CSC017 Midterm Fall 2024 Problems

Department of Computer Science

Hofstra University

Questions 1-3, 10

- What is polymorphism in the context of inheritance?
  - A) The ability to create multiple objects of the same class
  - B) The ability to override methods in a subclass
  - C) The ability for a superclass reference to call the appropriate subclass method
  - D) The ability to have multiple constructors in a class
  - Answer: B, C both are OK
- Which statement about private methods is true?
  - A) They can be overridden in subclasses
  - B) They are implicitly final
  - C) They are always visible to subclasses
  - D) They must be implemented in subclasses
  - Answer: B
- What happens if you don't provide a constructor for a class?
  - A) The compiler throws an error
  - B) The class becomes abstract
  - C) The compiler provides a default no-arg constructor
  - D) The class cannot be instantiated
  - Answer: C

Questions 4-6

- What is the output of System.out.println(new Student()); if Student overrides toString()?
  - A) The memory address of the Student object
  - B) The string "Student@<hashcode>"
  - C) The result of the overridden toString() method
  - D) A compilation error
  - Answer: C
- Which statement about the 'final' keyword is true when applied to a method?
  - A) The method cannot be called
  - B) The method cannot be overridden in subclasses
  - C) The method must be static
  - D) The method can only be called once
  - Answer: B
- In Java, can a class extend multiple classes?
  - A) Yes, always
  - B) No, Java doesn't support multiple inheritance of classes
  - C) Yes, but only if all superclasses are abstract
  - D) Yes, but only for inner classes
  - Answer: B

Questions 7-9

- What is the primary motivation for using inheritance in object-oriented programming?
  - A) To create multiple instances of a class
  - B) To keep common behavior in one class and split different behavior into separate classes
  - C) To override all methods in a superclass
  - D) To create private variables
  - Answer: B
- What is the correct order of object construction in inheritance?
  - A) Subclass to superclass
  - B) Superclass to subclass
  - C) Random order
  - D) Depends on the programmer's choice
  - Answer: B
- Which of the following is true about the 'super()' call in a constructor?
  - A) It must be the last line in the constructor
  - B) It must be the first line in the constructor
  - C) It can be placed anywhere in the constructor
  - D) It is optional in all cases
  - Answer: B

Question 11

- String text1 = "Hello World!”;
- String text2 = "Hello World!”;
- boolean b1=text1.equals(text2);
- boolean b2=(text1 == text2);
- For the above program, b1 should be ____, b2 should be ____
- ANS: true, true

Question 12

- String text1 = **new** String("Hello World!");
- String text2 = **new** String("Hello World!");
- boolean b1=text1.equals(text2);
- boolean b2=(text1 == text2);
- For the above program, b1 should be ____, b2 should be ____
- ANS: true, false
-

Question 13

- String text1 = **new** String("Hello World!");
- String text2 = text1;
- boolean b1=text1.equals(text2);
- boolean b2=(text1 == text2);
- For the above program, b1 should be ____, b2 should be ____
- ANS: true, true

Question 14

- String text1 = "Hello World!";
- String text2 = **new** String("Hello World!");
- boolean b1=text1.equals(text2);
- boolean b2=(text1 == text2);
- For the above program, b1 should be ____, b2 should be ____
- ANS: true, false
-

Question 15

- Regex: What is the special character that matches zero or more characters?
  - *
- Regex: What is the special character that matches zero or more characters?
  - +

Question 16

- Which of the following is NOT a way to combine regular expressions?
  - A Repetition
  - B Concatenation
  - C Alternation
  - D Multiplication
- ANS: D

Question 17

- Regex: Which of these will match the strings "revolution", "revolutionary", and "revolutionaries"?
  - revolution[a-z]*
- It will match strings like "revolution", "revolutionary", and "revolutionaries" because the asterisk (*) quantifier allows for zero or more occurrences of the specified character range [a-z]124.
- Here's a brief breakdown of the components:
  - revolution: Matches the exact string "revolution".
  - [a-z]: Matches any single lowercase letter from "a" to "z".
  - *: A quantifier that matches zero or more occurrences of the preceding element, which in this case is any lowercase letter from "a" to "z".

Question 18

- What does the regular expression ^[A-Z][a-z]+$ match?
  - Any capitalized word with at least one lowercase letter
- It is designed to match strings that represent a capitalized word, which begins with a single uppercase letter followed by one or more lowercase letters. Here's a detailed breakdown of the components:
  - ^: Asserts the start of the string.
  - [A-Z]: Matches exactly one uppercase letter from "A" to "Z".
  - [a-z]+: Matches one or more lowercase letters from "a" to "z". The plus sign (+) quantifier indicates that there must be at least one lowercase letter following the uppercase letter.
  - $: Asserts the end of the string.

Question 19

- Which regex matches a string that starts with "Hello"?
  - ^Hello
- The regular expression ^Hello is designed to match any string that starts with the exact sequence "Hello". Here's a breakdown of its components:
- ^: Asserts the start of the string. This means that "Hello" must appear at the very beginning of the string for a match to occur.
- Hello: Matches the exact sequence of characters "Hello".
- This regex pattern will match strings like "Hello world", "Hello123", or simply "Hello", as long as "Hello" is at the start of the string. It will not match strings where "Hello" appears later in the text, such as "Say Hello" or "A big Hello".

Question 20

- Which regex pattern matches a word with exactly three letters?
  - \w\w\w
  - \w{3}
- \w: Matches any word character, which typically includes letters (both uppercase and lowercase), digits, and underscores. If you want to restrict it to only letters, you can replace \w with [a-zA-Z].
- {3}: Specifies that the preceding element (\w or [a-zA-Z]) must occur exactly three times.
- \w\w\w is the same as \w{3}

Question 21

- Which regex pattern matches strings that start with "re" and end with "ed" (like "received" or "renewed")?
  - ^re\w+ed$
- The regular expression ^re\w+ed$ is designed to match strings that start with "re", followed by one or more word characters, and end with "ed". Here's a detailed breakdown of its components:
  - ^: Asserts the start of the string.
  - re: Matches the exact sequence of characters "re" at the beginning of the string.
  - \w+: Matches one or more word characters. A word character is typically any letter (uppercase or lowercase), digit, or underscore.
  - ed: Matches the exact sequence of characters "ed".
  - $: Asserts the end of the string.
- This regex pattern will match words like "replayed", "reminded", or "refunded", as long as they start with "re" and end with "ed", with at least one additional word character in between. It will not match strings that do not conform to this structure, such as "red", "reed", or "ready".

Question 22

- Which regex can be used to match a valid time in 24-hour format (HH:MM)?
  - (\d|1[0-9]|2[0-3]):[0-5]\d
- The regular expression (\d|1[0-9]|2[0-3]):[0-5]\d is designed to match time strings in a 24-hour format (HH:MM). Here's a detailed breakdown of its components:
- (\d|1[0-9]|2[0-3]): Matches the hour part of the time.
  - \d: Matches any single digit from 0 to 9, which would cover hours "0" to "9".
  - 1[0-9]: Matches hours from "10" to "19".
  - 2[0-3]: Matches hours from "20" to "23".
- :: Matches the colon character that separates the hours and minutes.
- [0-5]\d: Matches the minutes part of the time.
  - [0-5]: Matches any digit from 0 to 5, representing the tens place of minutes.
  - \d: Matches any single digit from 0 to 9, representing the units place of minutes.
- This regex pattern effectively captures valid hour and minute combinations in a 24-hour time format, such as "3:15", "12:45", and "23:59". However, it allows for single-digit hours without a leading zero (e.g., "3:15" instead of "03:15"). If you want to ensure that hours are always two digits, you might need a slightly different pattern.

Question 22

- Which regex can be used to match a valid time in 24-hour format (HH:MM)?
  - (\d|1[0-9]|2[0-3]):[0-5]\d
- The four original choices are all wrong, so everyone gets the full point for this question
  - [0-2]\d:[0-5]\d
    - [0-2]\d: This part matches the hour component of the time.
    - [0-2]: Matches any single digit from 0 to 2, representing the tens place of the hour.
    - \d: Matches any single digit from 0 to 9, representing the units place of the hour. Combined with [0-2], this allows for hour values from "00" to "29". However, this pattern is slightly incorrect for a 24-hour clock since it allows hours like "25" to "29", which are not valid.
    - :: Matches the colon character that separates hours from minutes.
    - [0-5]\d: This part matches the minute component of the time.
    - [0-5]: Matches any digit from 0 to 5, representing the tens place of the minutes.
    - \d: Matches any single digit from 0 to 9, representing the units place of the minutes. This ensures that minute values range from "00" to "59".
    - While this regex pattern captures many valid times, it incorrectly allows some invalid hour values (like "25:00").
  - (\d|2[0-3]):[0-5]\d
    - Does not match hours 11,12,13…
    - \d: Matches any single digit from 0 to 9. This allows for single-digit hours like "0" to "9".
    - 2[0-3]: Matches hours from "20" to "23". This ensures that valid two-digit hours in the range of 20 to 23 are captured.
  - \d\d:\d\d
  - [0-9]{2}:[0-9]{2}
    - Both are the same, matches strings like "12:34", "99:99", or "00:00"

Question 23

- Which regex pattern matches a valid IPv4 address?
  - (25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?)(\.(25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?)){3}
- This regex pattern consists of two main parts:
  1. (25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?): This part matches a single octet (0-255) of an IPv4 address.
  2. (\.(25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?)){3}: This part matches the remaining three octets, each preceded by a dot.
- **Matching a Single Octet**
  - The first part (25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?) matches numbers from 0 to 255:
    - 25[0-5]: Matches numbers from 250 to 255
    - 2[0-4][0-9]: Matches numbers from 200 to 249
    - [1]?[0-9][0-9]?: Matches numbers from 0 to 199. ? is a quantifier that specifies that the preceding element (in this case, "1") can appear zero or one time. Essentially, it makes the presence of "1" optional.
- **Matching the Full IP Address**
  - The second part (\.(25[0-5]|2[0-4][0-9]|[1]?[0-9][0-9]?)){3} repeats the octet pattern 3 more times, each preceded by a dot.

Question 23

- Wrong choice: \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
- Or (\d{1,3}\.){3}\d{1,3}
- The regular expression \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3} is designed to match patterns that resemble IPv4 addresses. Here's a breakdown of how it works:
- \d{1,3}: Matches between one and three digits. This pattern is repeated four times, once for each segment of an IPv4 address.
  - \d: Matches any single digit from 0 to 9.
  - {1,3}: Specifies that the preceding element (a digit) must occur at least once and at most three times.
- \.: Matches the literal dot character. The backslash (\) is used to escape the dot because, in regex, a dot normally matches any character except a newline.
- This pattern will match strings that look like IPv4 addresses, such as "192.168.0.1", "10.0.0.255", or "127.0.0.1". However, it does not validate whether each segment is within the valid range for an IPv4 address (0 to 255). For stricter validation that ensures each segment is within this range, a more complex regex would be required.
- (\d{1,3}\.){3}\d{1,3} is the same as \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}

Question 23

- Wrong choice: [0-255]\.[0-255]\.[0-255]\.[0-255]
  - The pattern [0-255] matches any single character that is either '0', '2', '5', or '1'. This does not correctly represent the range of numbers from 0 to 255.

Question 24

- Describe the worst case running time of the following code in “big-Oh” notation in terms of the variable n.

  -
    ```
    void f1(int n) {
    for(int i=0; i < n; i++) {
    for(int j=0; j < n; j++) {
    for(int k=0; k < n; k++) {
    for(int m=0; m < n; m++) {
    System.out.println("!");
    } } } }
    }
    ```
- ANS: $O(n^4)$
- 4 Nested loops, each runs for n, so total iterations is $O(n^4)$

Question 25

- Describe the worst case running time of the following code in “big-Oh” notation in terms of the variable n.

-
  ```
  void f2(int n) {
  for(int i=0; i < n; i++) {
  for(int j=0; j < 10; j++) {
  for(int k=0; k < n; k++) {
  for(int m=0; m < 10; m++) {
  System.out.println("!");
  } } } }
  }
  ```
- ANS: O(n^2)

1. Outer Loop (i loop): executes n times.
2. Second Loop (j loop): Runs from 0 to 9, so it executes 10 times.
3. Third Loop (k loop): executes n times.
4. Innermost Loop (m loop): Runs from 0 to 9, so it executes 10 times.

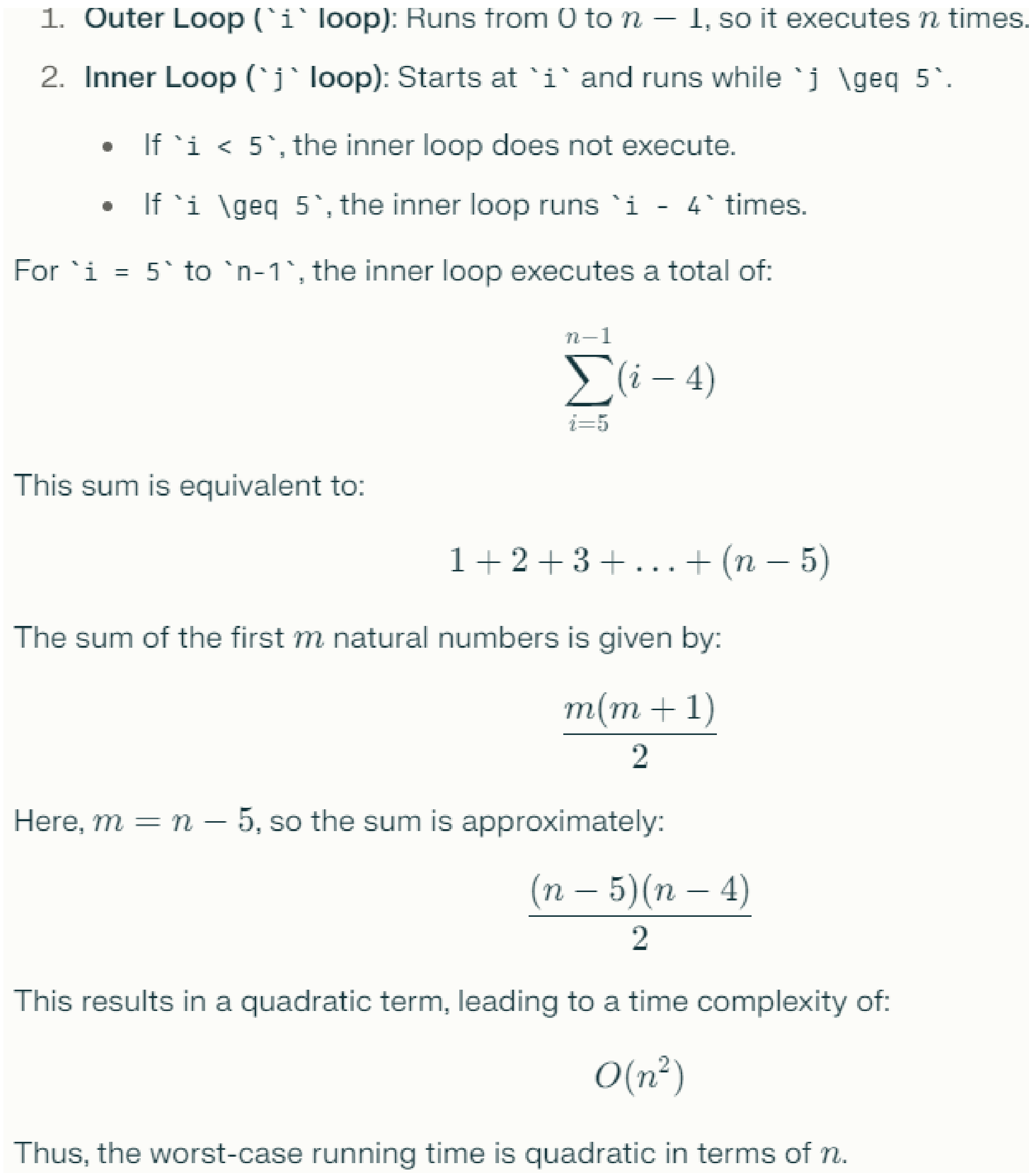
- To find the total number of iterations, we multiply the number of iterations of each loop:Total iterations = $n \times 10 \times n \times 10 = 100n^2$.
- The dominant term in this expression is n^2, and constants are ignored in big-Oh notation. Therefore, the worst-case running time of the function in big-Oh notation is:O(n^2)

Question 26

- Describe the worst case running time of the following code in "big-Oh" notation in terms of the variable n.

-
  ```
  int f3(int n) {
  int sum = 73;
  for(int i=0; i < n; i++) {
  for(int j=i; j >= 5; j--) {
  sum--;
  }}
  return sum;
  }
  ```
- ANS: O(n^2)

1. Detailed explanation on the right
2. Simplified reasoning:
   1. Outer loop executes n times
   2. Inner loop executes O(i) times. Since i goes from 0 to n-1, it is equivalent to O(n) times.
   3. So the overall complexity is O(n^2)



1. **Outer Loop (`i` loop)**: Runs from 0 to $n-1$, so it executes n times.
2. **Inner Loop (`j` loop)**: Starts at `i` and runs while `j \geq 5`.
   - If `i < 5`, the inner loop does not execute.
   - If `i \geq 5`, the inner loop runs `i - 4` times.

For `i = 5` to `n-1`, the inner loop executes a total of:

$$\sum_{i=5}^{n-1}(i-4)$$

This sum is equivalent to:

$$1+2+3+\ldots+(n-5)$$

The sum of the first m natural numbers is given by:

$$\frac{m(m+1)}{2}$$

Here, $m=n-5$, so the sum is approximately:

$$\frac{(n-5)(n-4)}{2}$$

This results in a quadratic term, leading to a time complexity of:

$$O(n^2)$$

Thus, the worst-case running time is quadratic in terms of n.

Question 27

- Describe the worst case running time of the following code in “big-Oh” notation in terms of the variable n.
-
```
int f5(int n) {
if (n < 10) {
System.out.println("!");
return n+3;
} else {
return f5(n-1) + 1;
}
}
```
- ANS: O(*n*)
- The function f5 is a recursive function with the following structure:
- Base Case: If n < 10, the function prints a message and returns n + 3. This operation takes constant time, O(1)*O*(1).
- Recursive Case: If n \geq 10, the function calls itself with n - 1 and adds 1 to the result of this recursive call.

To determine the worst-case running time in big-Oh notation, we need to analyze the recursive calls:

1. Recursion Depth: The recursion continues until n is less than 10. Therefore, if you start with a value of n, the function will make recursive calls until it reaches 9. This means there are n - 9 recursive calls.
2. Time Complexity: Each recursive call involves a constant amount of work (the addition operation and function call overhead), which is O(1)*O*(1). Therefore, the total time complexity is determined by the number of recursive calls, which is n - 9.

- Thus, the worst-case running time of this function in big-Oh notation is: O(n)

Question 28

- Describe the worst case running time of the following code in “big-Oh” notation in terms of the variable n.

-
  ```
  void f4(int n, int sum) {
  int j = n;
  while (j > 2) {
   sum++;
   j = j / 2;
  }
  }
  ```
- ANS: O(log *n*)

  The function f4 has a while loop that continues as long as j > 2, where j is initially set to n. Within each iteration of the loop, j is divided by 2 using integer division. This pattern of repeatedly halving j is characteristic of logarithmic behavior. Here's why:

1. Loop Behavior: The loop halves the value of j in each iteration. This means that the number of times the loop runs is proportional to how many times you can divide n by 2 before it becomes less than or equal to 2.
2. Logarithmic Complexity: The number of times you can divide a number by 2 before it becomes less than or equal to a constant (in this case, 2) is approximately the logarithm base 2 of that number. Thus, the number of iterations of the loop is roughly $\log_2(n)$).

- Therefore, the worst-case running time of the function f4 in big-Oh notation, in terms of the variable n, is:O(logn) (the base of 2 or 10 does not matter)

Question 29

- Describe the worst case running time of the following code in “big-Oh” notation in terms of the variable n.

-
  ```
  int silly(int n, int m) {
   if (m < 2) return m;
   if (n < 1) return n;
   else if (n < 10)
   return silly(n/m, m);
   else
   return silly(n - 1, m);
  }
  ```
- ANS: O(*n*)

1. Base Cases:
   1. If m < 2, the function returns m. This is a constant-time operation, O(1)*O*(1).
   2. If n < 1, the function returns n. This is also a constant-time operation, O(1)*O*(1).
2. Recursive Cases:
   1. If n < 10, the function calls itself with silly(n/m, m). This means that in each recursive call, n is divided by m.
   2. If n >= 10, the function calls itself with silly(n - 1, m). This means that in each recursive call, n is decremented by 1.

- Analysis of Recursive Calls
  - Case when n < 10: The recursion depth depends on how many times you can divide n by m until n becomes less than 10. This is logarithmic with respect to n in base m, leading to a complexity of O($\log_m$n).
  - Case when n >= 10: The recursion depth depends on decrementing n by 1 until it becomes less than 10. This results in a linear number of recursive calls proportional to the initial value of n, leading to a complexity of O(n).
- Worst-Case Complexity
  - O(n) + O($\log_m$n) Since O(n) dominates O($\log_m$n), the complexity is O(n)
  - The worst-case scenario occurs when the function repeatedly decrements n from a large value (when n >= 10). In this case, the time complexity is dominated by the linear decrement operation: O(n)

Question 30

- In the worst case, the number of comparisons needed to search a singly linked list of length n for a given element is
- ANS: n
- In the worst-case scenario, you may have to traverse the entire list to find the element or determine that it is not present. Each node must be checked one by one, leading to a linear number of comparisons relative to the size of the list.
- This question did not ask for worst-case complexity, which is O(n).

Questions 31-32

- What is a collision in a hash table implementation of a symbol table? Check the best definition.
  - ANS: Two key-value pairs that have different keys but hash to the same index.
- Suppose that your hash function does not satisfy the uniform hashing assumption. Which of the following can result?
  - A Poor performance for insert in linear-probing hash table.
  - B Uneven distribution of lengths of chains in separate-chaining hash table.
  - C Large clusters in linear-probing hash table.
  - D Poor performance for search hit or miss.
  - E All of the above.
  - ANS: E

Question 33

- An advantage of Closed Addressing: Separate Chaining over the open addressing scheme is
- Worst case complexity of search operations is less
- Space used is less
- Correct: Deletion is simpler and easier
- None of the above

- An advantage of Closed Addressing using Separate Chaining over the Open Addressing scheme is that **deletion is simpler and easier**. Here is why:
- 1. **Simpler Deletion**: In separate chaining, each bucket in the hash table contains a linked list (or another data structure like a dynamic array or binary search tree) to handle collisions. This makes deletion straightforward because you can directly remove an element from the list without needing to manage special markers or rehash other elements, as is often required in open addressing.
- The other options mentioned are not advantages of separate chaining over open addressing:
- - **Worst Case Complexity of Search Operations**: The worst-case complexity for search operations in separate chaining can be O(n) if all elements hash to the same bucket, forming a long chain[2]. In contrast, open addressing has a worst-case complexity that can also degrade significantly with high load factors, but it doesn't inherently provide better worst-case performance than separate chaining.
- - **Space Used is Less**: Separate chaining typically uses more space than open addressing because it requires additional storage for pointers or links in the linked lists or other structures used to store multiple items per bucket. Open addressing stores all entries directly in the hash table array, which can be more space-efficient.

Question 34

- Hashing: Given a hash table T with 25 slots that stores 2000 elements, the load factor α for T is __________
- ANS: 80
- The load factor α for a hash table is calculated as the ratio of the number of elements stored in the table to the number of slots available in the table. Given a hash table with 25 slots that stores 2000 elements, the load factor is calculated as 2000/25= 80. This indicates that, on average, each slot in the hash table contains 80 elements.

$$\text{Load factor} = \frac{\text{Total number of items stored}}{\text{Size of the array}}$$

Question 35

- Hashing: Consider a hash table of size seven, with starting index zero, and a hash function (7x+3) mod 4. Assuming the hash table is initially empty, which of the following is the contents of the table when the sequence 1, 3, 8, 10 is inserted into the table using closed hashing ? Here "__" denotes an empty location in the table.
- ANS: [3, 10, 1, 8, '__', '__', '__']

1. Calculate Initial Hash Indices:
   1. For 1: (7×1+3)mod 4=10mod 4=2(7×1+3)mod4=10mod4=2
   2. For 3: (7×3+3)mod 4=24mod 4=0(7×3+3)mod4=24mod4=0
   3. For 8: (7×8+3)mod 4=59mod 4=3(7×8+3)mod4=59mod4=3
   4. For 10: (7×10+3)mod 4=73mod 4=1(7×10+3)mod4=73mod4=1
2. Insert into Hash Table:
   1. Insert 1 at index 2.
   2. Insert 3 at index 0.
   3. Insert 8 at index 3.
   4. Insert 10 at index 1.

- Since there are no collisions with these initial indices, each element is placed directly into its calculated position.Thus, the final contents of the hash table are:text
- [3, 10, 1, 8, '__', '__', '__']

Linear Probing: Primary Clustering

What is the probability of next key going in each slot?

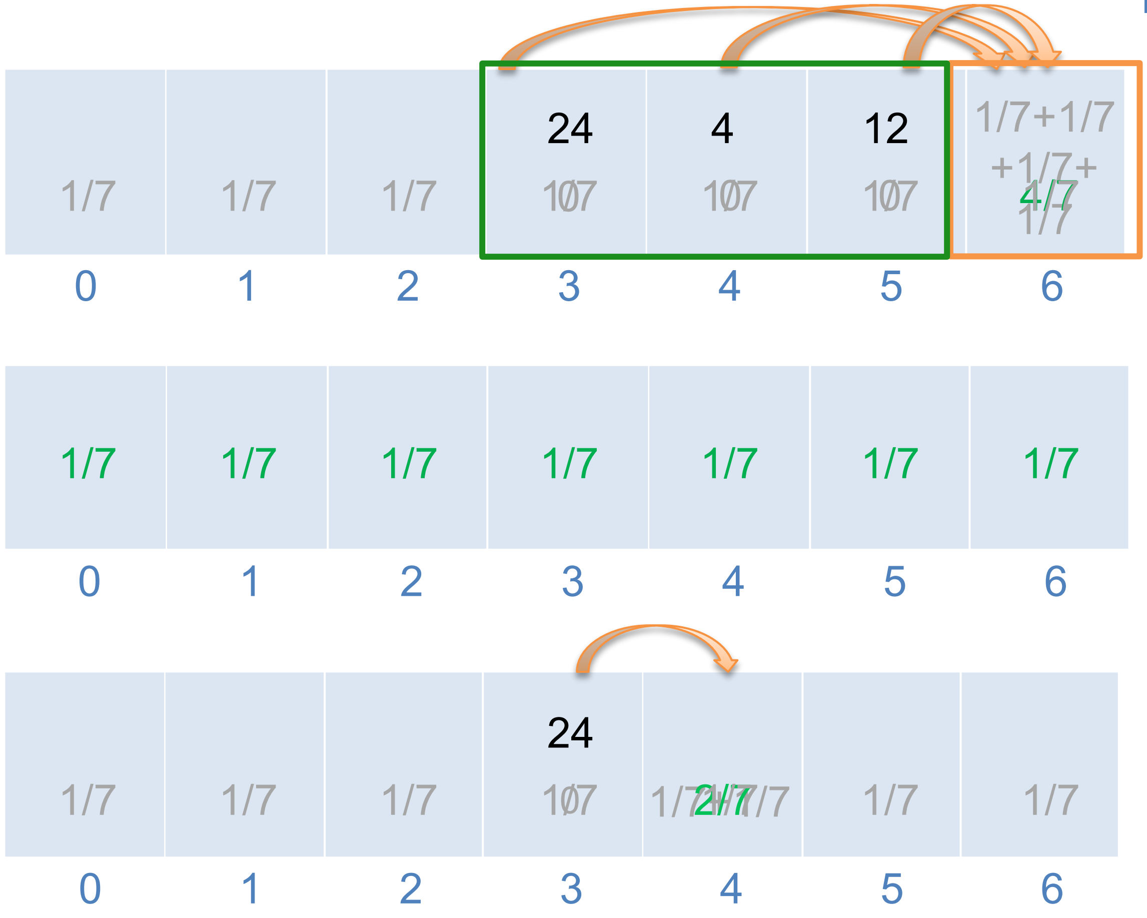
Hash(k) = k mod 7

All keys equally likely

Cluster is a contiguous block of items.

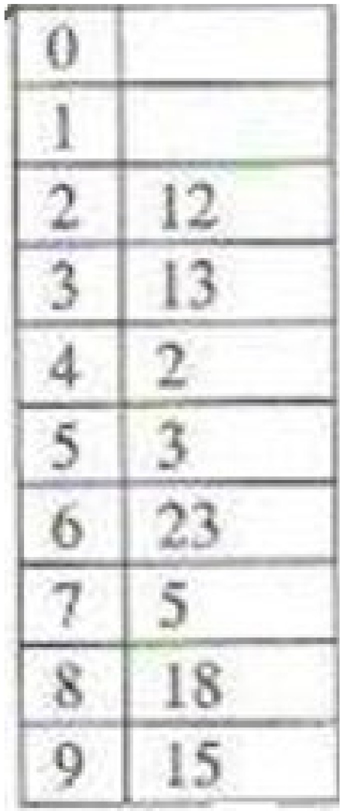
Observation. New keys likely to hash into middle of big clusters.

Higher insert and search costs - O(n)



Question 36

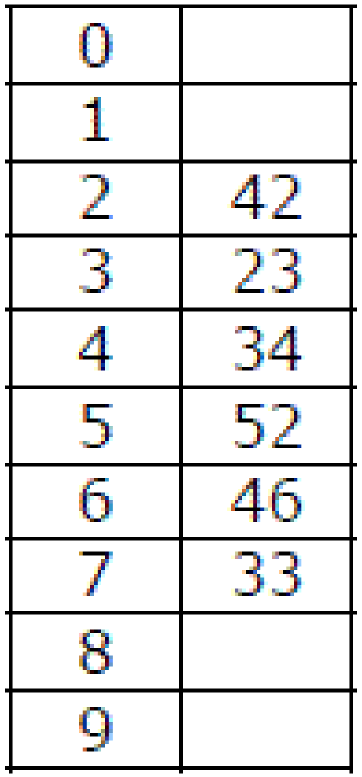
- Hashing: The keys 12, 18, 13, 2, 3, 23, 5 and 15 are inserted into an initially empty hash table of length 10 using open addressing with hash function h(k) = k mod 10 and linear probing. What is the resultant hash table?
- ANS: See notes for details



0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

Question 37

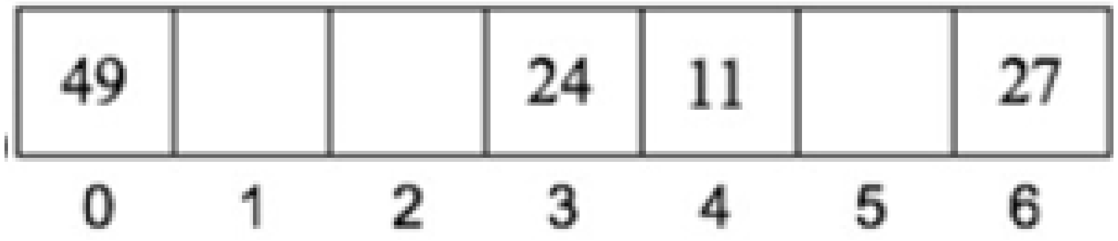
- Hashing: A hash table of length 10 uses open addressing with hash function h(k)=k mod 10, an linear probing. After inserting 6 values into an empty hash table, the table is as shown below.
- Which one of the following choices gives a possible order in which the key values could hav been inserted in the table?
- ANS: 46, 34, 42, 23, 52, 33
- 46, 34, 42, 23: no collision
- 52: collision, placed in 5
- 33: collision, placed in 7



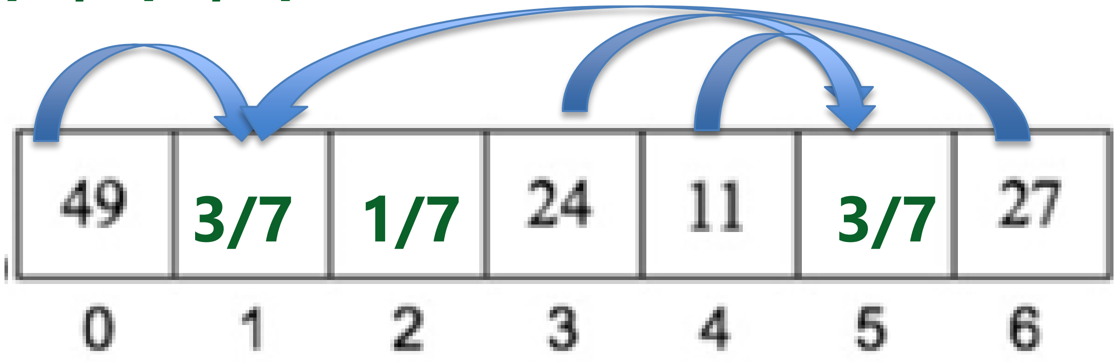
0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Question 38

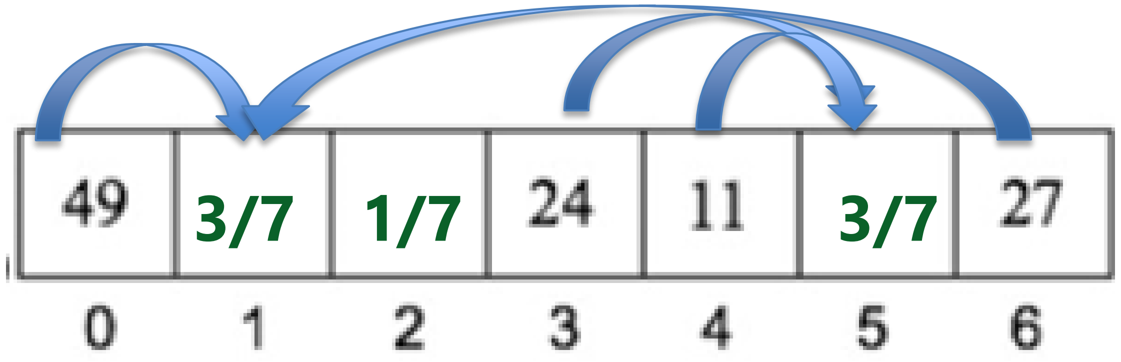
- What is the probability of next key going in the open slots in the following hash able? Assume each table index is equally likely for each key. Hash(k) = k mod 7



- ANS: **3/7, 1/7, 3/7**



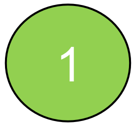
Question 38 Explanations



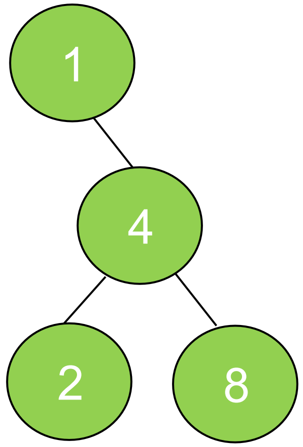
- Probability of placing into position 1 = prob(hashing into 6) + prob(hashing into 0) + prob(hashing into 1) = 1/7+1/7+1/7=3/7
- Probability of placing into position 2 = prob(hashing into 2) = 1/7
- Probability of placing into position 5 = prob(hashing into 3) + prob(hashing into 4) + prob(hashing into 5) = 1/7+1/7+1/7=3/7

Height of a Tree

- The height of a tree is defined as the number of edges in the longest path from the root node to a leaf node.
- For a tree with only a root node, the height is 0.

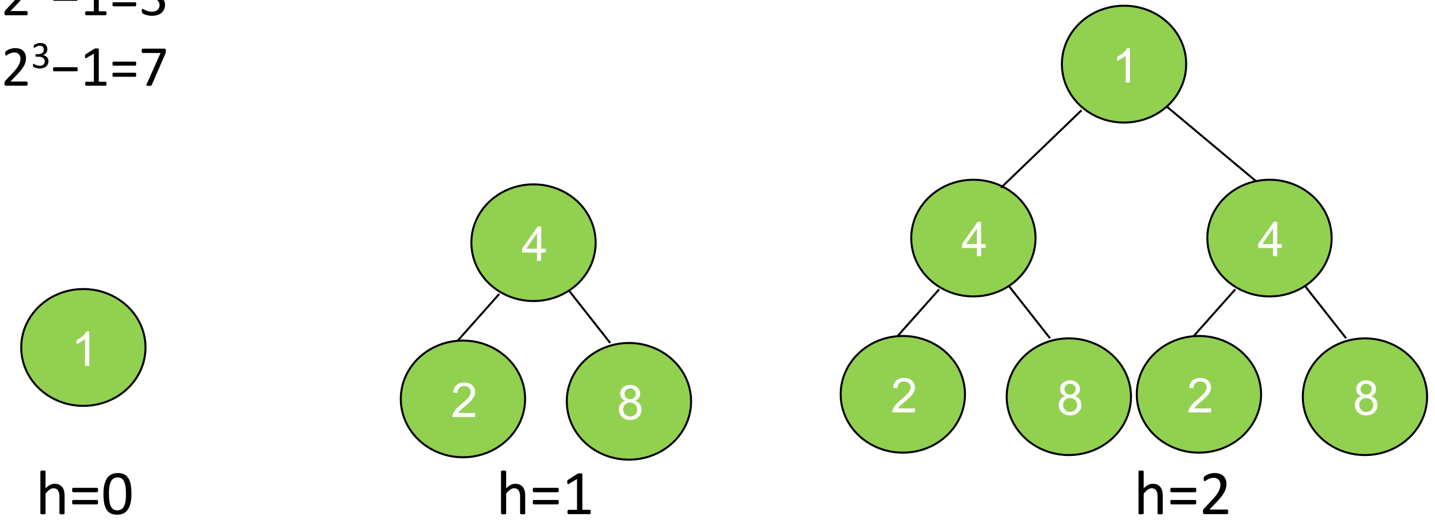


- For the tree below, the height is 2.



Full Binary Tree

- A full binary tree with height h has a total number of nodes given by the formula: $n = 2^{h+1}-1$
- This formula arises because, in a full binary tree, each level is completely filled. The number of nodes at each level l is 2^l. Therefore, the total number of nodes is the sum of nodes at all levels from 0 to h, which is a geometric series: $n=1+2+4+\ldots+2^h=2^{h+1}-1$
- This means that for a full binary tree, the total number of nodes grows exponentially with the height of the tree
- $h=0$: $n=2^1-1=1$
- $h=1$: $n=2^2-1=3$
- $h=2$: $n=2^3-1=7$

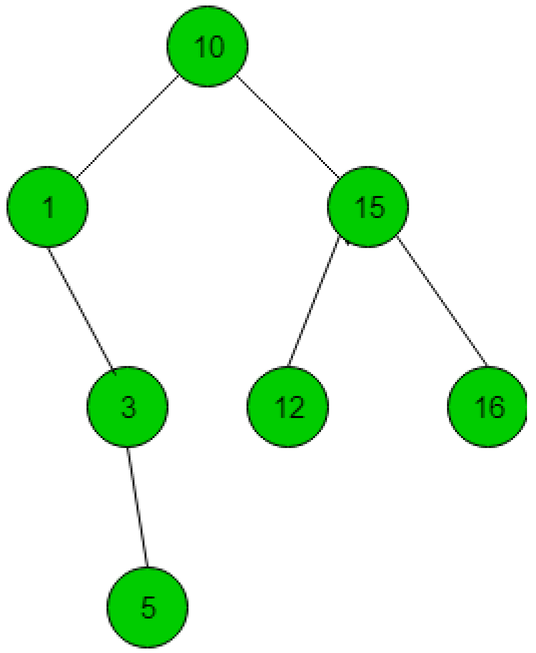


Height of a Binary Tree

- For a binary tree with n nodes, the height h is bounded by: $\lceil \log_2(n+1) \rceil - 1 \leq h \leq n - 1$
  - $\lceil\rceil$ is the ceiling operator. The lower bound represents a perfectly balanced tree, and the upper bound represents a degenerate tree (essentially a linked list).
  - The height of a tree is equivalent to the maximum depth of any node in the tree.
  - For a binary search tree, the minimum height with n nodes is $\lceil \log_2(n+1) \rceil - 1$, which occurs in the most balanced configuration, where $\lceil\rceil$ is the ceiling operator, e.g., $\lceil 1.0 \rceil=1$, $\lceil 1.3 \rceil=2$
  - The maximum height of a binary tree with n nodes is n-1, which occurs in the case of a skewed tree (essentially a linked list)

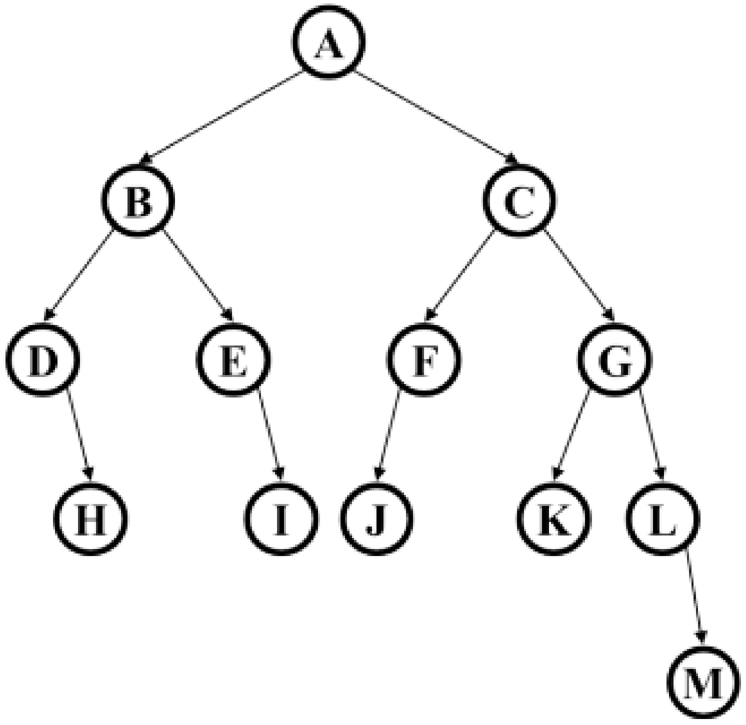
Question 39

- The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root, i.e. a tree with a single root node has height 0.)?
- ANS: 3



Question 40

- Assume this tree is a binary search tree. What is the maximum number of nodes that could be added to the tree without increasing its height?
- ANS: 18
- A full binary tree with height h has a total number of nodes given by the formula: $n = 2^{h+1}-1 = 2^{4+1}-1=31$, since the height is 4.
- Currently there are 13 nodes, so you can add 31-13=18 nodes without increasing its height.
- You can also count the inserted nodes until the tree is full.

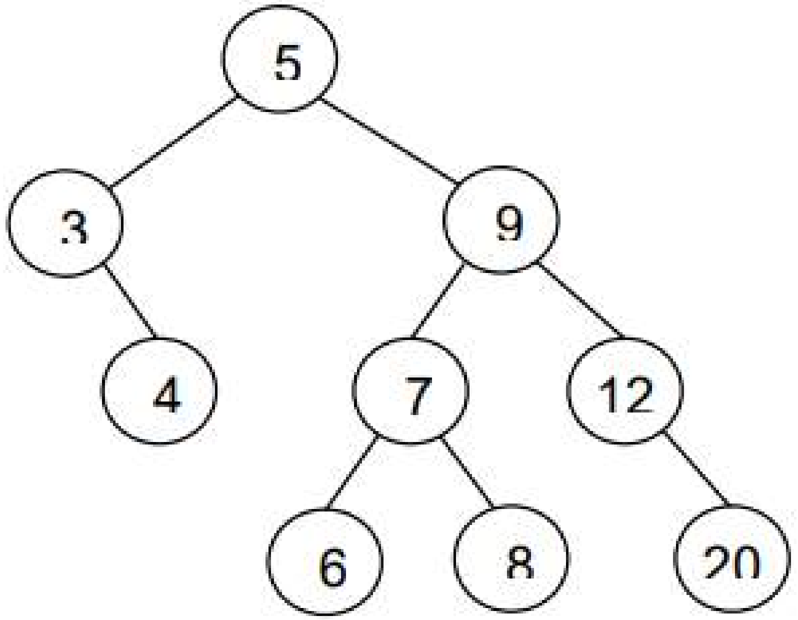


Question 41

- ____traversal always visits the tree root node first, and ____traversal always visits the tree root node last.
- ANS: _ Pre-order _traversal always visits the tree root node first, and _ post-order_traversal always visits the tree root node last.

Questions 42 43 44 45

- Preorder [5, 3, 4, 9, 7, 6, 8, 12, 20]
- In-order: [3, 4, 5, 6, 7, 8, 9, 12, 20]
- Post-order: [4, 3, 6, 8, 7, 20, 12, 9, 5]
  - The standard answers are wrong, so I manually re-graded Questions 42 43 44
- Level-order traversal by BFS: [5, 3, 9, 4, 7, 12, 6, 8, 20]



Question 46

- Given the post-order traversal SWTQXUVRP of a binary tree, what is the root node?
- post-order_traversal always visits the tree root node last, which is P

g

Question 47

- The in-order and pre-order traversal of a binary tree are: dbeafcg and abdecfg, respectively. The post order traversal of a binary tree is:
- ANS: debfgca

Pre-order: a b d e c f g

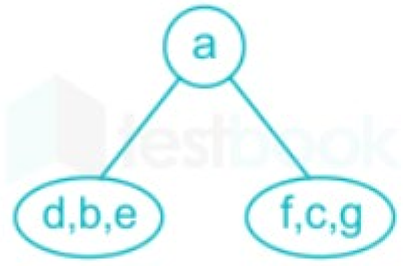
Therefore a is the root:

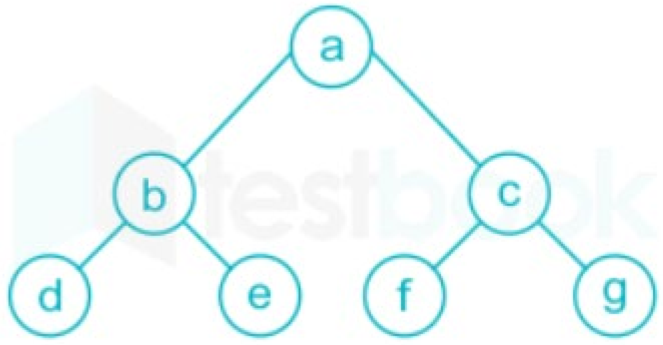
Inorder: **d b e** a **f c g**

Left of a : **d b e**

Right of a: **f c g**

Diagram:





postorder traversal : d e b f g c a

Question 48

- Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. What is the in-order traversal sequence of the resultant tree?
- ANS: 0 1 2 3 4 5 6 7 8 9
- In-order traversal of a binary search tree visits the nodes in ascending order of their values. You do not need to construct the tree.