## CSC 143 Java

Searching



## Overview

- Topics
  - Maintaining an ordered list
  - Sequential and binary search
  - Recursion
  - Sorting: insertion sort and QuickSort



## 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 | O(N) | O(N) |
| • Ordered list | O(log N) | O(1) |

## 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        // instance variable of the Ordered List class
1  apple           String[ ] words;    // list is stored in words[0..size-1]
2  banana          int size;           // # of words
3  cherry
4  kumquat
5  orange
6  pear
7  rutabaga
```



## 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: O(n) We can do better!



## 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 (with a loop)

```
// Return location of word in words, or – (where it would be + 1)
int find(String word) {
    int lo = 0, hi = size – 1;
    while (lo <= hi) {
      int mid = (lo + hi) / 2;
      int comp = word.compareTo(words[mid]);
      if (comp == 0)  { return mid; }
      else if (comp < 0)  { hi = mid – 1;}
      else /* comp > 0 */ { lo = mid + 1; }
    }
    return –lo - 1
}
```

## Binary Search (recursion)

```
// Return location of word in words, or – (where it would be + 1)
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 –(where it would be + 1) if interval lo..hi is empty
    if (lo > hi)   { return  –lo - 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 bSearch( lo, mid - 1); }
    else /* comp > 0 */ { return bSearch(mid + 1, hi); }
}
```

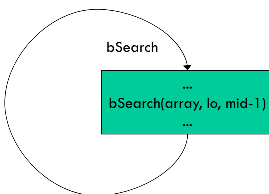


## Recursion

- 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!

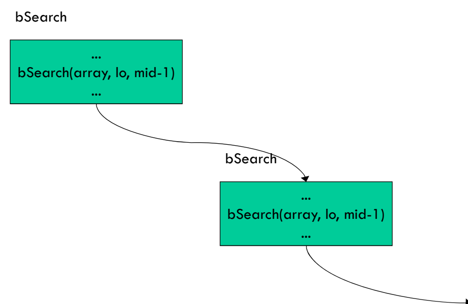


## Wrong Way to Think About It






## 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

| lo | hi | mid |
|---|---|---|
| 0 | 7 | 3 ("cherry") |
| 4 | 7 | 5 ("orange") |

5 is returned



## Trace

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

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

| lo | hi | mid |
|---|---|---|
| 0 | 7 | 3 ("cherry") |
| 4 | 7 | 5 ("orange) |
| 4 | 4 | 4 ("kumpat) |
| 4 | 3 | lo > hi |

-(4 + 1) = -5 is returned (since if "kiwi" was in the array, it would be at postion 4)

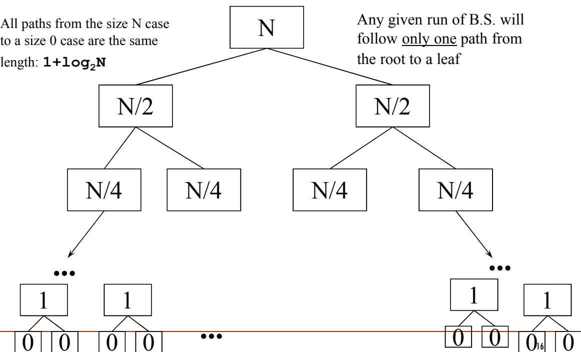


## Performance of Binary Search

- Analysis
  - Time (number of steps) per each recursive call: O(1)
  - Number of recursive calls: O(log N)
  - Total time: O(log N)
- 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 10, 100, 1000, 1,000,000

    Binary ~ 1, 2, 3, 6
  - What is incremental cost if size of list is doubled?
    Linear x 2, binary + log(2)
- 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?