CSC 427: Data Structures and Algorithm Analysis

Fall 2006

Inheritance and efficiency

- ArrayList → SortedArrayList
- tradeoffs with adding/searching
- timing code
- divide-and-conquer algorithms

Dictionary revisited

recall the Dictionary class from last week

- the ArrayList add method simply appends the item at the end → O(1)
- the ArrayList contains method performs sequential search → O(N)

this is OK if we are doing lots of adds and few searches

```
import java.util.List;
import java.util.ArrayList;
import java.util.Scanner;
import java.io.File;

public class Dictionary {
    private List<String> words;

    public Dictionary() {
        this.words = new ArrayList<String>();
    }

    public Dictionary(String filename) {
        this();

        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String nextWord = infile.next();
                this.words.add(nextWord.toLowerCase());
            }
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }

    public void add(String newWord) {
        this.words.add(newWord.toLowerCase());
    }

    public void remove(String oldWord) {
        this.words.remove(oldWord.toLowerCase());
    }

    public boolean contains(String testWord) {
        return this.words.contains(testWord.toLowerCase());
    }
}
```

Timing dictionary searches

we can use the `java.util.Date` class to verify the O(N) efficiency

dict. size	insert time
38,621	400 msec
77,242	751 msec
144,484	1222 msec

dict. size	search time
38,621	120 msec
77,242	291 msec
144,484	591 msec

execution time roughly doubles as dictionary size doubles

```
import java.util.Scanner;
import java.io.File;
import java.util.Date;

public class DictionaryTimer {

    public static void main(String[] args) {
        System.out.println("Enter name of dictionary file:");
        Scanner input = new Scanner(System.in);

        Date start1 = new Date();
        Dictionary dict = new Dictionary(input.next());
        Date end1 = new Date();

       System.out.println(end1.getTime()-start1.getTime());

        Date start2 = new Date();
        for (int i = 0; i < 100; i++) {
            dict.contains("zzyzfys");
        }
        Date end2 = new Date();

        System.out.println(end2.getTime()-start2.getTime());
    }
}
```

3

Sorting the list

if searches were common, then we might want to make use of binary search

- this requires sorting the words first, however

we could change the Dictionary class to do the sorting and searching

- a more general solution would be to extend the ArrayList class to SortedArrayList
- could then be used in any application that called for a sorted list

recall:

```
public class java.util.ArrayList<E> implements List<E> {
    public ArrayList() { … }
    public boolean add(E item) { … }
    public void add(int index, E item) { … }
    public E get(int index) { … }
    public E set(int index, E item) { … }
    public int indexOf(Object item) { … }
    public boolean contains(Object item) { … }
    public boolean remove(Object item) { … }
    public E remove(int index) { … }
    …
}
```

4

SortedArrayList (v.1)

using inheritance, we only need to redefine what is new

- add method sorts after adding; indexOf uses binary search
- no additional fields required

- big-Oh for add? big-Oh for indexOf?

```
public class SortedArrayList<E extends Comparable<? super E>> extends ArrayList<E> {
    public SortedArrayList() {
        super();
    }

    public boolean add(E item) {
        super.add(item);
        Collections.sort(this);
        return true;
    }

    public int indexOf(Object item) {
        return Collections.binarySearch(this, (E)item);
    }
}
```

5

SortedArrayList (v.2)

is this version any better? when?

- big-Oh for add?
- big-Oh for indexOf?

```
public class SortedArrayList<E extends Comparable<? super E>> extends ArrayList<E> {
    public SortedArrayList() {
        super();
    }

    public boolean add(E item) {
        super.add(item);
        return true;
    }

    public int indexOf(Object item) {
        Collections.sort(this);
        return Collections.binarySearch(this, (E)item);
    }
}
```

6

SortedArrayList (v.3)

if insertions and searches are mixed, sorting for each insertion/search is extremely inefficient

- instead, could take the time to insert each item into its correct position
- big-Oh for add? big-Oh for indexOf?

```
public class SortedArrayList<E extends Comparable<? super E>> extends ArrayList<E> {
    public SortedArrayList() {
        super();
    }

    public boolean add(E item) {
        int i;
        for (i = 0; i < this.size(); i++) {
            if (item.compareTo(this.get(i)) < 0) {
                break;
            }
        }
        super.add(i, item);
        return true;
    }

    public int indexOf(Object item) {
        return Collections.binarySearch(this, (E)item);
    }
}
```

7

Timing dictionary searches

note that repeated calls to add serve as insertion sort

dict. size	insert time
38,621	33.7 sec
77,242	129.9 sec
144,484	508.9 sec

dict. size	search time
38,621	0 msec
77,242	0 msec
144,484	10 msec

insertion time roughly quadruples as dictionary size doubles; search time is trivial

```
import java.util.Scanner;
import java.io.File;
import java.util.Date;

public class DictionaryTimer {

    public static void main(String[] args) {
        System.out.println("Enter name of dictionary file:");
        Scanner input = new Scanner(System.in);

        Date start1 = new Date();
        Dictionary dict = new Dictionary(input.next());
        Date end1 = new Date();

        System.out.println(end1.getTime()-start1.getTime());

        Date start2 = new Date();
        for (int i = 0; i < 100; i++) {
            dict.contains("zzyzfys");
        }
        Date end2 = new Date();

        System.out.println(end2.getTime()-start2.getTime());
    }
}
```

8

SortedArrayList (v.4)

if adds tend to be done in groups (as in loading the dictionary)

- it might pay to perform lazy insertions & keep track of whether sorted
- big-Oh for add? big-Oh for indexOf?
- if desired, could still provide addInOrder method (as before)

```
public class SortedArrayList<E extends Comparable<? super E>> extends ArrayList<E> {
    private boolean isSorted;

    public SortedArrayList() {
        super();
        this.isSorted = true;
    }

    public boolean add(E item) {
        this.isSorted = false;
        return super.add(item);
    }

    public int indexOf(Object item) {
        if (!this.isSorted) {
            Collections.sort(this);
            this.isSorted = true;
        }
        return Collections.binarySearch(this, (E)item);
    }
}
```

9

Timing dictionary searches

if we modify the Dictionary class to use a SortedArrayList

dict. size	insert time
38,621	461 msec
77,242	829 msec
144,484	1592 msec

dict. size	1st search
38,621	0 msec
77,242	0 msec
144,484	10 msec

dict. size	search time
38,621	0 msec
77,242	0 msec
144,484	10 msec

```
import java.util.Scanner;
import java.io.File;
import java.util.Date;

public class DictionaryTimer {

    public static void main(String[] args) {
        System.out.println("Enter name of dictionary file:");
        Scanner input = new Scanner(System.in);

        Date start1 = new Date();
        Dictionary dict = new Dictionary(input.next());
        Date end1 = new Date();

        System.out.println(end1.getTime()-start1.getTime());

        Date start2 = new Date();
        dict.contains("zzyfys");
        Date end2 = new Date();

        System.out.println(end2.getTime()-start2.getTime());

        Date start3 = new Date();
        for (int i = 0; i < 100; i++) {
            dict.contains("zzyzfys");
        }
        Date end3 = new Date();

        System.out.println(end3.getTime()-start3.getTime());
    }
}
```

10

Divide & Conquer algorithms

recursive algorithms such as binary search and merge sort are known as *divide & conquer algorithms*

the divide & conquer approach tackles a complex problem by breaking into smaller pieces, solving each piece, and combining into an overall solution

- e.g., to binary search a list, check the midpoint then binary search the appropriate half of the list

divide & conquer is applicable when a problem can naturally be divided into independent pieces

- e.g., merge sort divided the list into halves, conquered (sorted) each half, then merged the results

11

Iterative vs. divide & conquer

many iterative algorithms can naturally be characterized as divide-and-conquer

- sequential search for X in list[0..N-1] =

false	*if N == 0*
true	*if X == list[0]*
sequential search for X in list[1..N-1]	*otherwise*

- sum of list[0..N-1] =

0	*if N == 0*
list[0] + sum of list[1..N-1]	*otherwise*

- number of occurrences of X in a list[0..N-1] =

number of occurrences of X in list[1..N-1]	*if X != list[0]*
1 + number of occurrences of X in list[1..N-1]	*if X == list[0]*

interesting, but not very useful from a practical side (iteration is faster)

12

Euclid's algorithm

one of the oldest known algorithms is Euclid's algorithm for calculating the greatest common divisor (gcd) of two integers

- appeared in Euclid's *Elements* around 300 B.C., but may be even 200 years older
- defines the gcd of two numbers recursively, in terms of the gcd of smaller numbers

```
/** Calculates greatest common divisor of a and b
 *    @param a a positive integer
 *    @param b a positive integer (a >= b)
 *    @return the GCD of a and b
 */
public int gcd(int a, int b) {
    if (b == 0) {
        return a;
    }
    else {
        return gcd(b, a % b);
    }
}
```

e.g.., gcd(32, 12)	= gcd(12, 8) = gcd(8, 4) = gcd(4, 0) = 4
e.g.., gcd(1024, 96)	= gcd(96, 64) = gcd(64, 32) = gcd(32, 0) = 32
e.g.., gcd(17, 5)	= gcd(5, 2) = gcd(2, 1) = gcd(1, 0) = 1

if the larger number has N digits,

- Euclid's algorithm requires at most O(N) recursive calls
- however, each (a % b) requires O(N) steps

→$O(N^2)$

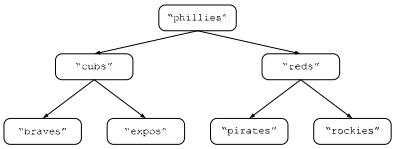
there is no known algorithm with better big-Oh (but is possible to reduce constants)

Multidimensional divide & conquer

we will see later that divide & conquer is especially useful when manipulating multidimensional structures

- e.g., print values in a binary tree

```
public void traverse(Node root) {
    if (root != null) {
        traverse(root.getLeft());
        System.out.println(root.getValue());
        traverse(root.getRight());
    }
}
```



- e.g., find the distance of the closest pair of points in a space

1. LDist = distance of closest pair in left half
2. RDist = distance of closest pair in right half
3. LClose = set of points whose x-coord are within min(LDist,RDist) to the left of center
4. RClose = set of points whose x-coord are within min(LDist,RDist) to the right of center
5. answer = min(LDist, RDist, distance($LClose_i$,$RClose_j$))