CSE 143 Java

Searching, Recursion, and Sorting

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

New 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 Kt”>
  <“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 diagrams, 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:

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

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 ________________________ ; }
}
```

Binary Search -- Detail of Last Lines

```
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 ______________________ ;
}
```

11/18/2002 (c) 2001, University of Washington 16-9

Binary Search -- Detail of Last Lines

```
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 ______________________ ;
}
```

11/18/2002 (c) 2001, University of Washington 16-10

Binary Search -- Detail of Last Lines

```
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*/
        ______________________ ;
}
```

11/18/2002 (c) 2001, University of Washington 16-11

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 ___*/

11/18/2002 (c) 2001, University of Washington 16-12

Last Lines --Comments Complete

```
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 function that calls itself is *recursive*
- Nothing really new here
- Function 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
- No difference if the function being called is the same one that is doing the calling

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

Recursive Definitions

- A recursive function needs two things to work properly
  - One or more *base cases* that are not recursive

    if (lo > hi) { return –1; }
    if (comp == 0) { return mid; }
  - One or more *recursive cases* that handle a “smaller” instance of the problem

    else if (comp < 0) { return bsearch(word,lo,mid-1); }
    else /* comp > 0 */ { return bsearch(word,mid+1,hi); }

    "Smaller" means: closer to a base case
    Without "smaller", what might happen?

Performance of Binary Search

- Analysis
  - Time of each recursive call:
  - Number of recursive calls:
  - Total time:
- 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?

Sorting

- Binary search is a huge speedup over sequential search
  - But requires the list be sorted
- Slight Problem: How do we get a sorted list?
  - Maintain the list in sorted order as each word is added
  - Sort the entire list when needed

Insert for a Sorted List

- Exercise: Assume that words[0..size-1] is sorted. Place new word in correct location so modified list remains sorted
  - Assume that there is spare capacity for the new word (what kind of condition is this?)
- Before coding:
  - Draw pictures of an example situation, before and after
  - Write down the postconditions for the operation

```
// given existing list words[0..size-1], insert word in correct place and increase size
void insertWord(String word) {



    size++;
}
```


Insertion Sort

- Once we have insertWord working...
- We can sort a list in place by repeating the insertion operation

```
void insertionSort( ) {
    int finalSize = size;
    size = 1;
    for (int k = 1; k < finalSize; k++) {
      insertWord(words[k]);
    }
}
```


Insertion Sort Trace

- Initial array contents

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

Insertion Sort Performance

- Cost of each insertWord operation:
- Number of times insertWord is executed:
- Total cost:
- Can we do better?

Analysis

- Why was binary search so much more effective than sequential search?
  - Answer: binary search divided the search space in half each time; sequential search only reduced the search space by 1 item
- Why is insertion sort $O(n^2)$?
  - Each insert operation only gets 1 more item in place at cost O(n)
  - O(n) insert operations
- Can we do something similar for sorting?

Divide and Conquer Sorting

- Idea: like binary search, divide the sorting problem into two subproblems; recursively sort each subproblem; combine results
  - Want division and combination at the end to be fast
  - Want to be able to sort two halves independently
- This is a particular example of an algorithm technique known as “divide and conquer”

Quicksort

- Invented by C. A. R. Hoare (1962)
- Idea
  - Pick an element of the list: the *pivot*
  - Place all elements of the list smaller than the pivot in the half of the list to its left; place larger elements to the right
  - Recursively sort each of the halves
- Before looking at any code, see if you can draw pictures based just on the first two steps of the description

Code for Quicksort

```
// Sort words[0..size-1]
void quickSort( ) {
    qsort(0, size-1);
}

// Sort words[lo..hi]
void qsort(int lo, int hi) {
    // quit if empty partition
    if (lo > hi) { return; }
    int pivotLocation = partition(lo, hi);       // partition array and return pivot loc
    qsort(lo, pivotLocation-1);
    qsort(pivotLocation+1, hi);
}
```

Recursion Analysis

- Base case? Yes.

```
// quit if empty partition
if (lo > hi) { return; }
```

- Recursive cases? Yes

```
qsort(lo, pivotLocation-1);
qsort(pivotLocation+1, hi);
```

  - Observation: recursive cases work on a smaller subproblem, so algorithm will terminate

A Small Matter of Programming

- Partition function
  - Pick pivot
  - Rearrange array so all smaller element are to the left, all larger to the right, with pivot in the middle
- How do we pick the pivot?
  - For now, keep it simple – use the first item in the interval

Partition design

- We need to partition words[lo..hi]
- Pick words[lo] as the pivot
- Picture:

Partition Algorithm

```
// Partition words[lo..hi]; return location of pivot in range lo..hi
int partition(int lo, int hi)
```

Partition Test

- Check: partition(0,7)
  - 0 orange
  - 1 pear
  - 2 apple
  - 3 rutabaga
  - 4 aardvark
  - 5 cherry
  - 6 banana
  - 7 kumquat

Quicksort Performance

- Cost of each recursive call
  - Cost of partition = O(n) where n is the size of the part of the list being sorted (a smaller part of the original array)
  - Some O(1) work
- Number of recursive calls
  - Assume that each partition operation divides list in half at cost O(n/2)
  - How many recursive calls?

Quicksort Performance (Ideal Case)

- Each partition divides the list parts in half
  - Sublist sizes on recursive calls: n, n/2, n/4, n/8….
  - Total depth of recursion: _________________
  - Total work at each level: O(n)
  - Total cost of quicksort: _______________ !
- For a list of 10,000 items
  - Insertion sort: $O(n^2)$: 100,000,000
  - Quicksort: O(n log n): 10,000 $\log_2$ 10,000 = 132,877

Quicksort Performance (Worst Case)

- Each partition manages to pick the largest or smallest item in the list as a pivot
  - Sublist sizes on recursive calls:
  - Total depth of recursion: _________________
  - Total work at each level: O(n)
  - Total cost of quicksort: _______________ !

Worst Case vs Average Case

- In practice, Quicksort works well, provided the pivot is picked with some care. Some strategies:
  - Compare a small number of list items (3-5) and pick the median for the pivot
  - Pick a pivot element randomly in the range lo..hi

Summary

- Recursion
  - Functions that call themselves to solve subproblems
  - Need base case(s) and recursive case(s)
  - Often a very clean way to formulate a problem (let the function call mechanism handle bookkeeping behind the scenes)
- Divide and Conquer
  - Algorithm design strategy that exploits recursion
  - Divide original problem into subproblems
  - Solve each subproblem recursively
  - Can sometimes yield dramatic performance improvements