CSC 143 Java

Searching and Recursion
N&H Chapters 13, 17

Overview

- Topics
  - Maintaining an ordered list
  - Sequential and binary search
  - Recursion
  - Sorting: insertion sort and QuickSort
- Reading
  - Textbook: ch. 13 & sec. 17.1-17.3

Problem: A Word Dictionary

- Suppose we want to maintain a real dictionary. Data is a list of <word, definition> pairs -- a "Map" structure

  <"aardvark", "an animal that starts with an A and ends with a K">
  <"apple", "a leading product of Washington state">
  <"banana", "a fruit imported from somewhere else">
  etc.
- We want to be able to do the following operations efficiently
  - Look up a definition given a word (key)
  - Retrieve sequences of definitions in alphabetical order

Representation

- Need to pick a data structure
- Analyze possibilities based on cost of operations

	search	access next in order
• unordered list		
• hash map		
• ?		

Ordered List

- One solution: keep list in alphabetical order
- To simplify the explanations for the present: we'll treat the list as an array of strings, and assume it has sufficient capacity to add additional word/def's when needed

0 aardvark
1 apple
2 banana
3 cherry
4 kumquat
5 orange
6 pear
7 rutabaga

```
// instance variable of the Ordered List class
String[ ] words;    // list is stored in words[0..size-1]
int size;           // # of words
```


Sequential (Linear) Search

- Assuming the list is initialized in alphabetical order, we can use a *linear search* to locate a word

```
// return location of word in words, or –1 if found
int find(String word) {
    int k = 0;
    while (k < size && !word.equals(words[k]) {
        k++
    }
    if (k < size) { return k; } else { return –1; }     // lousy indenting to fit on slide
}                                                        // don't do this at home
```

- Time for list of size n:

Can we do better?

- Yes! *If* array is sorted
- Binary search:
  - Examine middle element
  - Search either left or right half depending on whether desired word precedes or follows middle word alphabetically
- The list being sorted is a *precondition* of binary search.
  - The algorithm is not guaranteed to give the correct answer if the precondition is violated.

Binary Search

```
// Return location of word in words, or –1 if not found
int find(String word) {
    return bSearch(0, size-1);
}
// Return location of word in words[lo..hi] or –1 if not found
int bSearch(String word, int lo, int hi) {
    // return –1 if interval lo..hi is empty
    if (lo > hi)  { return  –1; }
    // search words[lo..hi]
    int mid = (lo + hi) / 2;
    int comp = word.compareTo(words[mid]);
    if (comp == 0)  { return mid; }
    else if (comp < 0)  { return ______________________ ; }
    else /* comp > 0 */ { return ______________________ ; }
}
```


"The Word Must Be Where?" Three Cases

```
int comp = word.compareTo(words[mid]);
if (comp == 0) {
        //the word must be where? _____________
        return _______________ ;
}
else if (comp < 0)  {
        //the word must be where? ___________
        return ______________________ ;
}
else  {     //comp > 0
        //the word must be where? ___________
        return ______________________ ;
}
```


"Where?" Answered

```
int comp = word.compareTo(words[mid]);
if (comp == 0) {
        //the word must be where? at position "mid"
        return _______________ ;
}
else if (comp < 0)  {
        //the word must be where? in the lower half of the array
        return ______________________ ;
}
else  {     //comp > 0
        //the word must be where? in the upper half of the array
        return ______________________ ;
}
```


Return Values: Three Cases

```
int comp = word.compareTo(words[mid]);
if (comp == 0) {
        //the word must be where? at position "mid"
        return mid;
}
else if (comp < 0)  {
        //the word must be where? in the lower half of the array
        return /*the result of searching the lower half of the array*/
        ____________________ ;
}
else  {     //comp > 0
        //the word must be where? in the upper half of the array
        return /*the result of searching the upper half of the array*/
        ____________________ ;
}
```


What is "The Lower Half"?

```
...     else if (comp < 0)  {
        //the word must be where? in the lower half of the array
        return /*the result of searching the lower half of the array*/
                ______________________ ;
        }
...
```

Remember the method header was:

// Return location of word in words[lo..hi] or –1 if not found
int bSearch(String word, int lo, int hi) {

So the lower half starts at _____ and ends at ______

return /*the result of searching the lower half of the array*/ becomes
return /*the result of searching the array from ___ to ___*/

Comments Complete, Code Incomplete

```
int comp = word.compareTo(words[mid]);
if (comp == 0) {
        //the word must be where? at position "mid"
        return mid;
}
else if (comp < 0)  {
        //the word must be where? in the lower half of the array
         return /*the result of searching from lo to mid-1*/
_______________________ ;
 }
else {      //comp > 0
        //the word must be where? in the upper half of the array
        return /*the result of searching from mid+1 to hi*/
_______________________ ;
}
```


Last Piece of the Puzzle

```
...
return /*the result of searching from lo to mid-1*/
_______________________ ;
}
```

How can we get the "result of searching from lo to mid-1"?

We have a method called bSearch that can search an array within a range of indexes.

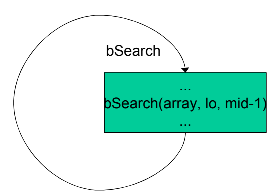
// Return location of word in words[x.y] or –1 if not found
int bSearch(String word, int x, int y)

Let x be lo, let y be mid-1
bSearch(String word, int **lo**, int **mid-1**)

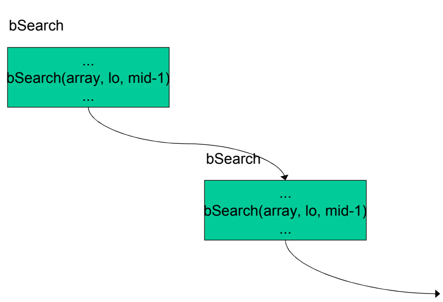
Recursion

- A method (function) that calls itself is *recursive*
- Nothing really new here
- Method call review:
  - Evaluate argument expressions
  - Allocate space for parameters and local variables of function being called
  - Initialize parameters with argument values
  - Then execute the function body
- What if the function being called is the same one that is doing the calling?
  - Answer: no difference at all!

Wrong Way to Think About It



Right Way to Think About It



Trace

- Trace execution of find("orange")

0 aardvark
1 apple
2 banana
3 cherry
4 kumquat
5 orange
6 pear
7 rutabaga

Trace

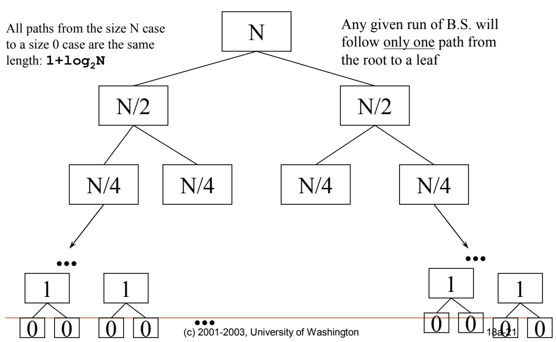
- Trace execution of *find("kiwi")*

  0 aardvark
  1 apple
  2 banana
  3 cherry
  4 kumquat
  5 orange
  6 pear
  7 rutabaga

Performance of Binary Search

- Analysis
  - Time (number of steps) per each recursive call:
  - Number of recursive calls:
  - Total time:
- A picture helps

Binary Search Sizes



Linear Search vs. Binary Search

- Compare to linear search
  - Time to search 10, 100, 1000, 1,000,000 words

    linear

    binary
  - What is incremental cost if size of list is doubled?
- Why is Binary search faster?
  - The data structure is the same
  - The precondition on the data structure is different: stronger
  - Recursion itself is *not* an explanation

    One could code linear search using recursion

More About Recursion

A recursive function needs three things to work properly

1. One or more *base cases* that are not recursive
   - if (lo > hi) { return –1; }
   - if (comp == 0) { return mid; }
2. One or more *recursive cases* that handle a else if (comp < 0) { return bsearch(word,lo,mid 1); }
   - else /* comp > 0 */ { return bsearch(word,mid+1,hi); }
3. The recursive cases must lead to "smaller" instances of the problem
   - "Smaller" means: closer to a base case
   - Without "smaller", what might happen?

Recursion vs. Iteration

- Recursion can completely replace iteration
- Some rewriting of the algorithm is necessary
  - usually minor
- Some languages have recursion only
- Recursion is often more elegant but less efficient
- Recursion is a natural for certain algorithms and data structures (where branching is required)
  - Useful in "divide and conquer" situations

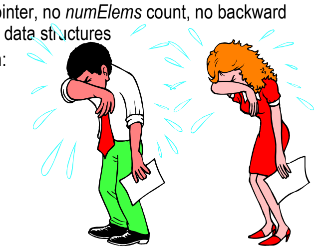
- Iteration can completely replace recursion
- Some rewriting of the algorithm is necessary
  - often major
- A few (mostly older languages) have iteration only
- Iteration is not always elegant but is usually efficient
- Iteration is natural for linear (non-branching) algorithms and data structures

Recursion and Elegance

- Problem: reverse a linked list
- Constraints: no *last* pointer, no *numElems* count, no backward pointers, no additional data structures
- Non-recursive solution:
  - try it!

Result of Trying To Reverse a Linked List Iteratively

- Problem: reverse a linked list
- Constraints: no *last* pointer, no *numElems* count, no backward pointers, no additional data structures
- Non-recursive solution:
  - try it and weep



- Better hope this wasn't a question on your Microsoft job interview!

Recursive Solution: Simple, Elegant

- Problem: reverse a linked list
- Constraints: no *last* pointer, no *numElems* count, no backward pointers, no additional data structures

```
newList = reverse(oldList.first);
...
List reverse(Link firstLink) {
   if (firstLink == null) { return new SimpleList(); }
   return reverse(firstLink.next).add(firstLink.data));
   }
```

- Better hope this *is* a question on your Microsoft job interview!
- PS: Did we cheat??