Return location of str in the list, or –1 if not present */
public int contains(String str) {

    while ( _____ ) {



    }

}
```


Binary Search – Test

- Invent some data, try the algorithm

Binary Search – Test

Binary Search – Performance

- Is the extra complexity worth it?
- How much work is done to search a list of a given size?
- or, How big a list can be searched with *n* comparisons?

Binary & Linear Search Compared

- Linear search: work ~ size
- Binary search: work ~ $\log_2$ size
- Graph:

Sorting

- Great, but this only works if the list is sorted
- When do we need to sort the list?
  - Answer: only *required* to be sorted if we want to do binary search
- Choices
  - Keep list sorted at all times
  - Sort list before searching

Maintaining a Sorted List

- Nothing in the client interface changes
- Implementation now relies on list being sorted, so it's crucial that we record this information in a comment

```
// instance variables
private String[ ] strings;      // Strings in this StringList are stored in
private int       numStrings;   //   strings[0] through strings[numStrings-1],
                                //   and the strings are stored in ascending
                                //   order:  strings[0] <= strings[1] <= ...
                                //   <= strings[numStrings-1]
```


Unordered Collections

- Observation: this is no longer a list where the elements are organized by the order in which they are added
- This is a collection of Strings, possibly with duplicates, organized to make searches fast
  - Order is at the convenience of the implementation, not necessarily part of the specification
- Terminology: This is a *multiset* or *bag* of strings

```
/** Unordered collection of Strings, possibly with duplicate elements */
public class StringBag {
    ...
}
```

Method add

- **Only method from original StringList that needs to be changed (true?)**

```
/** Add str to this StringBag.  Return true if successful, otherwise return false */
public boolean add(String str) {
    if (this.numStrings == this.strings.length) {
        return false;
    }
    // find correct location to place str
    …
    // shift larger elements one position to the right
    …
    // place str in correct location
    …
    numStrings++;
}
```


Modified method add

- **Picture:**

Search & Shift

- **Observation: We can find the correct insertion point and shift larger elements to the right in one right-to-left search**
- **Picture:**

Search/shift code

```
// Shift all elements larger than str one position to the right.  When done,
// strings[pos] is the correct location for str


while ( _____ ) {



}
strings[pos] = str;
numStrings++;
```


Sorting an Unsorted Array

- **What if we didn't keep the list sorted as elements are added?**
- **Answer: can apply our shift/search to the existing contents of the StringList**

```
for (k = 1; k < numStrings; k++) {
    // place strings[k] in correct location in strings[0..k-1]
}
```

- **Picture:**

- **This is *insertion sort*, a common, simple sorting algorithm**