PG4200

Candidate: 1061

Kristiania

PG4200 exam 2023

5/15/2023

LO1: Understanding Data Structures

1. Question 1

1.1. Programs use variable names to represent data, and the compiler will translate names into addresses in memory. When using an array, it has to calculate size during compilation time to store in HEAP, which is the "free pool" memory segment. Thus, data structure of an array is allocated together in a contiguous way. In Java, the Base Address is given by the JVM and the size of a data type is determined by its type. We can calculate their physical address using formula **memoryAddress = Base Address + (Index of data * data size)**

I.e. Given an Array = [1300……1900] - Given Base Address is 1022 - Given data size is 2 bytes. We want to find X = 1704. Using given formula:

X = 1022 + (1704 * 2) = 1022 + (3408) = **4430**

This means the memory address of index 1704 in the Array of range 1300 - 1900 is 4430.

1.2. Problem: Find the largest unit in an array. A step-by-step algorithm is as the following:

1. Declare a variable to track the largest given unit.
2. Having a loop that will traverse the entire array.
3. For each iteration, compare the current unit with the current largest.
4. If current unit is greater than current largest, update current to the new largest unit.
5. Proceed until we hit the end of the array.
6. No more elements left, we have found the largest unit.

LO1: Stack push, pop, get operations

2. Question 2

2.1. Stack follows a LIFO last in first out principle. In a strict stack, this means we can't reach the first item without popping out the last. This can be visualized with a tennis ball container. A container has a maximum capacity to push tennis balls into the available space. A container can be full, contains tennis balls, or empty. To reach the bottom ball - the first ball we pushed, we have to pop out the top most ball, until we reached the desired ball.

Step-by-step algorithm for push

1. Declare and initialize an array with a maximum capacity as the container.
2. Declare a variable name as top to track the current capacity, initialize it to -1 as empty.
3. Before we push a tennis ball, check the container if it's full, as we can't push when the container is already full. This will cause overflow otherwise.
4. If it's not full, we push the ball into the container as the top most ball.
5. Update the size, set the previous size and add current size with 1 each time we push a new ball. I.e. top = top + 1

Step-by-step algorithm for pop

1. Before popping a unit, we have to check the container if it's empty, as there is nothing to pop when the container is empty. This will cause underflow otherwise.
2. Before popping a unit, we have to check the container if it's empty, as there is nothing to pop when the container is empty. This will cause underflow otherwise.
3. If it has items, the popped item will be the top most unit we have the access to.
4. Update the size, set the previous size and minus current size with 1 each time we pop. I.e. top = top - 1.

2.2. Implementing a Stack data structure. The Stack shall consist of three methods, push, pop and get minimum. The constraint of values is in the range of 1 - 100. The number of elements is arbitrary chosen. But I want to emphasize "number of queries" could have more context as I don't really understand what you mean, so I excluded it. My argumentation is, you set the constraint on range 1 - 100. I am using a loop to initialize random data into my stack, why not? So long I am within the constraint I don't see any reason why I should put an extra layer of constraint on top my operations, nevertheless the conditions are exact the same.
But having said that, getting the minimum value from the stack need to be in constant time, one way to implement this is to track my lowest value in the stack.

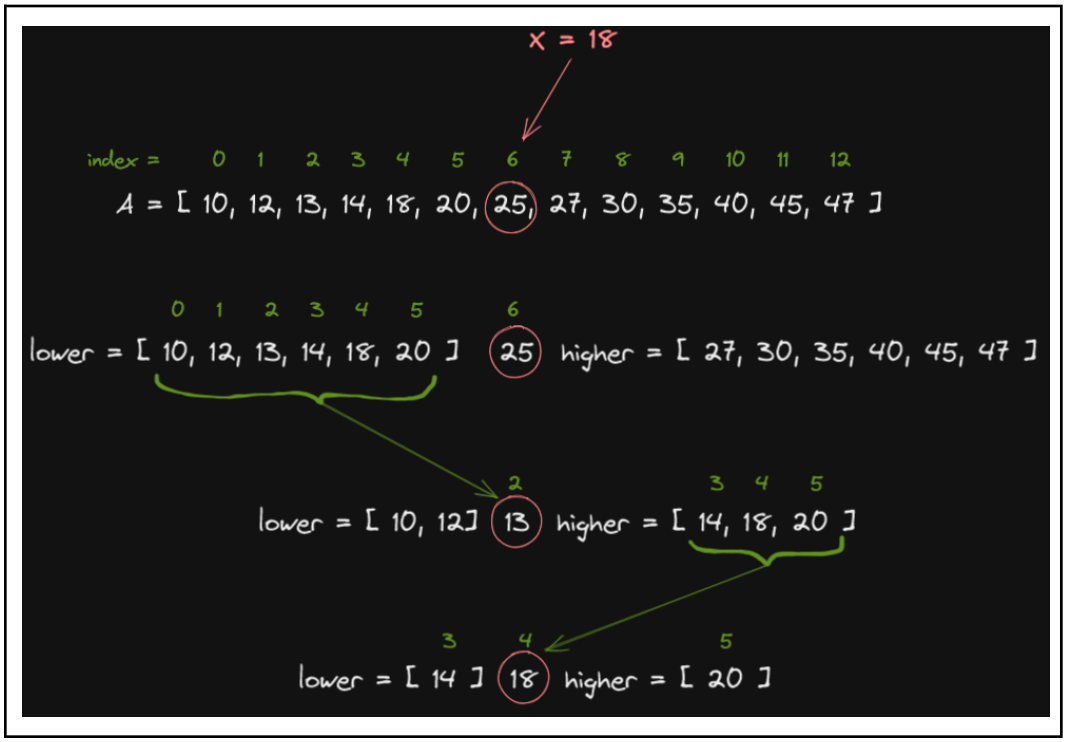
I'll be using another stack that will track the first unit in the current stack. It will keep pushing into the minimum stack so long the current unit is lower than the previous. Why using another stack and not just a field like data type int to update the state? Because stack implementation is more intuitive. If I pop my current stack, it will also update synchronously and in constant time. I think the evaluation should be fair, even if I didn't implement "queries" as I have expressed the clarity in my code and my interpretation.

LO2: Searching Algorithms

3. Question 3

3.1. Binary Search Algorithm works on an already sorted array. This means the array has to be sorted from lowest to highest, quoting "non-descending" order. It has too many terms, but non-descending per se is in ascending order. Strictly speaking, I think adding no-duplicates in front of many terms of sorting order will make it clear. Anyhow, I will denote it as "non-descending" order per request as the correct term. Yes, no duplicates i.e. 1,2,3,4.

Binary Search, short "BS", use a divide and conquer approach to compare X with the middle unit in a given array. If it is a match, we found the target position, otherwise split the array into two subarrays denoted as lower and higher bound. Lower contains units that are less than the middle unit, higher bound is respectively higher than middle.



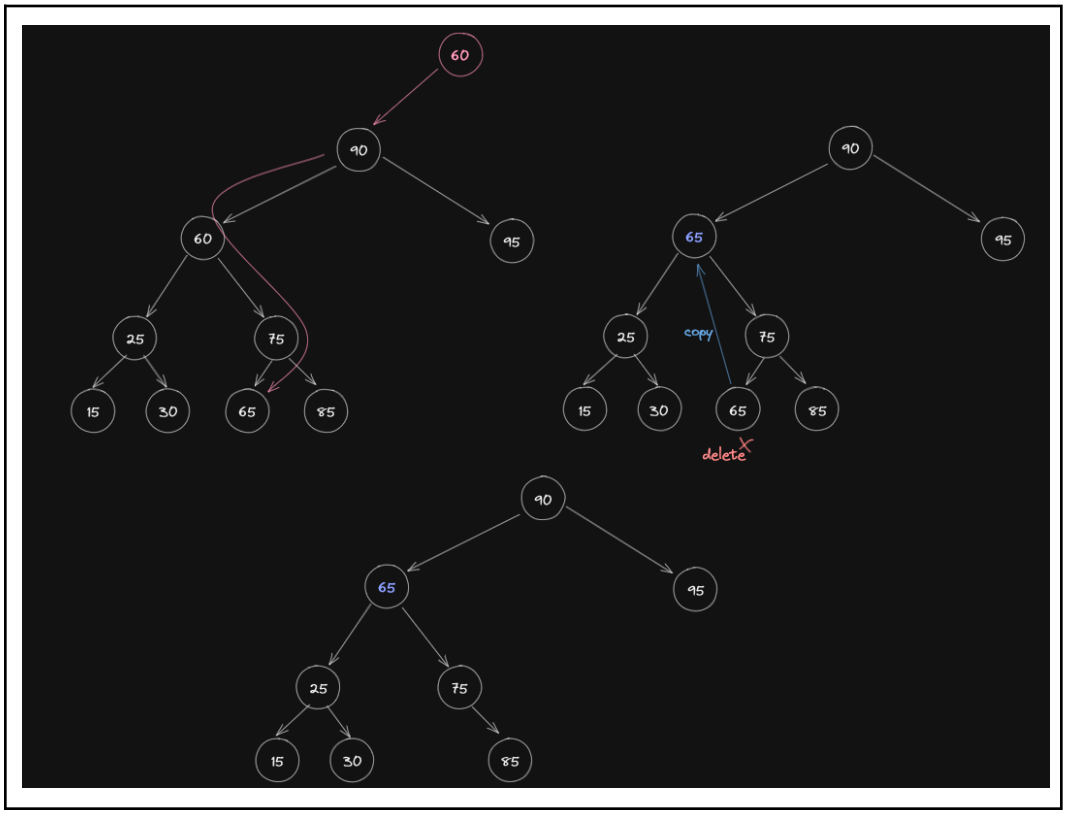
Step-by-step solution.

1. Search term X = 18.
2. Divide by finding the middle index using the formula (lower + higher) / 2. I.e. mid = (0 + 12) / 2 = 6.
3. Conquer by comparing X with the middle value at position 6.
4. 18 is less than 25, it may be on the lower bound.
5. Divide again in the lower, mid = (0 + 5) / 2 = ⌊2.5⌋ = 2.

   We are excluding middle because we already compared it. Floor or Ceiling are both functional, I am sticking with floor.
6. Conquer again by comparing X with new middle.
7. 18 is greater than 13, it may be on the higher bound.

8. Divide again in the higher, mid = (3 + 5) / 2 = 4.
9. Conquer again by comparing X with the new middle.
10. 18 equals to value at position 4, found the term.

3.2. Algorithm for deleting a node in a BST. We have 3 edge cases when we are deleting a node. If the deletion node is a leaf node, remove it. If it has one child, copy the child and replace the deletion node. However, if it has two children, we will find the successor from the deletion node's right subtree using in-order traversal, Left-Root-Right.



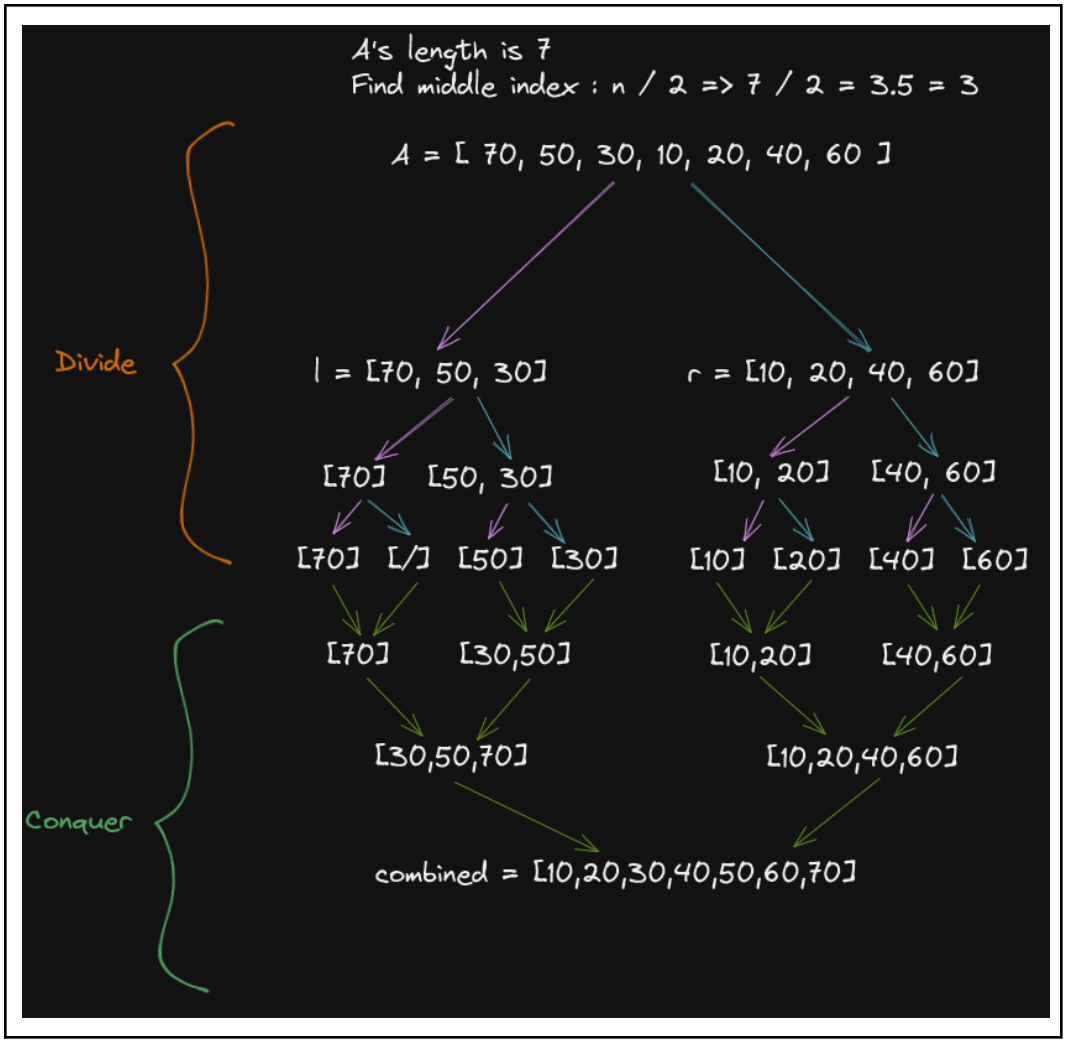
1. We need to traverse our tree and search for the term. If the search term is less, we continue to traverse the left side else the term is greater, we traverse the right side. Given X = 60.
2. Starting from the root, compare 60 with current node 90
3. 60 is less than 90, go left.
4. We found the term X equals to current node 60.
5. Check if current node has children. Since the current node has two children, we will traverse on current's right subtree. Because right subtree contains nodes greater than current. We need to keep BST as BST after deletion.
6. Standing on subtree's root 75, we use in-order traversal Left-Root-Right we find the smallest value as the successor. 65-75-85, 65 is the smallest in order.
7. Assign 65 as the new successor, then remove the deletion node.
8. In-order after deletion: 15-25-30-65-75-85-90-95

LO3: Sorting Algorithms

4. Question 4

4.1. Merge Sort is a sorting algorithm that use divide and conquer approach to sort data. We calculate the middle index using floor division, ceiling is also functional just a preference, but we need to be consistent. Then we split the array into two halves from the middle index. We repeat this split process until we have one unit left. Now we compare all units in the left and right part. For every comparison,

the lowest unit will be appended into a new list. If either of the left or right list becomes empty during comparison, return all units as is into the new list as sorted combined list.



Merge sort divide

1. Find mid = 7 / 2 = 3.5 = 3
2. Split the list in two halves from mid, left index from 0 - 2, right index from 3 - 6.

3. Left contains 70,50,30 - find mid = (0 + 2) / 2 = 1, split in two halves from mid, left index from 0 [70], right index from 1-2 [50,30]
4. Right contains 10,20,40,60 find mid = (3+6) / 2 = 4.5 = 4, split in two halves from mid, left index from 3-4 [10, 20], right index from 5-6 [40,60]
5. Left index 0 [70] is alone done.
6. Right index 1-2 [50, 30] split in two halves from mid = (1+2) / 2 = 1.5 = 1, left index from 1 [50], right index from 2 [30].
7. Left index 1 [50] is alone done, right index 2 [30] is also alone done
8. Repeat the split process for the index from 3-4 and respectively index from 5-6.

Comparison table for each pass, sort it and returned the combined array

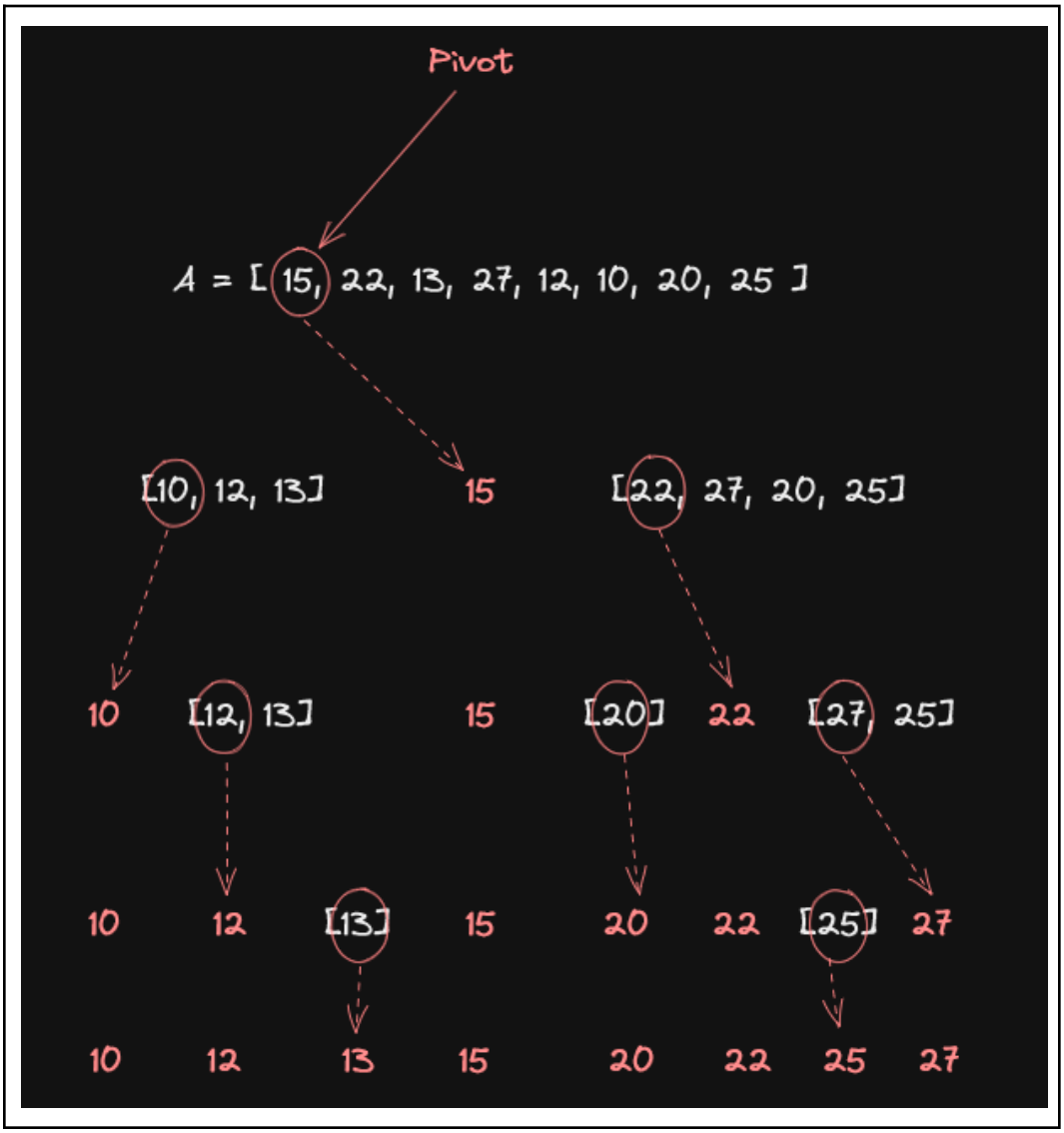
K	L	R	return
1	70	/	70
2	30	50	30, 50
3	10	20	10, 20
4	40	60	40, 60
5	70	30, 50	30, 50, 70
6	10, 20	40, 60	10, 20, 40, 60
7	30,50,70	10,20,40,60	10,20,30,40,50,60,70

Merge sort conquer

1. Left 70 compare with right, right is null, return 70 as sorted combined list.
2. Compare 30 with 50, 30 is less than 50, insert 30 first, then 50 into combined list.
3. 40 is less than 60, insert 40 first, then 60 into combined list.

4. Compare 70 with 30, 70 is greater than 30, insert 30. Compare 70 with 50, 70 is greater than 50, then insert 50. Right becomes empty, insert 70 back to sorted combined list.
5. Compare 10 with 40, 10 < 40, insert 10. Compare 20 with 40, 20 < 40, insert 20. Left becomes empty, insert back 40, 60
6. Compare 30 with 10, 30 > 10, insert 10. Compare 30 with 20, 30 > 20, insert 20. Compare 30 with 40, 30 < 40, insert 30. Compare 50 with 40, 50 > 40, insert 40. Compare 70 with 60, 70 > 60, insert 60. Insert back 70 as the sorted combined list.
7. Check if list is sorted, 10, 20, 30, 40, 50, 60, 70.
8. Otherwise, repeat divide and conquer again.

4.2. Quick Sort is a sorting algorithm that works on divide and conquer approach. Unlike the merge sort, we need to select a pivot as a reference point. Then we want to partition all units less than pivot on the left and higher on the right side, So we know the pivot will be at the sorted position. There are tactics from making a pivot selection, that will further improve the efficiency, like random index or median of three selection. I will not dive into how it works because it was not taught at this course, as I can’t justify my statements.



1. We select the first unit as the pivot for simplicity.
2. We will partition the list into two parts based on the pivot
3. All units less than pivot comes before the pivot, and all units that are greater goes after the pivot.
4. 10, 12, 13 are less than 15, they come before. 22, 27, 20, 25 is greater than 15, they come after. We know pivot 15 is in the sorted position.

5. Select 10 as the new pivot, 12, 13 is greater than 10, so they come after 10. 10 is in sorted position.
6. Select 22 as the new pivot. 20 is less than 22, it comes before. 27 and 25 is greater they come after. 20 is alone and in the sorted position, 22 is also in the sorted position.
7. Select 27, 25 is less than 27, it comes before 27. 25 and 27 is now alone, they are now in the sorted position.
8. List is now sorted 10, 12, 13, 15, 20, 22, 25, 27.

LO4: Traversing Graphs Algorithms

5. Question 5

5.1. The calculation of the Fibonacci sequence is the sum of the first number and second number. I.e. 0 + 0 = 0, 0 + 1 = 1 etc.

For the given nth term from 1 to 5 using recursion will make a total of 15 calls to itself due to the nature of recursion breaking down of a problem into subset of problems, until a sentinel condition is reached. Here the sentinel value is given as "n >= 1"

While the iterative method is straight forward looping so long a condition is true. By using pointers previous and current we can calculate the next nth term, and update the pointers accordingly. While the recursion is less efficient in terms of space efficiency, it may be more intuitive to implement and more readable. However, when working with a larger data set, the call stack might build up too much, and eventually you will expect a stack overflow in the long run. For

smaller dataset, dynamic programming like recursion is perfectly fine.

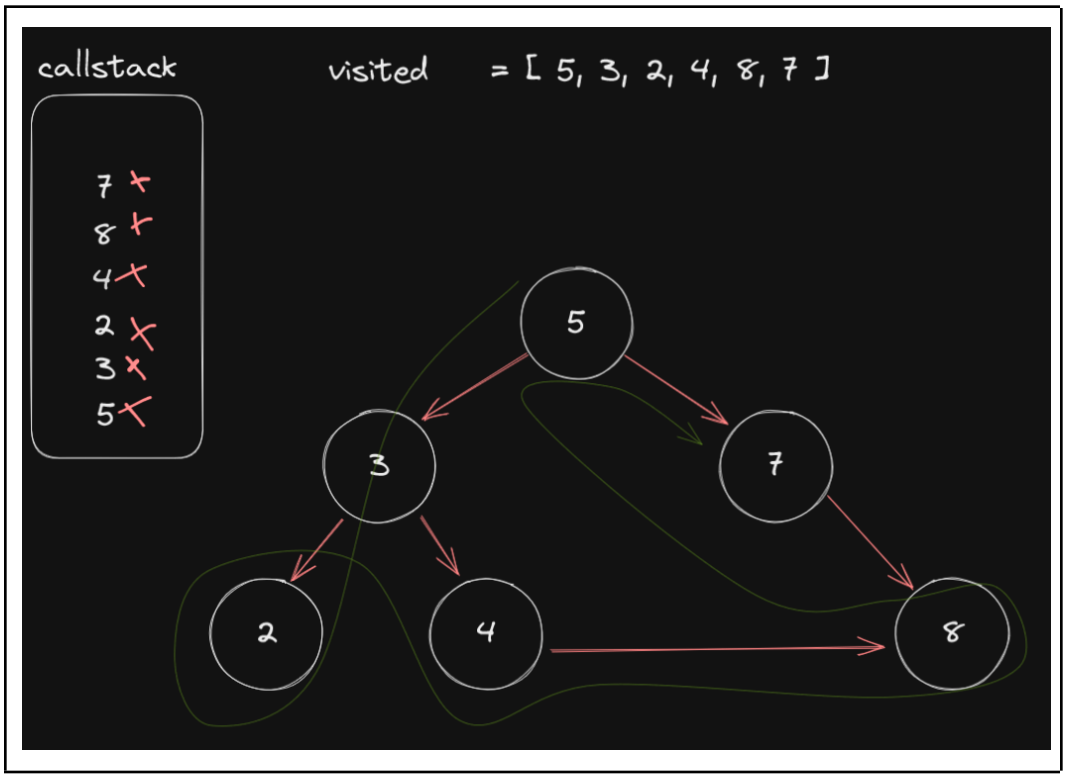
On the other hand, iterative method is a different approach to solve the problem, it eliminates the stack overflow issues and increased memory usage. When it comes to choosing a method, it depends on the specific usage or implementation, which one is more optimal as a whole picture. While recursion is an elegant way of solving some problems and iteration is a jack-of-all trades, whereas both comes with trade-offs.

5.2. Some analysis on the given graph. The graph we can inspect that it is a digraph, meaning it has connecting edge to vertices, but in a directional way. I.e. 3 has an edge to 2, but not vice versa. The current graph also happens to be a binary tree. A graph is also a tree, but not all graphs are binary trees.

Traversing in a graph using Depth-First-Search, DFS is a recursive algorithm based on the principle of tracking current node as visited and explore the connecting vertices. For each new node we stand on, we mark it as visited to avoid visiting it again. We will use two stacks as call and visited. When I mean call stack on, e.g. Node 5, it means the current Node I am standing on. When I pop the call stack, it is the top most Node - current active stack. When I mean mark as visited, it means push the current node into visited stack.

Traversing in a graph using Breadth-First-Search, BFS is an iterative algorithm based on the principle of visiting a node, then explore all adjacent vertices that has direct edge to current, before exploring next level. We can keep track of

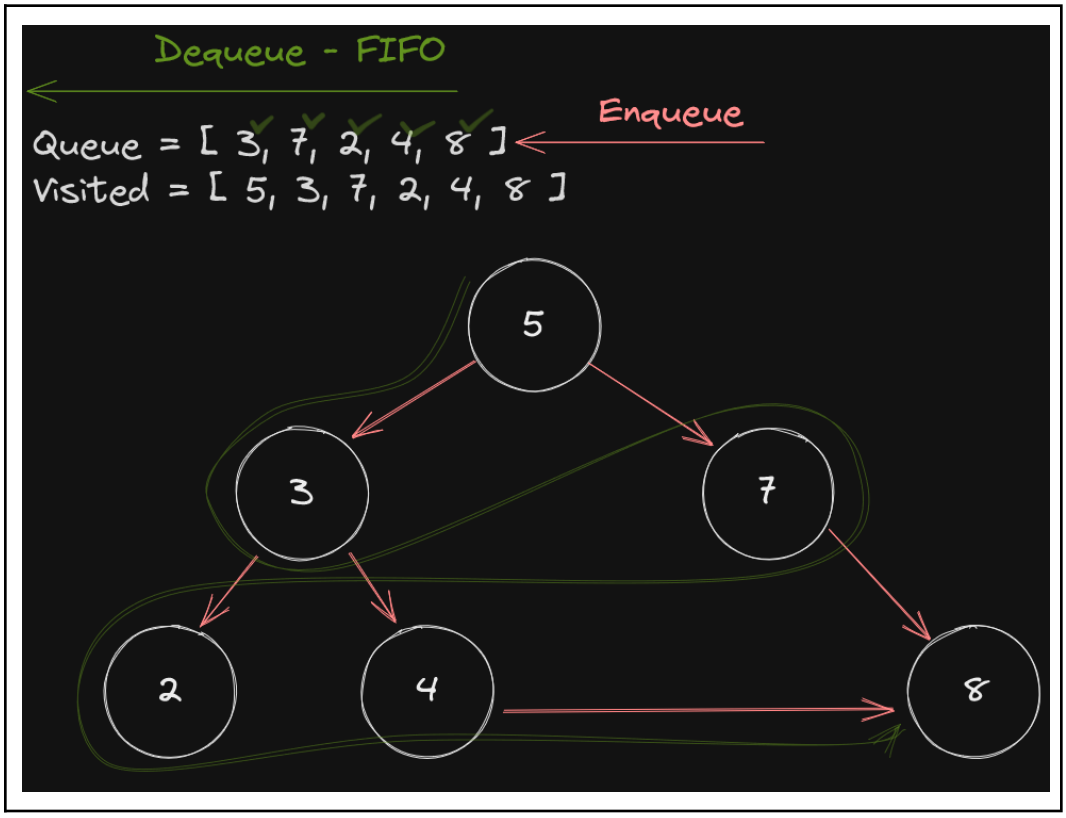
whom to visit next by dequeuing from our queue.



1. You can start with any node, but since 5 has edges to all vertices, I think it will be a great place to start. Calling stack on vertex 5. Mark 5 as visited.
2. Explore 5, 5 has adjacency to 3 and 7. Which one first? Anyone you like, I call stack on 3. Mark 3 as visited.
3. 3 has adjacency to 2 and 4. Call stack on 2, mark 2 as visited.
4. 2 has no edge, pop active stack(2). Current active stack is now 3. Call stack on 4, mark 4 as visited.
5. Vertex 4 has an edge to 8, call stack on 8. 8 has no edge, pop the stack(8). Current active stack is 4, 4 has nothing left, pop active stack(4), current active stack is 3.

Both 2 and 4 are already visited, pop active stack(3). Current active stack is 5, call stack on 7.

6. Vertex 7 has no more edge, as 8 is already marked as visited. Pop active stack(7)
7. Current active stack is 5. All vertices are visited, pop active stack(5).



1. We will start by visiting vertex 5, as it covers all paths to vertices in the graph.
2. Visit 5, 5 has adjacency to 3 and 7, enqueue 3 and 7. Mark 5 as visited.
3. In our queue, we have now vertices in line waiting to be visited. We dequeue 3.
4. Visit 3, 3 has adjacency to 2 and 4, enqueue 2 and 4. Mark 3 as visited.
5. Dequeue 7, 7 has adjacency to 8, enqueue 8. Mark 7 as visited.

6. Dequeue 2, 2 has no edge, mark 2 as visited.
7. Dequeue 4, 8 is already in the queue. Mark 4 as visited.
8. Dequeue 8, 8 has no edge, mark 8 as visited.
9. Queue is now empty, we have traversed one possible path using BFS order.

LO5 and LO6: Computability and Complexity

6. Question 6

6.1. In Computer Science, Complexity class is used to define a group of problem based on time and space. For instance, we can classify a group of problems based on a sorting or searching algorithm that requires a computational task. With the given task we can further classify them based on their efficiency, how fast or slow one algorithm is, or how much space it requires to compute a specific task.
In relation to data structures, different data structures are built different, and we can compare their difference using complexity classes.

6.2. We use Big O Notation as “Ordnung”, order of approximation, by Paul Bachman. This is a mathematical notation that describes the “limiting behavior” of a function’s efficiency. For instance, we can use Big O notation to describe the growth rate of time in upper bound. E.g. for different data structures like array or linked list has different time efficiency when it comes to prepend. We know that a linked list has a pointer to the first node, so prepending will happen in a constant operation. While array are stored in a contiguous way so prepending will cause

the positional index to shift where the time is determined by the number of operations it has to shift, as it will be a linear growth.

6.3. P class is a Polynomial Time class, that consist of problems we can solve using existing algorithms. These are algorithms that are deterministic, meaning the time and space output is determined by its input. For instance, we have a problem to sort a data in a non-descending order. We have a bubble sort that has a sorting growth Big O(n^2) in the upper bound, meaning we are interested in the worst case scenario. “N” is the number of elements in a given data being sorted and ^2 is the quadratic relation to “n”, because for “n” unit, we have to perform “n” time. In contrast to bubble sort, we have more efficient sorting algorithm like Merge Sort that takes divide and conquer approach whereas the Big O(n log n). In Computer Science logarithmic has a base 2, this is also true since we divide the array into two subarrays then conquer “n” times. “Log n” is in relation to “n” where it refers to the number of “n” times we need to perform “log n” times.
It may seem like not a huge deal, but when working with big data, it makes a huge difference. This is probably if not one of the most efficient sorting algorithms.

6.4. NP stands for Non-Polynomial, this complexity class is about proving the problem given in a polynomial time. This means that to prove a solution of a problem in NP class can be verified in using P class. It requires expensive computation. For instance, in a graph how can we find a specific route from A - B. We have to test and verify all possibilities and then determine a specific path after calculation it is the most optimal path.

6.5. NP-complete class is also a part of NP class that contains much more difficult problems. In NP-complete problems don't have known polynomial time algorithm. The challenge of NP problem is to find a solution with an algorithm that can provide a solution in polynomial time. In a real world example, this could be putting together a large puzzle piece. There is no solution to how you can solve the same puzzle every time when you break it down and put it back to pieces again and again to be deterministic. You have to eventually try to fit pieces by pieces together. Imagine one day in the future, someone has a magic to make a solution of solving a puzzle in a deterministic way. If I do exact way, I have determined result. Today is magic, tomorrow maybe I have a solution. This also needs to be tested and proven in order to be reduced to P class from NP class.