

CSE 143 Java

Searching and Recursion

Reading: Ch. 14 & Secs. 19.1-19.2

11/18/2004

(c) 2001-4, University of Washington

18-1

Overview

- Topics
 - Sequential and binary search
 - Recursion

11/18/2004

(c) 2001-4, University of Washington

18-2

Problem: A Word Dictionary

- Suppose we want to maintain a list of words
 - "aardvark"
 - "apple"
 - "tomato"
 - "orange"
 - "banana"
 - etc.
- Use the same basic representation as in `SimpleArrayList`

```
String[] words; // the list of words is stored in words[0..size-1]
int size; // number of words currently in the list
```
- We would like to be able to determine efficiently if a particular word is in the list

11/18/2004

(c) 2001-4, University of Washington

18-3

Sequential (Linear) Search

- If we don't know anything about the order of the words in the list, we basically have to use a *linear search* to look for 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
```

- Search time for list of size n:
 - Can we do better?

11/18/2004

(c) 2001-4, University of Washington

18-4

Can we do better?

- Yes if the list is in alphabetical order

```
0 aardvark // instance variable of the Ordered List class
1 apple // list is stored in words[0..size-1]
2 banana // and words are in ascending
3 cherry // order
4 kumquat
5 orange
6 pear
7 rutabaga
```

11/18/2004

(c) 2001-4, University of Washington

18-5

Binary Search

- Key idea: to search a section of the array,
 - Examine middle element
 - Search either left or right half depending on whether desired word precedes or follows middle word alphabetically
- A *precondition* for binary search is that the list is sorted
 - The algorithm is not guaranteed (or required) to give the correct answer if the precondition is violated

11/18/2004

(c) 2001-4, University of Washington

18-6

Binary Search Sketch (not quite legal Java)

```
/** Return the location of word in words[lo..hi], or -1 if not found */
int bSearch(String word, int lo, int hi) {
    if (lo > hi) { return -1; }           // empty interval
    int mid = (lo + hi) / 2;
    if (word == words[mid]) { return mid; } // found it!
    else if (word <= words[mid]) {
        // look for word in the left half
        return _____;
    } else { // word >= words[mid]
        // look for word in the right half
        return _____;
    }
}
```

11/18/2004

(c) 2001-4, University of Washington

18-7

Recursion

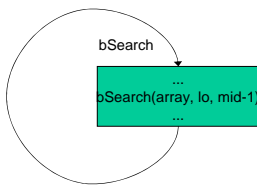
- A method (function) that calls itself is *recursive*
- Nothing really new here
- Method call review:
 - Evaluate argument expressions
 - Allocate space for parameters and local variables of function being called
 - Initialize parameters with argument values
 - Then execute the function body
- What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

11/18/2004

(c) 2001-4, University of Washington

18-8

Wrong Way to Think About It

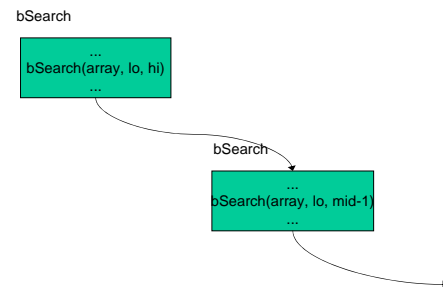


11/18/2004

(c) 2001-4, University of Washington

18-9

Right Way to Think About It



11/18/2004

(c) 2001-4, University of Washington

18-10

Recursive Definitions

- We see these all the time in mathematics
- Simple example: factorial function

$$n! = \begin{cases} 1, & \text{if } n \leq 1 \\ n \times (n-1)!, & \text{otherwise} \end{cases}$$

11/18/2004

(c) 2001-4, University of Washington

18-11

Recursive Implementation in Java

- We can use the definition directly to create a java method to compute factorial

```
/** Return n! */
int fact(int n) {
    if (n <= 1) {
        return _____;
    } else {
        return _____;
    }
}
```

11/18/2004

(c) 2001-4, University of Washington

18-12

Trace

- Execution of: `result = fact(4);`

11/18/2004

(c) 2001-4, University of Washington

18-13

Recursive Cases, Base Cases, and Termination

- A recursive definition needs to have two parts
 - One or more *base cases* that are not recursive
`if (n <= 1) { return 1; }`
 - One or more *recursive cases* that handle a “smaller” instance of the problem
`else { return n * fact(n-1); }`
- The recursive cases must “make progress” towards a base case
 - If not, or if no base case(s) – infinite recursion

11/18/2004

(c) 2001-4, University of Washington

18-14

Back to Binary Search – Real Java This Time

```
/* Return word loc. in the list 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 bSearch(word, lo, mid-1);
    } else /* comp > 0 */ {
        return bSearch(word, mid+1, hi);
    }
}
```

- Which are the

- Base case(s)?

- Recursive case(s)?

- How do the recursive case(s) make progress towards the base case(s)?

11/18/2004

(c) 2001-4, University of Washington

18-15

Trace

- Trace execution of `find("orange")`

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

11/18/2004

(c) 2001-4, University of Washington

18-16

Trace

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

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

11/18/2004

(c) 2001-4, University of Washington

18-17

Analysis of Binary Search

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

11/18/2004

(c) 2001-4, University of Washington

18-18

How Many Calls Needed for a List of Size n ?

of recursive calls needed (t) List size (n)

11/18/2004

(c) 2001-4, University of Washington

18-19

Graph: Linear vs Binary Search

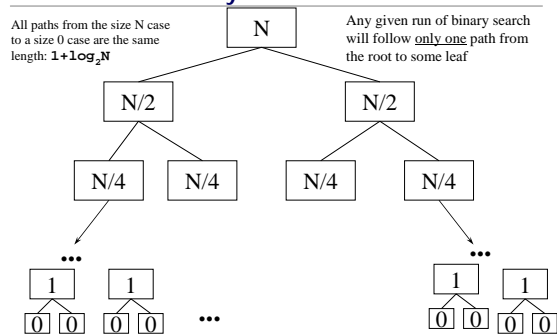
11/18/2004

(c) 2001-4, University of Washington

18-20

Another Way to Picture the Cost

All paths from the size N case to a size 0 case are the same length: $1 + \log_2 N$



11/18/2004

(c) 2001-4, University of Washington

18-21

Linear Search vs. Binary Search

- What is incremental cost if size of list is doubled?
 - Linear search:
 - Binary search:
- 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, or binary search with a loop

11/18/2004

(c) 2001-4, University of Washington

18-22

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 has some extra overhead (often not a major issue, but can be)
- Recursion is a natural for certain algorithms and data structures
 - 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

11/18/2004

(c) 2001-4, University of Washington

18-23

Recursion Summary

- Recursive definition: a definition that is (partially) given in terms of itself
- Recursive method (function): a method that is (partially) implemented by calling itself
- Need base case(s) and recursive case(s)
 - Recursive cases must make progress towards reaching a base case – must solve "smaller" subproblems
- Often a very elegant way to formulate a problem
 - Let the method call mechanism handle the bookkeeping behind the scenes for you
- A powerful technique – add it to your toolbag

11/18/2004

(c) 2001-4, University of Washington

18-24