CSC 427: Data Structures and Algorithm Analysis

Fall 2011

Divide & conquer (part 1)

- divide-and-conquer approach
- familiar examples: binary search, merge sort, quick sort
- graphics example: closest points
- tree recursion
- HW2 review

1

X & Conquer algorithms

many algorithms solve a problem by breaking it into subproblems, solving each subproblem (often recursively), then combining the results

- if the subproblem is proportional to the list size (e.g., half), we call it *divide-and-conquer*
- if the subproblem is reduced in size by a constant amount, we call it *decrease-and-conquer*
- if the subproblem is a transformation into a simpler/related problem, we call it *transform-and-conquer*

divide-and-conquer is probably the best known algorithmic approach

e.g., binary search as divide-and-conquer

1. if list size = 0, then FAILURE
2. otherwise, check midpoint of list
   a) if middle element is the item being searched for, then SUCCESS
   b) if middle element is > item being searched for, then binary search the left half
   c) if middle element is < item being searched for, then binary search the right half

2

Iteration vs. divide & conquer

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

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

  { 0 — *if N == 0*
  { list[N/2] + sum of list [0..N/2-1] + sum of list[N/2+1..N-1] — *otherwise*

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

  { 0 — if N == 0
  { 1 + num of occurrences of X in list[0..N/2-1] + num of occurrences of X in list[N/2+1..N-1] — if X == list[N/2]
  { 0 + num of occurrences of X in list[0..N/2-1] + num of occurrences of X in list[N/2+1..N-1] — if X != list[N/2]

recall: Cost(N) = 2Cost(N/2) + C ➔ O(N), so no real advantage in these examples over brute force iteration

3

Merge sort

we have seen divide-and-conquer algorithms that are more efficient than brute force

e.g., merge sort list[0..N-1]

1. if list N <= 1, then DONE
2. otherwise,
   a) merge sort list[0..N/2]
   b) merge sort list[N/2+1..N-1]
   c) merge the two sorted halves

recall: Cost(N) = 2Cost(N/2) +CN ➔ O(N log N)

- merging is O(N), but requires O(N) additional storage and copying

4

Quick sort

Collections.sort implements quick sort, another O(N log) sort which is faster in practice

e.g., quick sort list[0..N-1]

1. if list N <= 1, then DONE
2. otherwise,
   a) select a pivot element (e.g., list[0], list[N/2], list[random], …)
   b) partition list into [items < pivot] + [items == pivot] + [items > pivot]
   c) quick sort the < and > partitions

best case: pivot is median

$Cost(N) = 2Cost(N/2) + CN \rightarrow O(N \log N)$

worst case: pivot is smallest or largest value

$Cost(N) = Cost(N-1) + CN \rightarrow O(N^2)$

5

Quick sort (cont.)

average case: O(N log N)

there are variations that make the worst case even more unlikely

- switch to selection sort when small (as in HW3)
- median-of-three partitioning

  instead of just taking the first item (or a random item) as pivot, take the median of the first, middle, and last items in the list

  ✓ if the list is partially sorted, the middle element will be close to the overall median

  ✓ if the list is random, then the odds of selecting an item near the median is improved

refinements like these can improve runtime by 20-25%

however, $O(N^2)$ degradation is possible

6

Closest pair

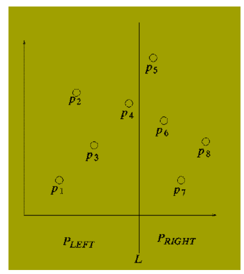
given a set of N points, find the pair with minimum distance

- brute force approach:
  consider every pair of points, compare distances & take minimum

  big Oh? $O(N^2)$

- there exists an O(N log N) divide-and-conquer solution

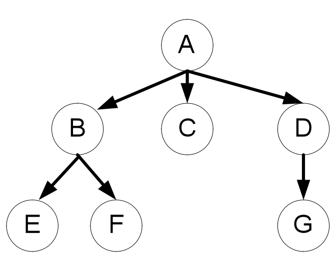
1. sort the points by x-coordinate
2. partition the points into equal parts using a vertical line in the plane
3. recursively determine the closest pair on left side (Ldist) and the closest pair on the right side (Rdist)
4. find closest pair that straddles the line, each within min(Ldist,Rdist) of the line (can be done in O(N))
5. answer = min(Ldist, Rdist, Cdist)



Trees & divide-and-conquer

a tree is a nonlinear data structure consisting of nodes (structures containing data) and edges (connections between nodes), such that:

- one node, the *root,* has no *parent* (node connected from above)
- every other node has exactly one parent node
- there is a unique path from the root to each node (i.e., the tree is connected and there are no cycles)

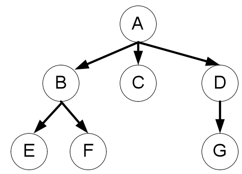


nodes that have no children (nodes connected below them) are known as *leaves*

Recursive definition of a tree

trees are naturally recursive data structures:

- the empty tree (with no nodes) is a tree
- a node with subtrees connected below is a tree



empty tree

tree with 1 node (empty subtrees)

tree with 7 nodes

a tree where each node has at most 2 subtrees (children) is a *binary* tree

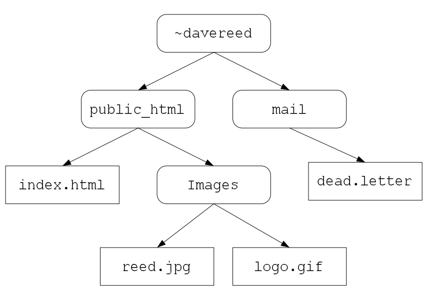
9

Trees in CS

trees are fundamental data structures in computer science

example: file structure

- an OS will maintain a directory/file hierarchy as a tree structure
- files are stored as leaves; directories are stored as internal (non-leaf) nodes



descending down the hierarchy to a subdirectory
⇳
traversing an edge down to a child node

DISCLAIMER: directories contain links back to their parent directories, so not strictly a tree

10

Recursively listing files

to traverse an arbitrary directory structure, need recursion

to list a file system object (either a directory or file):

1. print the name of the current object
2. if the object is a directory, then
   - recursively list each file system object in the directory

in pseudocode:

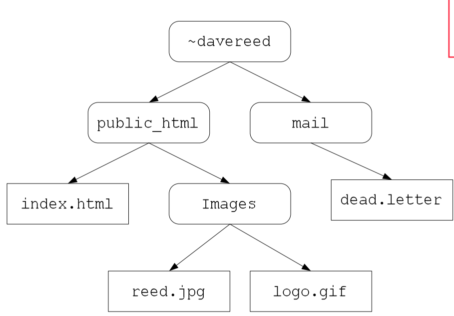
```
public static void ListAll(FileSystemObject current) {
  System.out.println(current.getName());
  if (current.isDirectory()) {
    for (FileSystemObject obj : current.getContents()) {
      ListAll(obj);
    }
  }
}
```

11

Recursively listing files

```
public static void ListAll(FileSystemObject current) {
  System.out.println(current.getName());
  if (current.isDirectory()) {
    for (FileSystemObject obj : current.getContents()) {
      ListAll(obj);
    }
  }
}
```

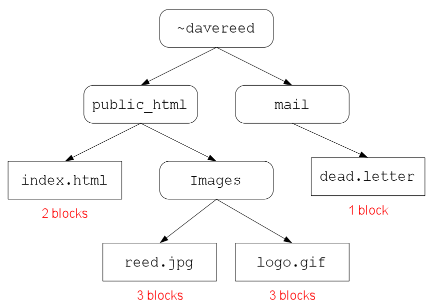
this method performs a *pre-order traversal*: prints the root first, then the subtrees



12

UNIX du command

in UNIX, the du command lists the size of all files and directories



from the ~davereed directory:

```
unix> du -a
2  ./public_html/index.html
3  ./public_html/Images/reed.jpg
3  ./public_html/Images/logo.gif
7  ./public_html/Images
10 ./public_html
1  ./mail/dead.letter
2  ./mail
13 .
```

```
public static int du(FileSystemObject current) {
   int size = current.blockSize();
   if (current.isDirectory()) {
     for (FileSystemObject obj : current.getContents()) {
       size += du(obj);
     }
   }
   System.out.println(size + " " + current.getName());
   return size;
}
```

this method performs a *post-order traversal*: prints the subtrees first, then the root

13

HW2 v. 1

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

public class QuizDriver {
    public static void main(String[] args) {
        ClassList students = new ClassList();

        System.out.println("Enter the file name: ");
        Scanner input = new Scanner(System.in);
        String filename = input.next();

        try {
            Scanner infile = new Scanner(new File(filename));
            while (infile.hasNext()) {
                String lastName = infile.next();
                String firstName = infile.next();
                int quizNum = infile.nextInt();
                int points = infile.nextInt();
                int possible = infile.nextInt();
                students.recordQuiz(lastName, firstName, quizNum, points, possible);
            }
            students.displayAll();
        }
        catch (java.io.FileNotFoundException e) {
            System.out.println("FILE NOT FOUND");
        }
    }
}
```

similar to the WinLoss League example, can have driver class that reads scores from file and processes

- real work is passed on to ClassList

note: driver focuses on I/O - as little problem-specific logic as possible

14

HW2 v. 1

```
import java.util.TreeMap;

public class ClassList {
    private TreeMap<String, StudentRecord> stuMap;

    public ClassList() {
        this.stuMap = new TreeMap<String, StudentRecord>();
    }

    public void recordQuiz(String lastName, String firstName, int quizNum,
                           int points, int possible) {
        String name = lastName + " " + firstName;
        if (!this.stuMap.containsKey(name)) {
            this.stuMap.put(name, new StudentRecord(name));
        }
        this.stuMap.get(name).recordQuiz(quizNum, points, possible);
    }

    public void displayAll() {
        for (String str : this.stuMap.keySet()) {
            System.out.println(this.stuMap.get(str));
        }
    }
}
```

`ClassList` stores student info
- construct with a name
- can call `recordQuiz` to store individual quizzes

since a `StudentRecord` has a `toString` method, can print easily

15

HW2 v. 1

```
public class StudentRecord {
    private String name;
    private int totalPoints;
    private int totalPossible;

    public StudentRecord(String name) {
        this.name = name;
        this.totalPoints = 0;
        this.totalPossible = 0;
    }

    public void recordQuiz(int quizNum, int pointsEarned, int pointsPossible) {
        this.totalPoints += pointsEarned;
        this.totalPossible += pointsPossible;
    }

    public double quizAverage() {
        if (this.totalPossible == 0) {
            return 0.0;
        }
        else {
            return 100.0*this.totalPoints/this.totalPossible;
        }
    }

    public String toString() {
        return this.name + " " + this.quizAverage();
    }
}
```

for v. 1, `StudentRecord` stores name and point totals for the quizzes
- no need to store individual quizzes
- however, can't just store quiz percentages since quizzes are not equally weighted

16

HW2 v. 2

```
import java.util.HashMap;

public class StudentRecord {
    private HashMap<Integer, Quiz> quizzes;
    private String name;

    public StudentRecord(String name) {
        this.name = name;
        quizzes = new HashMap<Integer, Quiz>();
    }

    public void recordQuiz(int quizNum, int pointsEarned, int pointsPossible) {
        this.quizzes.put(quizNum, new Quiz(pointsEarned, pointsPossible));
    }

    public double quizAverage() {
        int totalPoints = 0;   int totalPossible = 0;
        for (Integer i : this.quizzes.keySet()) {
            totalPoints += this.quizzes.get(i).getPoints();
            totalPossible += this.quizzes.get(i).getPossible();
        }

        if (totalPossible == 0) {
            return 0.0;
        }
        else {
            return 100.0*totalPoints/totalPossible;
        }
    }

    public String toString() {
        return this.name + " " + this.quizAverage();
    }
}
```

for v. 2, `QuizDriver` and `ClassList` are unchanged!

- file info is the same
- task of reading & storing is same
- how to store & calculate avg is different → `StudentRecord`

new version uses a Map to store quizzes, with quiz # as key

- using `put` operator, any retake will overwrite the old quiz

17

HW2 v. 2

```
public class Quiz {
    private int numPoints;
    private int numPossible;

    public Quiz(int numPoints, int numPossible) {
        this.numPoints = numPoints;
        this.numPossible = numPossible;
    }

    public int getPoints() {
        return this.numPoints;
    }

    public int getPossible() {
        return this.numPossible;
    }
}
```

need a class to store the two quiz components

- could define our own
- or, could repurpose an existing class (e.g., `Point`)

recall: you want classes to be loosely coupled

- object/method behavior should not be tied to implementation/sequencing details

common design flaw in HW2:

- `calculateAvg` method calculates average and stores in a field
- `getAvg` method accesses the field and returns the average
- correct behavior requires `calculateAvg` is called since last quiz grade added

18

HW2 v. 3

```
public class StudentRecord implements Comparable<StudentRecord> {
    ...

    public int compareTo(StudentRecord other) {
        double thisAvg = this.quizAverage();
        double otherAvg = other.quizAverage();
        if (thisAvg > otherAvg) {
            return -1;
        }
        else if (thisAvg < otherAvg) {
            return 1;
        }
        else {
            return this.name.compareTo(other.name);

        }
    }
}

public class ClassList {
    ...

    public void displayAll() {
        TreeSet<StudentRecord> students = new TreeSet<StudentRecord>();
        for (String str : this.stuMap.keySet()) {
            students.add(this.stuMap.get(str));
        }

        for (StudentRecord stu : students) {
            System.out.println(stu);
        }
    }
}
```

to list averages in descending order, only minor changes to `ClassList` & `StudentRecord`

- `StudentRecords` must be `Comparable`
- `ClassList` can then sort the `StudentRecords`

19