CSE 373 20su: Exam I Sample Solution

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. **Solution:**

> $\Theta(1)$. Since the underlying Linked List has a pointer to the front element, no iteration is needed – it simply needs to add a new node and rearrange pointers taking constant time.

(b) `undoLastContributionOfUser` worst case Big-Theta. **Solution:**

> $\Theta(n)$. Since we don't know what order users made contributions to the document, and we store contributions from all users in a single linked list with the most recent first, in the worst case the last contribution made by the specified user was the very first and the implementation has to iterate through every contribution to find and remove it.

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. **Solution:**

> $\Theta(n)$. Since we insert the new contribution at index 0 of the underlying array, the implementation needs to shift all other elements in the array over to make space, requiring a $\Theta(n)$ operation. Note that this question also doesn't specify we can assume no resizing, in which case copying all elements of the underlying array would also be a $\Theta(n)$ operation – although it wouldn't change the overall simplified bound.

(b) `undoLastContributionOfUser` worst case Big-Theta. **Solution:**

> $\Theta(n)$. Since we don't know what order users made contributions to the document, and we store contributions from all users in a single array with the most recent first, in the worst case the last contribution made by the specified user was the very first and the implementation has to iterate through every contribution to find and remove it.

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)
Solution:

> There are many correct answers to this question. The following is one example:
>
> We could be using a Hash Map data structure (implementing the Map ADT) in combination with Linked Stack data structures (implementing the Stack ADT), where the map keys are the names of users (Strings) and the map values are stacks containing their edits (Strings). The `addContribution` method would look up the given user in the map to get their corresponding stack and push the new contribution on top. The `undoLastContributionOfUser` method would look up the given user in the map to get their corresponding stack and the pop the last contribution off the top, returning the result.
>
> A common deduction was giving a description that was not unambiguous enough to verify the Big-Theta bounds in parts (b) and (c) (for example, not mentioning both the map lookup and the stack operation here). We awarded full credit for using "Stack" as a data structure and an ADT due to the confusing Java `Stack` class.

(b) `addContribution` worst case Big-Theta. **Solution:**

> There are many correct answers to this question, as long as they match part (a). In our example part(a), the answer would be $\Theta(1)$ in the "in practice" case or $\Theta(n)$ in the worst case. To reflect real-world decisionmaking, we awarded full credit for not mentioning the "in practice" case and simply assuming $\Theta(1)$ lookup in a Hash Map or assuming an underlying array wouldn't resize.

(c) `undoLastContributionOfUser` worst case Big-Theta. **Solution:**

> There are many correct answers to this question, as long as they match part (a). In our example part(a), the answer would be $\Theta(1)$ in the "in practice" case or $\Theta(n)$ in the worst case. To reflect real-world decisionmaking, we awarded full credit for not mentioning the "in practice" case and simply assuming $\Theta(1)$ lookup in a Hash Map or assuming an underlying array wouldn't resize.

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."

Solution:

B: “Returns the user who made the specified contribution. If no matching contribution is found, returns null.” Note that this description is going into the ADT, so we don’t want to choose C or D because they describe implementation details that may not be true for every data structure that implements the ADT (for example, a Map & Stack). We also prefer B to A because it fully describes the behavior of the method (the ”contract” that data structures must follow) by mentioning the edge case where the contribution is not found.

However, we also gave full credit for choosing A under the argument that ”null” might be considered as a language-specific detail, and therefore not belong in an ADT.

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)

Solution:

ALWAYS. Any function that is upper-bounded by n^2 must also be upper-bounded by n^3 according to the definition of Big-Oh.

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

- ◯ ALWAYS
- ◯ NEVER
- ◯ SOMETIMES (Need more information)

Solution:

SOMETIMES. Any function that is upper-bounded by n must also be upper-bounded by n^2 (so functions exist where this is true), but it’s also possible to have a function in $\Theta(n^2)$ that is in $O(n^2)$ but not $O(n)$ (so functions exist where this is not true).

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

- ◯ ALWAYS
- ◯ NEVER
- ◯ SOMETIMES (Need more information)

Solution:

ALWAYS. Intuitively, this can be explained because the definition of Big-Oh allows a constant factor, so we could just double whatever constant factor was used for $O(n^2)$ to show that the function is also in $O(0.5n^2)$.

Formally, if $f(n)$ is in $O(n^2)$, by the definition of Big-Oh we know there must exist some pair of values c, n_0 such that $f(n) \leq c \times n^2$ for $n \geq n_0$. Since the new function we're considering is only a constant factor (0.5) multiplied by n^2, we can use that fact to determine that $f(n)$ is in its Big-Oh as well. We can choose a new constant factor $d = 2c$, and we would get $f(n) \leq d \times 0.5n^2$ for $n \geq n_0$, so this statement is always true if $f(n)$ is in $O(n^2)$.

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

- ◯ ALWAYS
- ◯ NEVER
- ◯ SOMETIMES (Need more information)

Solution:

SOMETIMES. This statement is close to the definition of Big-Oh, but recall that the definition of Big-Oh has two components: the constant factor c, and **the starting value** n_0. This statement doesn't take n_0 into account, so it does not always have to be true for a function to be in $O(n^2)$, which might only be upper-bounded by $c \times n^2$ after a certain value of n (so it's not true to say "for *any* value of n").

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

- ◯ ALWAYS
- ◯ NEVER
- ◯ SOMETIMES (Need more information)

Solution:

NEVER. Any function that is upper-bounded by n^2 cannot also be lower-bounded by n^3, by the definitions of Big-Oh and Big-Omega.

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

- ◯ ALWAYS
- ◯ NEVER
- ◯ SOMETIMES (Need more information)

Solution:

SOMETIMES. If f(n) is also in $\Omega(n^2)$, then it will have a Big-Theta bound, but we'd need more information to know. There are functions that have different Big-Oh and Big-Omega bounds and therefore do not have a Big-Theta (for example, the isPrime function from lecture).

(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)

Solution:

NEVER. Based on our definition, case analysis is a tool that can only be used on code to come up with different runtime functions; it is not meaningful to do case analysis on a function, because each case has exactly one runtime function that describes it. It's also worth noting that different cases exist only due to variation in variables other than n (the variable we've chosen to do our asymptotic analysis in terms of). Since $f(n)$ has no inputs other than n, even if it were possible to do case analysis on a function, $f(n)$ would not have different best and worst cases because there is no other source of variation besides n that could impact the cases.

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)

Solution:

ALWAYS. By the definition of Big-Theta, a function in $\Theta(n)$ must also be in $O(n)$ and $\Omega(n)$.

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

◯ ALWAYS

◯ NEVER

◯ SOMETIMES (Need more information)

Solution:

NEVER. Since Big-Theta describes a tight bound, a function cannot belong to more than one Big-Theta because it cannot have more than one tight bound.

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

◯ ALWAYS

◯ NEVER

◯ SOMETIMES (Need more information)

Solution:

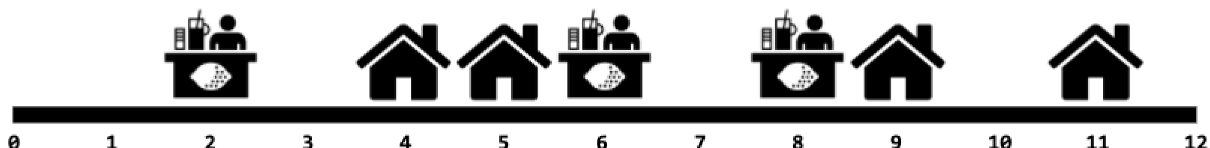
SOMETIMES. Since case analysis is a step that comes before asymptotic analysis and is applied to code to produce runtime functions for the different cases of that code, when given a function we have no way of knowing what case it corresponds to. It's possible to have a runtime function that is in $\Theta(n)$ that describes the best, worst, "in-practice", or any other case of the code. For example, consider if the best case of the code were modeled by the isPrime function, which doesn't have a Big-Theta.

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?
Solution:

> $\Theta(n)$. Since the number of stands is a constant, we only need to consider the loop over all of the houses, which has n iterations. The worst case here wouldn't differ from the best case, because the only thing that could change the runtime would be whether we run line 07 (`count++`), which is only a constant amount of work and therefore can't affect the runtime.

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

$\Theta(n)$. Again since the number of stands is a constant, we only need to consider the loop over all of the houses, which has n iterations. Note that the best case cannot be where $n = 0$: remember that only sources of variation other than n can be the reason for a best or worst case. In other words, we still need to be able to come up with a runtime function describing the best case that is in terms of all n.

(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) **Solution:**

There is no difference between the best and worst cases for this function.

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? **Solution:**

$\Theta(n^2)$. In the worst case, every house in the houses list would be close to a stand, so they would all be added to the closeHouses list in the first loop (which still only has $\Theta(n)$ runtime because 30 and the size of the stands list are constants). Then, in the second loop where we remove duplicates, in the worst case closeHouses would contain n elements so the doubly-nested loop would run $\Theta(n^2)$ times. Since the n^2 term dominates in $\Theta(n^2 + n)$, we're left with $\Theta(n^2)$ as the simplified runtime.

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

$\Theta(n)$. In the best case, no house in the houses list is close to a stand, so none would be added to the closeHouses list in the first loop. The first loop still has $\Theta(n)$ runtime, because it has to inspect every

house even if none are added. However, in the second loop the size of closeHouses would be 0 (regardless of the original size of houses), so it would add no additional runtime.

(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) **Solution:**

The best and worst cases would be different. In the best case, none of the houses in the houses list would be close to any of the lemonade stands in the stands list, but in the worst case all of the houses would be close to a lemonade stand.

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: **Solution:**

c1 n $\leq$1. The base case is triggered when we enter the if statement on line 03, and only some constant amount of work is done.

(b) Give RECURSIVE_CASE and RECURSIVE_CONDITION: **Solution:**

T(n-2) + n + c2 otherwise. The recursive case involves a linear amount of work (the loop on lines 09-11 that iterates through all of dinos, minus the constant 2), and the constant c2 is added to represent that there will be some extra constant work throughout the recursive case. Importantly, even though there are two recursive calls to countPairs in the body of the method, only one call can happen, so there is no 2 multiplied by T(n-2). We use n-2 because that's how much the input size changes on each recursive call.

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.

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

Solution:

$\Theta(n^2)$. Either the Master Theorem or Tree Method would work for this recurrence and give the same result.

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.

Solution:

False, True, True, True. The first statement is false because the height of both trees will be the same: the parameter in the recursive calls for both methods is the same. The second statement is true because f1 will generate only one child call for each call, but f2 will generate two child calls for each call and will therefore have many more nodes in its tree. The third statement is true because the same amount of work is done in each node for f1 and f2, so if they differ in the number of total nodes, they will also differ in the amount of total work. The fourth statement is true because even though each *node* in the last level of the tree does the same amount of work in f1 and f2, there will be more total nodes in the last level of the f2 tree.

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:
04      // if key matches, return value
05      // if not, go to next element. if on last element, return null
06  }
```

```
01  public void put (K key, V value) {
02    // compute bucket by hashing key and modding by number of buckets
03    // iterate through that bucket in order. for each element:
04      // if key matches, replace old value with the the new value
05      // if not, go to next element. if on last element, stop iterating
06    // if load factor over threshold: call resize()
07  }
```

```
01  private void resize() {
02    // create new array of buckets twice as large
03    // go through all key/value pairs in all buckets in RANDOM
04    //        order, compute bucket as in put, and add to end of that bucket
05  }
```

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.

Solution:

C: To limit the average length of the chains.

A is incorrect because even if we didn't resize the hash map, we'd have fewer chains to distribute keys among but would still be able to scan through the corresponding chain to always return the correct result. B is incorrect because the best case runtime for `get` is always going to be constant time – that happens when the element we are searching for is in the front of its chain (it doesn't matter how long the chains are in the best case). C is correct because resizing is based on the load factor, which is an expression of the average length of the chains, and we resize to keep the average chain length short (so the "in practice" case will have an efficient lookup runtime). D is incorrect because even if we resize, we can't guarantee that the maximum length of the chains will be limited – resizing is only triggered by the average chain length, and while it might tend to have an effect on the maximum length it doesn't specifically limit it.

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:

```
01  int hashCode(K key) {
02    return 1;
03  }
```

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

◯ Yes

◯ No

Solution:

Yes, this would change the runtime of the "In Practice" case. All of the keys would be hashed to the same index, and thus be in the same chain, so no matter how many chains we resize to, we'll no longer be able to assume an even distribution of keys across those chains which is a fundamental assumption for the "In Practice" case.

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

◯ Yes

◯ No

Solution:

No, this would not change the runtime of the worst case. The worst case is already that all keys would be hashed to the same chain, which is exactly the same as the result of using this hash function.

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:

```
01  public void put(K key, V value) {
02    // compute bucket by hashing key and modding by number of buckets
03    // insert new entry with key/value pair at front of that bucket
04    // if load factor over threshold: call resize()
05  }
```

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?

Solution:

A: We'd expect the get(1000) call in the modified Hash Map to take **about the same** amount of time as in the original.

Despite the modifications to the put method, in the get method we'd still have to iterate through the same number of keys stored in the corresponding chain, because there would be no more or fewer keys stored there as compared to the original Hash Map if we never call put with duplicate keys.

A common deduction was describing the runtime of the **`put`** method instead of the **`get`** method, which is what the question is asking.

(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

Solution:

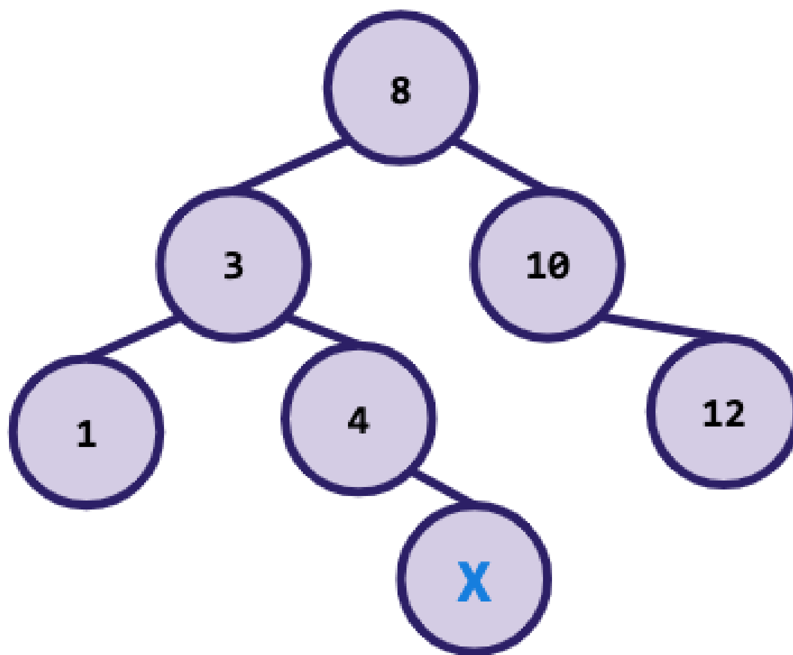
g3 may contain a different result for the two versions, but g1 and g2 will not be different. The new version of put leaves old values corresponding to a key in a chain, and simply "shadows" them with the new values it places in front. g1 will be the same (no change in the underlying map), and g2 will still be the same because the get method will stop at the first matching entry for the key 3 and ignore the old entry left behind it. However, when resizing, note that we go through the entries in the entire map in random order: that means we cannot predict whether the most recent value or an old value corresponding to a key will be present at the front of its chain in the resized hash map, and g3 could be a different result with this new version of put.

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

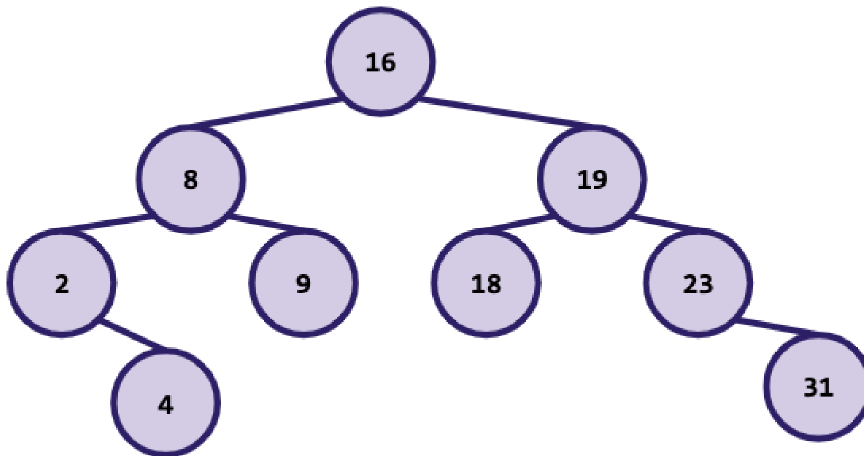
Consider this binary search tree of integers. Assume duplicates aren't allowed. What possible values could go in spot X?



Solution:

The value stored in node X must be greater than 4 but also less than 8 to satisfy the BST invariant. Since duplicates aren't allowed, the only possible values of X are 5, 6, or 7.

7.2. AVL Rotations



Solution:

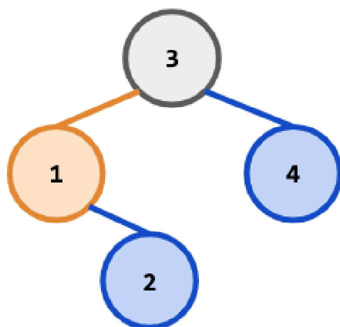
After adding 6 as the right child of the 4, we would perform a left rotation on nodes 2, 4, and 6 to restore the AVL invariant.

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 ≠ 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)

Solution:

Doesn't Work. To show this invariant doesn't work, we can describe a tree that satisfies the invariant but is not balanced to height $\Theta(\log n)$. For example, consider a tree where the root's left child is the start of a degenerate tree consisting of only left children, and the root's right child is the start of a degenerate tree consisting of only right children. If each degenerate tree has an equal number of nodes, this tree will satisfy the invariant, but its height is only a constant factor of n and is not limited to $\log n$.