CSE 373 20su: Exam I

Note: This exam was originally offered via Gradescope. Free response fields are omitted in this PDF reproduction.

1. Instructions

- This exam is written to be completed in 1-2 hours. To maximize flexibility, it will be available for 48 hours. **It is due strictly at 11:59pm PDT on Saturday, July 25.** You may not use late days on this exam.
- You may work in multiple sessions, and submit as many times as you'd like (only your last submission will be graded).
- You may work alone, or in groups of up to 8 (and submit a single exam). Group submission instructions:
  - First, have one person click "Save Answer" on any question. Then, scroll to the very bottom and click "Submit & View Submission".
  - You can now add your group members in the upper right corner, just like a programming assignment.
  - Then, you can click "Resubmit" to edit your answers as many times as you want, with the same group.
  - Be careful not to have two people edit answers at the same time (Gradescope may lose something). We recommend coordinating through a different platform and having one person submit for the whole group.
- This test is open-note and open-internet. However, you are not permitted to share these questions or get help from anybody not enrolled in 373 20su, such as former students.
- During the exam, you may ask clarification questions on Piazza, in office hours, or via email to cse373-staff@cs.washington.edu. Course staff will not answer questions about course concepts or give hints on specific exam questions.
- Sentence estimates are just to clarify our expectations. You will not be penalized for writing more or less.
- Each question is annotated with approximate corresponding learning objectives from lecture. This is purely extra information and only intended to make the link between what you've learned and what we're testing clearer: don't read too much into them or worry whether you've adequately demonstrated the skill! :)
- Any significant exam clarifications will be posted on Piazza and posted in these instructions.

2. ADTs and Design Decisions

Learning Objectives: Compare the runtime of ArrayList and LinkedList (LEC02), Distinguish the List ADT from its implementations (LEC02), Describe the Stack, Queue, and Map ADTs (LEC03), Compare the runtime of Stack, Queue, and Map operations on a resizable array vs. linked nodes (LEC03).

Scenario: Suppose you've just started a 48-hour exam where you are allowed to work in groups. For some reason, your group decides to start by designing your own version of Google Docs to collaborate. This system allows multiple users to make edits to a document, so you come up with the `ContributionHistory` ADT, which describes how contributions can be saved and accessed. For simplicity, we'll assume that all contributions are appending text, so each contribution has a user (`String`) and text to insert (`String`).

The ADT includes the following methods:

- `addContribution(String user, String text)` – Associates a user with the text they inserted. Until another call is made to addContribution with the same user, this text is considered the last contribution for this user.
- `String undoLastContributionOfUser(String user)` – Removes a user's last contribution from the ContributionHistory, and returns its text. The previous contribution then becomes the "last" contribution, which will be removed/returned if `undoLastContributionOfUser` is called again. If a user has no contributions left to undo, return null. (Presumably, the code using this class would then figure out how to remove that from the

actual document, but all your class needs to do is remove the contribution from its internal records and return it.)

When doing analysis for these problems, let the variable n refer to the total number of contributions in the data structure. You should be able to express all of your bounds in terms of n. For analyzing any arrays in the worst case, you may not assume that resizing doesn't happen.

2.1. Linked Nodes Implementation

Suppose we implement this ADT with a singly-Linked List of contributions, where each contribution is represented by a Pair object containing the user and text. Adding edits is done by inserting a new Pair at the *FRONT* of the list (i.e. index 0). For each method, give a simplified worst-case Big-Theta bound in terms of n (the number of total contributions) for how the method would need to be implemented, or write "N/A" if one doesn't exist. You don't need to justify your answers.

(a) `addContribution` worst case Big-Theta.

(b) `undoLastContributionOfUser` worst case Big-Theta.

2.2. Array Implementation

Now, suppose we implement this ADT with an ArrayList of contributions, again where each contribution is represented by a Pair object containing the user and text. As before, adding edits is done by inserting a new Pair at the *FRONT* of the list (i.e. index 0). For each method, give a simplified worst-case Big-Theta bound in terms of n (the number of total contributions) for how the method would need to be implemented, or write "N/A" if one doesn't exist. You don't need to justify your answers.

(a) `addContribution` worst case Big-Theta.

(b) `undoLastContributionOfUser` worst case Big-Theta.

2.3. Custom Implementation

Come up with your own implementation of the ADT that optimizes for the `addContribution` and `undoLastContributionOfUser` runtimes. You may use any data structures implementing the List, Stack, Queue, Deque, and Map ADTs covered in the course, and you may use any number of data structures (or combine them). Describe your implementation below, then give a worst case Big-Theta bound for each method. There may be many correct answers to this question.

(a) Describe your implementation. For full credit, we'll look for: (1) you explicitly mention the names of any data structures (and their corresponding ADTs) that you use, (2) your chosen implementation demonstrates you've thought about how to do these tasks reasonably efficiently, and (3) you give a high-level description of what happens in each method that is specific enough to come up with Big-Theta bounds. (2-3 sentences)

(b) `addContribution` worst case Big-Theta.

(c) `undoLastContributionOfUser` worst case Big-Theta.

2.4. Extending the ADT

Suppose we wanted to add a new method called `String findAuthor(String text)` that searches for a particular contribution's text and returns the user who made it. If we were preparing a description of the `ContributionHistory` ADT that includes `findAuthor`, which of the following would be the best description to use?

◯ "Returns the user who made the specified contribution."

◯ "Returns the user who made the specified contribution. If no matching contribution is found, returns null."

◯ "Returns the user who made the specified contribution. Scans through each node starting from the most recent, checks if the contribution texts are equal, and returns the user if so."

◯ "Returns the user who made the specified contribution. Scans through each node starting from the most recent, checks if the contribution texts are equal, and returns the user if so. If no matching contribution is found, returns null."

3. Oh/Omega/Theta Bounds

Learning Objectives: Identify whether Big-Oh (and Big-Omega, Big-Theta) statements about a function are accurate (LEC04), Explain why we can throw away constants when we compute Big-Oh bounds (LEC04), Differentiate between Big-Oh, Big-Omega, and Big-Theta (LEC05), Describe the difference between Case Analysis and Asymptotic Analysis (LEC05).

3.1. Big-Oh

Suppose we know that f(n) is a function in $O(n^2)$. Given that that's all we know about f(n), for each of the following statements, indicate if it is ALWAYS true, NEVER true, or SOMETIMES true (more information about f(n) would be needed to determine if it's true). You do not need to justify your answers.

(a) f(n) is in $O(n^3)$.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES (Need more information)

(b) f(n) is in $O(n)$.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES (Need more information)

(c) f(n) is in $O(0.5n^2)$.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES (Need more information)

(d) For some constant c, $f(n) \leq c \times n^2$ for any value of n.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES (Need more information)

(e) f(n) is in $\Omega(n^3)$.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES (Need more information)

(f) f(n) has a Big-Theta bound.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES (Need more information)

(g) We can find different best and worst case runtimes (depending on its input) for f(n) by performing case analysis.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES (Need more information)

3.2. Big-Theta

Suppose we know that g(n) is a function in $\Theta(n)$. Given that that's all we know about g(n), for each of the following statements, indicate if it is ALWAYS true, NEVER true, or SOMETIMES true (more information about f(n) would be needed to determine if it's true). You do not need to justify your answers.

(a) g(n) is in $O(n)$.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES (Need more information)

(b) g(n) is in $\Theta(n^2)$.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES (Need more information)

(c) g(n) describes a runtime in the best case of case analysis.

◯ ALWAYS

◯ NEVER

◯ SOMETIMES (Need more information)

4. Algorithmic Analysis

Learning Objectives: Come up with Big-Oh, Big-Omega, and Big-Theta bounds for a given function (LEC05), Perform Case Analysis to identify the Best Case and Worst Case for a given piece of code (LEC05).

Scenario: To raise money so he can buy a new laptop that won't catch fire during lecture, Aaron is planning to open a lemonade business with several lemonade stands along a single street in Seattle and needs to figure out where to place the stands. For planning purposes, he decides he will open 3 lemonade stands, and he represents each house and lemonade stand with the number of meters (represented as an integer) from the start of the street.

For example, the following picture would be represented with the input `{4, 5, 9, 11}` for houses, and `{2, 6, 8}` for stands:

The following are two implementations of a method that takes a list of houses and stands (each represented by a number) and returns the number of houses that are "close" to a stand (within 30 meters), to assess proposed stand locations. Assume that the same number won't appear twice in either list (that is, that no two houses or stands are the same number of meters from the start of the street).

For these problems, assume that the number of stands is a constant. **When asked to analyze runtime, do so in terms of n, the size of the houses parameter.**

4.1. Implementation A

```
01  public int countCloseA(List<Integer> houses, List<Integer> stands) {
02    int count = 0;
03    for (int house : houses) {
04      for (int stand : stands) {
05        int distance = Math.abs(house - stand);
06        if (distance < 30) {
07          count++;
08        }
09      }
10    }
11    return count;
12  }
```

(a) What is the simplified Big-Theta runtime in terms of n (the size of houses) for this method in the worst case?

(b) What is the simplified Big-Theta runtime in terms of n (the size of houses) for this method in the best case?

(c) Describe the inputs that would lead to the best and worst case for this method (consider giving example inputs if it would make your description more clear). If there is no difference between the two cases, note that instead (no explanation needed). (1-3 sentences)

4.2. Implementation B

```
01  public int countCloseB(List<Integer> houses, List<Integer> stands) {
02    List<Integer> closeHouses = new ArrayList<>();
03    for (int house : houses) {
04      for (int i = -30; i < 30; i++) {
05        for (int stand : stands) {
06          if (stand == house + i) {
07            closeHouses.add(house);
08          }
09        }
10      }
11    }
12
```

```
13    int count = closeHouses.size();
14
15    // remove duplicates
16    for (int i = 0; i < closeHouses.size(); i++) {
17      for (int j = i; j < closeHouses.size(); j++) {
18        if (closeHouses.get(i).equals(closeHouses.get(j))) {
19          count-;
20        }
21      }
22    }
23    return count;
24  }
```

(a) What is the simplified Big-Theta runtime in terms of n (the size of houses) for this method in the worst case?

(b) What is the simplified Big-Theta runtime in terms of n (the size of houses) for this method in the best case?

(c) Describe the inputs that would lead to the best and worst case for this method (consider giving example inputs if it would make your description more clear). If there is no difference between the two cases, note that instead (no explanation needed). (1-3 sentences)

5. Recursive Code Analysis

Learning Objectives: Model recursive code using a recurrence relation (LEC06), Describe the 3 most common recursive patterns and identify whether code belongs to one of them (LEC06), Use the Master Theorem to characterize a recurrence relation (LEC06), Characterize a recurrence with the Tree Method (LEC07).

5.1. Recursive Code Modeling

Consider the following code. In this question, you will write a recurrence to model the **runtime** (not the result) of this Java method in terms of n, the length of the dinos array. Assume the String .equals method has constant runtime, and creating a new array (e.g. new String[] on line 07) also has constant runtime.

```
01  public int countPairs(String[] dinos) {
02
03    if (dinos.length <= 1) {
04      return 0;
05    }
06
07    String[] nextDinos = new String[dinos.length - 2];
08
09    for (int i = 0; i < nextDinos.length; i++) {
10      nextDinos[i] = dinos[i + 2];
11    }
12
13    if (dinos[0].equals(dinos[1])) {
14      return 1 + countPairs(nextDinos);
15    } else {
16      return countPairs(nextDinos);
17    }
18  }
```

Write a recurrence for the worst-case runtime of this code. If you need to use constants but their exact value isn't known or doesn't matter, use c1, c2, c3, etc.

Follow this recurrence template:

$$T(n) = \begin{cases} \text{BASE\_CASE} & \text{BASE\_CONDITION} \\ \text{RECURSIVE\_CASE} & \text{RECURSIVE\_CONDITION} \end{cases}$$

(a) Give BASE_CASE and BASE_CONDITION:

(b) Give RECURSIVE_CASE and RECURSIVE_CONDITION:

5.2. Characterizing Recurrences

Give a Big-Theta bound for this recurrence. You may use any technique to do so; you do not have to show your work.

$$T(n) = \begin{cases} 25 & \text{if } n = 0 \\ 3T(\frac{n}{4}) + n^2 & \text{otherwise} \end{cases}$$

5.3. Recursive Patterns & The Tree Method

Consider these two recursive methods. Do your analysis in terms of the parameter n.

```
01  public int f1(int n) {
02    if (n <= 1) {
03      return 1;
04    } else {
05      return f1(n / 2) + 1;
06    }
07  }
08
09  public int f2(int n) {
10    if (n <= 1) {
11      return 1;
12    } else {
13      return f2(n / 2) + f2(n / 2) + 1;
14    }
15  }
```

Which of the following statements are true about the recursive call trees for these two methods, as would be visualized when applying the tree method? Select all that apply.

- ☐ f1 and f2 differ in the height of their recursive call trees.
- ☐ f1 and f2 differ in the total number of nodes in their recursive call trees.
- ☐ f1 and f2 differ in the total amount of work done in their recursive call trees.
- ☐ f1 and f2 differ in the total amount of work done in the last level of their recursive call trees.

6. Hash Maps

Learning Objectives: Differentiate between the "worst" and "in practice" runtimes of a Separate Chaining Hash Map, and describe what assumptions the latter involves (LEC08), Compare the relative pros/cons of various Map implementations (LEC08), Describe the properties of a good hash function and the role of a hash function in making a Hash Map efficient (LEC09), Trace operations in a Separate Chaining Hash Map on paper (such as insertion, getting an element, resizing) (LEC08).

For these problems, assume we have a simplified Separate Chaining Hash Map. Pseudocode for the relevant methods of the data structure are provided below, intended to be the same as the one presented in lecture except for the capitalized portion in resize:

```
01  public V get(K key) {
02    // compute bucket by hashing key and modding by number of buckets
03    // iterate through that bucket in order. for each element:
```

```
    // if key matches, return value
    // if not, go to next element. if on last element, return null
}
```

```
public void put (K key, V value) {
  // compute bucket by hashing key and modding by number of buckets
  // iterate through that bucket in order. for each element:
    // if key matches, replace old value with the the new value
    // if not, go to next element. if on last element, stop iterating
  // if load factor over threshold: call resize()
}
```

```
private void resize() {
  // create new array of buckets twice as large
  // go through all key/value pairs in all buckets in RANDOM
  //        order, compute bucket as in put, and add to end of that bucket
}
```

6.1. Resizing

Which of the following is a correct statement about why we resize our hash map?

- ◯ To prevent it from returning incorrect results.
- ◯ To keep the best case `get` runtime fast.
- ◯ To limit the average length of the chains.
- ◯ To limit the maximum length of the chains.

6.2. Hash Functions

Suppose we modify the Separate Chaining Hash Map implementation to use the following hash function for every key instead of calling Java's built-in `hashCode` method:

```
int hashCode(K key) {
  return 1;
}
```

(a) Would this change the runtime of the "In Practice" case for `get`?

- ◯ Yes
- ◯ No

(b) Would this change the runtime of the worst case for `get`?

- ◯ Yes
- ◯ No

6.3. put()

Suppose we modify the Separate Chaining Hash Map put method to always insert at the front of each chain (not replacing anything if the key already exists somewhere else in that chain), rather than iterating through to look for the key:

```
public void put(K key, V value) {
  // compute bucket by hashing key and modding by number of buckets
  // insert new entry with key/value pair at front of that bucket
  // if load factor over threshold: call resize()
}
```

Note that making this change slightly changes the invariants of the data structure. Assume all other methods are the same as above, and that we do *not* use the replacement hash code from 6.2.

(a) Suppose we insert the keys 0 - 999 into the modified Hash Map (with any values), then remove keys 0 - 499. In the "In Practice" case, how would you expect the call 'get(1000)' on the modified Hash Map to perform, relative to the same call on the original presented above? Ignore any external factors (e.g. other processes running in the background).

- ◯ We'd expect the get(1000) call in the modified Hash Map to take **about the same** amount of time as in the original.
- ◯ We'd expect the get(1000) call in the modified Hash Map to be **faster** than in the original.
- ◯ We'd expect the get(1000) call in the modified Hash Map to be **slower** than in the original.

Why?

(b) Consider the following sequence of operations, executed **in order**. Suppose we run this code once for the original Hash Map, and once for the modified Hash Map.

```
01  put(3, 'a');
02  put(5, 'b');
03  char g1 = get(3);
04  put(3, 'c');
05  char g2 = get(3);
06  put(5, 'd'); // triggers a resize
07  char g3 = get(5);
```

Indicate below which of these variables **may** contain a different result for the two versions. Select all that apply (or None, if applicable).

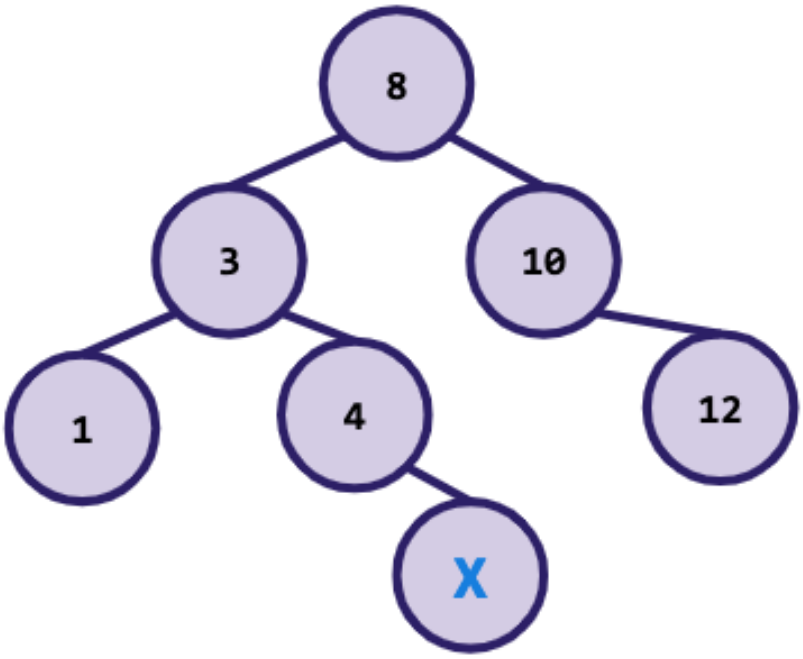
- ☐ g1
- ☐ g2
- ☐ g3
- ☐ None

7. Trees

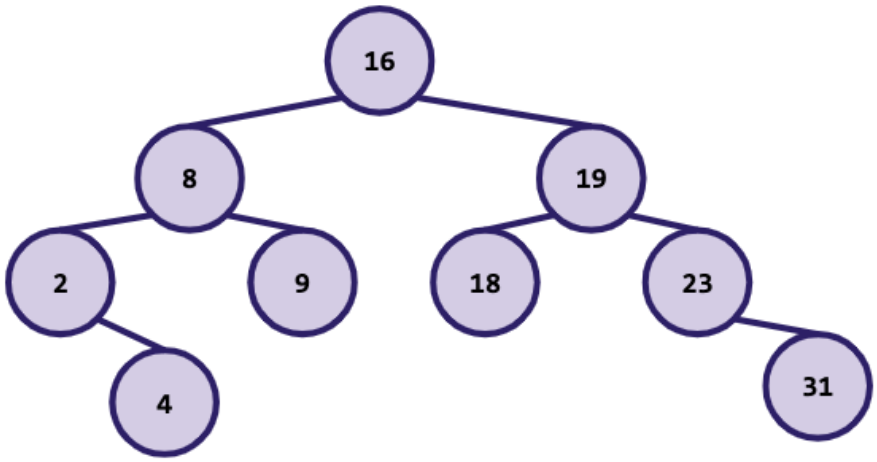
Learning Objectives: Apply the BST invariant and describe its implications (LEC10), Describe how AVL Rotations and the AVL invariants lead to an efficient runtime (LEC10), Identify invariants for data structures (LEC03), Evaluate invariants based on their strength and maintainability, and come up with the invariants for data structure implementations (LEC09).

7.1. Binary Search Trees

Consider this binary search tree of integers. Assume duplicates aren't allowed. What possible values could go in spot X?



7.2. AVL Rotations

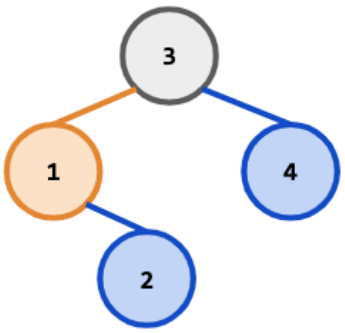


7.3. Tree Invariants

Suppose we have a BST data structure and want to impose an additional invariant that will limit its height to $\Theta(\log n)$, similar to the AVL invariant.

Proposal: "The overall tree must have an equal number of nodes that are the left child of their parent node and nodes that are the right child of their parent node".

Here is a sample tree that does not fit the invariant:



Total number of nodes that "are the left child of their parent node":	1
Total number of nodes that "are the right child of their parent node":	2
Does this tree satisfy the invariant?	No, because $1 \neq 2$

For this proposed invariant, indicate whether or not following it will limit the height.

- ◯ Works (limits the height to $\Theta(\log n)$
- ◯ Doesn't Work (doesn't limit the height)

Justify your answer: if it will limit the height, explain why the tree cannot become unbalanced. If not, describe a tree that follows the invariant but is not balanced. (1-2 sentences)