—-

Studies SDE Interview

Bruna Zamith Santos

05/2020
http://bzamith.github.io

Contents

1 Behavioral Interview

- Study Amazon's Leadership Principles (LPs)[1]
- Try to think of examples from previous works that fit each LP
  - Having more than one example per LP is desirable
  - The more diverse the examples are, the better it is
- Structure each example in Amazon's STAR Method[2]
- Prepare questions to ask the interviewers

[1] https://www.aboutamazon.com/working-at-amazon/our-leadership-principles
[2] https://www.amazon.jobs/en/landing_pages/in-person-interview

2 Coding

2.1 Big-O

- Big-O Cheatsheet
  - https://github.com/bzamith/BigO-Cheatsheet
- Asymptotic analysis
- Best case, worst case, expected case
- Space complexity and Time complexity
- Stack space (from recursion) counts as space complexity as well

```
int sum(int n) {
    if(n <= 0)
        return 0;
    return (n + sum (n-1));
}

Stack memory:
- sum(4)
 - sum(3)
  - sum(2)
   - sum(1)
    - sum(0)
```

- Drop constants and non-dominant terms
  - $O(2N) = O(N)$
  - $O(N^2 + N) = O(N^2)$
  - $O(N + log(N)) = O(M)$
- Add vs Multiply

```
> Sequential loops: O(A+B)
for(int a: arrA) {
    print(a);
}
for(int b: arrB) {
    print(b);
}

> Nested loops: O(A*B)
for(int a: arrA) {
    print(a);
    for(int b: arrB) {
        print(a + "."+ b);
    }
}
```

- Comparison
  - $O(1) < O(log(N)) < O(N) < O(N * log(N)) < O(N^2) < O(N^3) < O(2^N) < O(N!)$
- Amortized Time
  - ArrayList example: When inserting, if not full, O(1). If full, O(N), because it will create an array with double the current size and copy the previous N elements.

- $O(log(N))$
  - Example: Binary Search

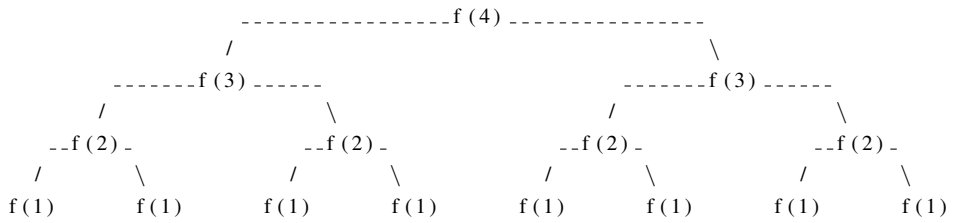
```
N = 16 /* divide by 2 */
N = 8 /* divide by 2 */
N = 4 /* divide by 2 */
N = 2 /* divide by 2 */
N = 1

2^k = N
k = log(N) base 2
```

  - When you see a problem where the number of elements in the problem space gets halved each time, that will likely be a $O(log(N))$ runtime

- Recursive runtime

```
int f(int n) {
    if(n <= 1) {
        return (1);
    }
    return (f(n-1) + f(n+1));
}
```



Level	#Nodes	...
0	1	2^0
1	2	2^1
2	4	2^2
3	8	2^3
4	16	2^4
		———
		2^(N+1) − 1

  - $O(2^N)$ time complexity
  - Although we have $O(2^N)$ in the tree total, only $O(N)$ exist at any given time. So $O(N)$ space complexity.

2.2 Tree Traversals

- Binary Tree: Tree data structure in which each node has at most two children
- Binary Search Tree:
  - The left subtree of a node contains only nodes with keys lesser than the node's key
  - The right subtree of a node contains only nodes with keys greater than the node's key
  - The left and the right subtree each must also be a binary search tree
- Inorder Traversal
  - Example:

```
public void inorderTraversal(TreeNode root) {
    if(root != null) {
        inorderTraversal(root.left);
        System.out.println(root.data + " ");
        inorderTraversal(root.right);
    }
}
```

```
       __1_______
      /          \
     2           ___3
    / \         /    \
   4   5       6      7
  /           / \
 8           9   10
```

```
8 > 4 > 2 > 5 > 1 > 9 > 6 > 10 > 3 > 7
```

- Inorder Traversal of a Binary Search Tree will always give you nodes in a sorted manner

- Preorder Traversal
  - Example:

```
public void preorderTraversal(TreeNode root) {
    if(root != null) {
        System.out.println(root.data + " ");
        preorderTraversal(root.left);
        preorderTraversal(root.right);
    }
}
```

```
       __1_______
      /          \
     2           ___3
    / \         /    \
   4   5       6      7
  /           / \
 8           9   10
```

```
1 > 2 > 4 > 8 > 5 > 3 > 6 > 9 > 10 > 7
```

- Postorder Traversal
  - Example:

```
public void postorderTraversal(TreeNode root) {
    if(root != null) {
        postorderTraversal(root.left);
        postorderTraversal(root.right);
        System.out.println(root.data + " ");
    }
}
```

```
       __1_______
      /          \
     2           ___3
    / \         /    \
   4   5       6      7
  /           / \
 8           9   10
```

```
8 > 4 > 5 > 2 > 9 > 10 > 6 > 7 > 3 > 1
```

- Level Order Traversal
  - Breadth-first search (Section 2.4):
  - Example:

```
public void levelorderTraversal(TreeNode root) {
    if(root == null) {
        return;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    while(!queue.isEmpty()) {
        TreeNode node = queue.remove();
        System.out.println(node.data + " ");
        if(node.left != null) {
            queue.add(node.left);
        }
        if(node.right != null) {
            queue.add(node.right);
        }
    }
}
```

```
        __1_______
       /          \
      2           ___3
     / \         /    \
    4   5       6      7
   /           / \
  8           9   10
```

```
1 > 2 > 3 > 4 > 5 > 6 > 7 > 8 > 9 > 10
```

2.3 Divide and Conquer

- Binary Search
  - $O(log(N))$
  - It is a divide and conquer algorithm. It divides a large array into two smaller sub-arrays and the recursively or iteratively operate the subarrays. But instead of operating on both subarrays, it discards one subarray and continues on the other one. Needs to be sorted[3].
  - Case 1: if $target == A[mid]$, return mid
  - Case 2: if $target < A[mid]$, right = mid - 1
  - Case 3: if $target > A[mid]$, left = mid + 1
  - Iterative solution

```
public int binarySearch(int[] A, int x) {
    int left = 0;
    int right = A.length - 1;
    while(left <= right) {
        int mid = (left+right)/2;
        if(x == A[mid]) {
            return mid;
```

[3] https://www.techiedelight.com/binary-search/

```
        }
        else if(x < A[mid]) {
            right = mid - 1;
        }
        else {
            left = mid + 1;
        }
    }
}
```

  - Recursive solution

```
public int binarySearch(int[] A, int left, int right, int x) {
    if(left > right) {
        return -1;
    }
    int mid = (left+right)/2;
    if(x == A[mid]) {
        return mid;
    }
    else if(x < A[mid]) {
        return binarySearch(A, left, mid-1, x);
    }
    else {
        return binarySearch(A, mid+1, right, x);
    }
}
```

- Maximum Sum Subarray
  - Given an array of integers, find maximum sum subarray among all possible subarrays
  - Example: [2, -4, (1, 9, -6, 7), -3]
  - Use divide and conquer:
    - $O(N * log(N))$
    - Divide the array into two equal subarrays
    - Recursively calculate the max subarray sum for left subarray
    - Recursively calculate the max subarray sum for right subarray
    - Find the max subarray sum that crosses mid element
    - Return the max of above three sums

```
public int maxSum(int[] A, int left, int right) {
    if(right == left) {
        return A[left];
    }
    int mid = (left+right)/2;
    int leftMax = Integer.MIN_VALUE;
    int sum = 0;
    for(int i = mid; i >= left; i--) {
        sum += A[i];
        if(sum > leftMax) {
            leftMax = sum;
        }
    }
    int rightMax = Integer.MIN_VALUE;
    sum = 0;
    for(int i = mid+1; i <= right; i--) {
        sum += A[i];
        if(sum > rightMax) {
```

```
            rightMax = sum;
        }
    }
    int maxLeftRight = Integer.max(maxSum(A,left,mid), maxSum(A,mid+1,right
    ));
    return Integer.max(maxLeftRight,leftMax+rightMax);
}
```

- Power function
  - Implement pow(x,n)
  - Use divide and conquer:
    * $O(N)$

```
public int pow(int x, int n) {
    if(n==0) {
        return 1;
    }
    if((n&1)==1) { // odd
        return x * pow(x,n/2) * pow(x,n/2);
    }
    else { // even
        return pow(x,n/2) * pow(x,n/2);
    }
}
```

    * Optimizing ($O(log(N))$):

```
public int pow(int x, int n) {
    if(n==0) {
        return 1;
    }
    int pow = pow(x,n/2);
    if((n&1)==1) { // odd
        return x * pow * pow;
    }
    else { // even
        return pow * pow;
    }
}
```

- Find frequency of each element in a sorted array
  - Split the array to two equal halves and recur for both of the halves. The base condition checks if the last element of the subarray is the same as it first element. If they are equal, then that implies that all elements in the subarray are similar (since the array is sorted) and we update the element county by number of elements in the subarray[4].
  - $O(m * log(n))$, with m the number of distinct elements in the array and n the size of the input.

```
public void findFrequency(int[] A, int left, int right, Map<Integer, Integer>
    freq) {
    if(left > right) {
        return;
    }
    if(A[left] == A[right]) {
        Integer count = freq.get(A[left]);
        if(count == null) {
            count = 0;
        }
```

[4] https://www.techiedelight.com/find-frequency-element-sorted-array-containing-duplicates/

```
            freq.put(A[left],count+(right-left+1));
            return;
        }
        int mid = (left+right)/2;
        findFrequency(A, left, right, freq);
        findFrequency(A,mid+1,right, freq);
    }
```

2.4 Breadth-first search

BFS is an algorithm for traversing (or searching) trees or graph data structures. It starts at the root and explores the neighbor nodes first, before moving on to the next level neighbors[5].

- Applications of BFS:
  - Finding the shortest path between two nodes u and v, with path length measured by number of edges (advantage over depth first search)
  - Testing a graph bipartiteness
  - Minimum Spanning Tree for unweighted graph
  - Finding nodes in any connected component of a graph
  - Fold-Fulkerson method for computing the maximum flow in a flow network
  - Serialization/Deserialization of a binary tree
- Implementation:
  - Queue
  - Finds the shortest path
  - Requires more memory than a DFS
- In matrix:

```
from collections import deque
def bfs(matrix):
    if not matrix:
        return []
    rows, cols = len(matrix), len(matrix[0])
    visited = set()
    directions = ((0,1),(0,-1),(1,0),(-1,0))
    for i in range(rows):
        for j in range(cols):
            traverse(i,j)

def traverse(i,j):
    queue = deque([(i,j)])
    while queue:
        curr_i, curr_j = queue.popleft()
        if(curr_i, cuur_j) not in visited:
            visited.add((curr_i,curr_j))
            for direction in directions:
                next_i = curr_i + direction[0]
                next_j = curr_j + direction[1]
                if 0 <= next_i < rows and 0 <= next_j < cols:
                    queue.append((next_i, next_j))
```

[5] https://www.techiedelight.com/breadth-first-search/

2.5 Depth-first search

We start at the root (or another arbitrarily select node) and explore each branch completely before moving on to the next branch. DFS is often preferred if we want to visit every node in the graph. In DFS, we visit a node a and then iterate through each of a's neighbors. When visiting a node b that is a neighbor of a, we visit all of b's neighbors before going to a's neighbors. That is, a exhaustively searches b's branch before any of its other neighbors. Note that pre-order and other forms of tree traversal are a form of DFS. The key difference is that when implementing this algorithm for a graph, we must check if the node has been visited [1].

```
void search(Node root) {
    if(root == null) {
        return;
    }
    visit(root);
    root.visited = true;
    for each(Node n in root.adjacent) {
        if(n.visited == false) {
            search(n);
        }
    }
}
```

2.6 Quicksort

Uses the divide and conquer technique (Section 2.3).

- Implementation:
  - Find a pivot p (usually the first element)
  - If the elements of array x are rearranged so that p is put in position j and that the following conditions are taken into account:
    - All the elements in between positions 0 and $j-1$ are smaller or equal to p
    - All the elements in between positions $j+1$ and $n-1$ are higher than p
  - Then p will be kept in position j at the end of sorting
  - Repeat the process to subarrays $x[0..j-1]$ and $x[j+1..n-1]$

```
def swap(i,j):
    aux = elements[i]
    elements[i] = elements[j]
    elements[j] = aux

def partition(start, end):
    i = start
    for j in range(start,end):
        if elements[j] <= elements[end]:
            swap(i,j)
            i += 1
        swap(i,end)
        return i

def quickSort(start,end):
    if start >= end:
        return
    pivot = partition(start, end)
    quickSort(start, pivot-1)
```

```
    quickSort(pivot+1, end)

if __name__ == "__main__":
    quickSort(0, len(elements)-1)
    print(elements)
```

- Big O:
  - Worst case: $O(N^2)$
  - Average case: $O(N * log(N))$
  - Space complexity: $O(log(N))$
- Performance depends largely on pivot selection

2.7 Mergesort

Uses the divide and conquer technique (Section 2.3)[6].

- Divide
  - If q is the half-array point between p and r, then we can split the subarray $A[p..n]$ into two arrays $A[p..q]$ and $A[q+1..n]$
- Conquer
  - We try to sort both subarrays $A[p..q]$ and $A[q+1..r]$. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.
- Combine
  - When the conquer step reaches the base step and we get two sorted subarrays $A[p..q]$ and $A[q+1..r]$ for array $A[p..r]$, we combine the results by creating a sorted array A[p..r] from two sorted subarrays $A[p..q]$ and $A[q+1..r]$

```
def merge(array, p, q, r):
    arrayA = array[p:q+1]
    arrayB = array[q+1:r+1]
    total = r - p + 1
    result = [0] * total
    pA = pB = pR = 0
    while pR < total:
        if pA == len(arrayA) - 1:
            result[pR:total] = arrayB[pB:len(arrayB)]
            return result
        if pB == len(arrayB) - 1:
            result[pR:total] = arrayB[pA:len(arrayA)]
            return result
        if arrayA[pA] <= arrayB[pB]:
            result[pR] = arrayA[pA]
            pR += 1
            pA += 1
        else:
            result[pR] = arrayB[pB]
            pR += 1
            pB += 1
        return result
```

[6] https://medium.com/@paulsoham/merge-sort-63d75df76388

```
def mergeSort(array, p, r):
    if p >= r:
        return
    q = (p+r)//2
    mergeSort(array, p, q)
    mergeSort(array, q+1, r)
    merge(array, p, q ,r)
```

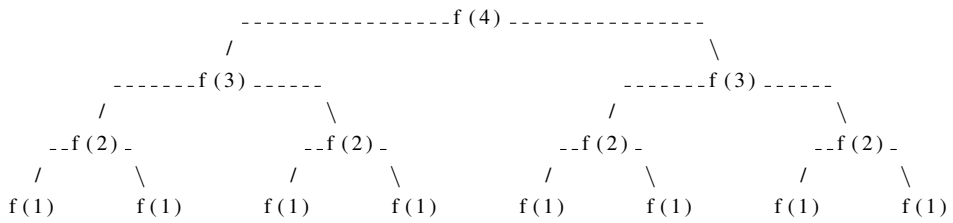
- Big O:
  - Worst case: $O(N * log(N))$
  - Average case: $O(N * log(N))$
  - Space complexity: $O(N)$

2.8 Dynamic Programming

Taking a recursive algorithm and finding the overlapping subproblems. Use memoization!

- Fibonacci without DP

```
int f(int i) {
    if(i==0) return 0;
    if(i==1) return 1;
    return f(i-1) + f(i-2);
}
```



- DP with top-down approach

```
int f(int n) {
    return f(n, new int[n+1]);
}

int f(int i, int[] memo) {
    if(i==0||i==1) return i;
    if(memo[i]==0) {
        memo[i] = f(i-1, memo) + f(i-2, memo);
    }
    return memo[i];
}
```

- DP with bottom-up approach

```
int f(int n) {
    if(n==0) return 0;
    int a = 0;
    int b = 1;
    for (int i=2; i < n; i++) {
        int c = a + b;
        a = b;
        b = c;
    }
    return a + b;
}
```

3 Python

3.1 Set

Collection of items. Uses a hash.

- Creating a set

```
set()
```

- Checking if item is in set
  - Time complexity: $O(1)$ on average
  - Worst case: $O(N)$
- Adding elements

```
set.add()
```

- Union
  - Two sets can be merged using union() function
  - $O(len(s1) + len(s2))$
- Intersection
  - $O(min(len(s1), len(s2)))$
- Difference

```
difference()
```

- Examples

```
a = set()
a.add('a')
a.add('b')
a.add('a')
'a' in a #return True
a.remove('b')
```

3.2 List

- [] vs $list()$
  - [] is literal and $list()$ is function call
  - [] is faster because it doesn't involve loading and calling a separate function
  - $[(a, b, c)]$ returns $[(a, b, c)]$ and $list((a, b, c))$ returns $[a, b, c]$
  - $list()$ accepts only iterables as argument
- Examples

```
sea_creatures = ['shark', 'fish', 'squid', 'mantis', 'anemone']
sea_creatures[1:4] # returns ['fish', 'squid', 'mantis']
sea_creatures[:3] # returns ['shark', 'fish', 'squid']
sea_creatures[2:] # returns ['squid', 'mantis', 'anemone']

numbers = [0,1,2,3,4,5,6,7,8,9,10,11,12]
numbers[1:11:2] # returns [1,3,5,7,9]
numbers[::3] # returns [0,3,6,9,12]
```

```
numbers + [13,14] # O(1). Worst case: Double the size, O(N)

letters = ['a','b','c']
letters*2 # returns ['a','b','c','a','b','c']

del sea_creatures[1:3] # returns ['shark', 'mantis', 'anemone']. O(N)
sea_names = [['shark', 'octopus', 'squid']['Sammy', 'Jesse', 'Drew']]
sea_names[0][0] = 'shark'
sea_names[1][2] = 'Drew'
```

- += vs $append()$
  - $append()$ is less time and space expensive because it doesn't have to create a new list every time
- min and max
  - Both costs $O(N)$
- in vs $find()$
  - "x in s" costs $O(N)$ and returns $true$ or $false$
  - $find()$ returns the index. $l.find(':')$
- $insert()$
  - $list.insert(index, element)$
  - $O(N)$

3.3 Dict

- Examples

```
thisDict = {"brand": "Ford",
            "model": "Mustang",
            "year": 1964}
x = thisDict["model"] # or
x = thisDict.get("model")
thisDict["year"] = 2018

for x in thisDict:
    print(x) # prints key name
for x in thisDict:
    print(thisDict[x]) # prints values
for x in thisDict.values():
    print(x)
for x,y in thisDict.items():
    print(x,y)

# Checks if "model" is present in the dictionary
"model" in thisDict

# Adds item
thisDict["color"] = "red"

# Removes item
thisDict.pop("model")

# Clear dict
thisDict.clear()
```

- How Python dict works[7]
  - Implemented as hash tables
  - Uses open addressing to resolve hash collisions
  - Contiguous block of memory
  - Each slot in the table can store one and only one entry
  - Each entry in the table is actually a combination of three values $< hash, key, value >$
  - When a new dict is initialized it starts with 8 slots
  - When adding entries to the table, we start with a slot that is based on the hash of the key
  - If that slot is empty, the entry is added to the slot
  - If that slot is occupied, compares the hash and the key of the entry in the slot against the hash and the key of the current entry to be inserted. If both match, then it thinks the entry already exists and gives up. If they don't match, it starts probing.
  - Probing means it searches slot by slot to find an empty slot
  - The dict will be resized if it is two-thirds full

3.4 Sort

Python's "$list.sort()$" and "$sorted()$" methods use timesort, a stable mergesort and insertionsort hybrid. The first sorts in-place and the second returns a new sorted list. The first is a bit more efficient if you don't need to keep the original list. They both take a parameter "key", which is a key function (i.e. if you want to sort by a specific property).

- Time complexity
  - Worst case: $O(N * log(N))$
  - Best case: $O(N)$
  - Average case: $O(N * log(N))$
- Space complexity
  - $O(N)$
- Examples

```
sorted([5,2,3,1,4])

a = [5,2,3,1,4]
a.sort()

student_tuples = [('joe', 'A', 15),
                  ('jane', 'B', 12),
                  ('dave', 'B', 10)]
sorted(student_tuples, key = lambda student:student[2])

class Student:
    def __init__(self, name, grade, age):
        self.name = name
        self.grade = grade
        self.age = age
```

[7] https://stackoverflow.com/questions/21595048/how-python-dict-stores-key-value-when-collision-occurs

```
student_objects = [Student('joe', 'A', 15),
                   Student('jane', 'B', 12),
                   Student('dave', 'B', 10)]
sorted(student_objects, key = lambda student:student.age)
```

3.5 Join

- Join all items in a tuple into a string, using a hash character as separator.

```
x = myseparator.join(myDict) # myDict is any iterable
```

- Equivalent of C's stringBuilder. Imagine you are concatenating a list of strings as shown below:

```
our_str = ''
for num in range(loop_count):
    our_str += 'num'
return our_str
```

- On each concatenation, a new copy of the string is created and the two strings are copied over, character by character. Join can help you avoid this problem.

```
str_list = []
for num in range(loop_count):
    str_list.append('num')
return ''.join(str_list)
```

- Big O
  - $O(N)$ where N is the total length of the output
- *join* vs *split*

```
txt = "hello, my name is Peter, I am 26 years old"
x = txt.split(", ")
print(x)
# ['hello', 'my name is Peter', 'I am 26 years old']
```

3.6 Strip

- $O(N)$
- Example

```
str = "ooooo hi!! ooo"
str = str.strip('o')
print(str) # " hi!! "
```

4 Data Structures

4.1 Linked List

Represents a sequence of nodes. In a singly linked list, each node points to the next node in the list. In a doubly linked list, each node points to both the previous and the next nodes in the list. Unlike an array, a linked list does not provide constant time access to a particular ”index” within the list. The benefit is that you can add and remove items from the beginning of the list in constant time.

```
class Node {
    Node next = null;
    int data;

    public Node(int d) {
        data = d;
    }

    void appendToTail(int d) {
        Node end = new Node(d);
        Node n = this;
        while (n.next != null) {
            n = n.next;
        }
        n.next = end;
    }
}
```

- Deleting a node

```
Node deleteNode(Node head, int d) {
    Node n = head;
    if (n.data == d) {
        return head.next;
    }
    while (n.next != null) {
        if (n.next.data == d) {
            n.next = n.next.next;
            return head;
        }
        n = n.next;
    }
    return head;
}
```

4.2 Hash Table

Data structure that maps keys to values for highly efficient lookup. Array of linked lists + hash code function.

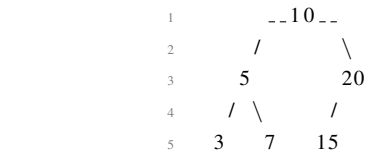
1. First, compute the key’s hash code, which will usually be an int or long. Two different keys could have the same hash code.
2. Map the hash code to an index in the array.
3. At this index, there is a linked list of keys and values. Store the key and the value in this index. We must use a linked list because of collision.

Picking a good hash function and a good hash size avoid collisions. Another way to resolve collisions is probing:

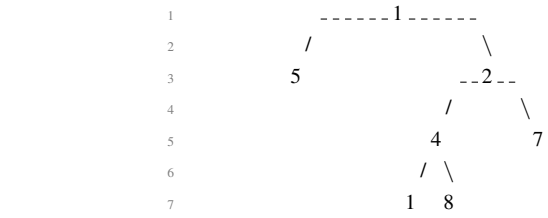
1. Use a hash function and find the index for a key
2. If that spot contains a value, use the next available spot ("a higher index"). If you reach the end of the array, go back to the front.
3. If the item was deleted, mark "deleted" in the slot.

4.3 Binary Tree

- Complete Binary Tree: Every level of the tree is fully filled, except for perhaps the last level. To the extent that the last level is filled, it is filled left to right.



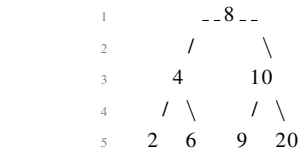
- Full Binary Tree: Every node has either zero or two children



- Perfect binary Tree: Both full and complete

4.4 Binary Search Tree

A Binary Search Tree is a binary tree in which every node fits a specific ordering property: All left descendants $<=$ N $<$ all right descendants.



- Sorted array to balanced BST:
  - Big O: $O(N)$

1. Get the middle of the array and make it root
2. Recursively do the same for left half and righ half
   (a) Get the middle of the left half and make it left child of the root created in the step 1
   (b) Get the middle of the right half and make it right child of the root created in the step 1

```
def sortedArrayToBST(arr):
    if not arr:
        return None
    mid = (len(arr))//2
    root = Node(arr[mid])
    root.left = sortedArrayToBST(arr[:mid])
    root.right = sortedArrayToBST(arr[mid+1:])
    return root
```

- Search in BST:
  - Big O: $O(log(N))$

```
def searchBST(root, key):
    if root is None or root.val == key:
        return root
    if root.val < key:
        return searchBST(root.right, key)
    return searchBST(root.left, key)
```

- Height of a BST:
  - Min: $log(N)$
  - Max: N
- Insert in BST

```
def insertBST(root, node):
    if root is None:
        root = node
    else:
        if root.val < node.val:
            if root.right is None:
                root.right = node
            else:
                insertBST(root.right, node)
        else:
            if root.left is None:
                root.left = node
            else:
                insertBST(root.left, node)
```

4.5 Min-heap

Complete binary tree where each node is smaller than its children. The root, therefore, is the minimum element in the tree.

- Insert: We begin by inserting the element at the bottom, at thre rightmost spot. Then we "fix" the tree by swapping the new element with its parent, until we find an appropriate spot for the element.
  - Big O: $O(log(N))$
- Extract Min Element: First, we remove the minimum element and swap it with the last element in the heap (the bottom most, rightmost element). Then, we bubble down this element, swapping it with one of its children until the minheap property is restored.
- Min-Heap in Python: Represented as an array. Root element is $arr[0]$. For any i'th node:
  - $arr[(i-1)/2]$ returns its parent
  - $arr[(2*i)+1]$ returns its left child node
  - $arr[(2*i)+2]$ returns its right child node

```
class MinHeap:
    def __init__(self, maxsize):
        self.maxsize = maxsize
        self.size = 0
        self.Heap = [0] * (self.maxsize + 1)
```

```
        self.Heap[0] = -1 * sys.maxsize
        self.FRONT = 1
    def parent(self, pos):
        return pos//2
    def leftChild(self, pos):
        return (2*pos) + 1
    def rightChild(self, pos):
        return (2*pos) + 2
    def isLeaf(self, pos):
        if pos >= (self.size//2) and pos <= self.size:
            return True
        return False
    def swap(self, fpos, spos):
        temp = self.Heap[fpos]
        self.Heap[fpos] = self.Heap[spos]
        self.Heap[spos] = temp
    def minHeapify(self, pos):
        if not self.isLeaf(pos):
            if(self.Heap[pos] > self.Heap[self.leftChild(pos)] or self.Heap[pos] >
    self.Heap[self.rightChild(pos)]):
                if self.Heap[self.leftChild(pos)] < self.Heap[self.rightChild(pos)
    ]:
                    self.swap(pos, self.leftChild(pos))
                    self.minHeapify(self.leftChild(pos))
                else:
                    self.swap(pos, self.rightChild(pos))
                    self.minHeapify(self.rightChild(pos))
    def insert(self, element):
        if self.size >= self.maxsize:
            return
        self.size += 1
        self.Heap[self.size] = element
        current = self.size
        while self.Heap[current] < self.Heap[self.parent(current)]:
            self.swap(current, self.parent(current))
            current = self.parent(current)
    def minHeap(self):
        for pos in range(self.size//2, 0, -1):
            self.minHeapify(pos)
    def remove(self):
        popped = self.Heap[self.FRONT]
        self.Heap[self.FRONT] = self.Heap[self.size]
        self.size -= 1
        self.minHeapify(self.FRONT)
        return popped
```

4.6 Stack

Last-in, first-out (LIFO)

```
stack = []
stack.append(1) # means stack.push(1)
stack.pop()
stack[-1] # means stack.peak()
if stack: # means stack.isEmpty()
    ...
```

4.7 Queue

First-in, first-out (FIFO)

```
queue = []
queue.append(1) # means queue.add(1)
queue.pop(0) # means queue.remove()
queue[0] # means queue.peak()
if queue: # means queue.isEmpty()
    ...

# or
from collections import deque
q = deque()
q.append('c')
q.popLeft()
```

5 Object-oriented Programming

5.1 Concrete Class, Abstract Class and Interface

Concrete class can be instantiated because it provides (or inherits) the implementation for all of its methods. Abstract class cannot be instantiated because at least one method has not been implemented. Interface must contain only method signatures and static members.

- In abstract class, keyword ”abstract” is mandatory to declare a method as an abstract. In interface, ”abstract” is optional.
- Interface can only have public methods
- One concrete class extends only one abstract class, but can implement many interfaces

```
interface Readable {
    public void open();
    public void read();
}

abstract class Animal() {
    String species;
    public void eat();
    abstract void walk();
}
```

- Abstract classes are used when we require classes to share a similar behavior (or methods). However, if we need classes to share method signatures, and not the method themselves, we should use interfaces
- Interfaces can add to the existing functionality of a class. They are not necessarily integral to the identity of the classes that reference them

5.2 Static

The keyword ”static” is used to refer to common property of all objects. Can be applied to variables, methods and nested classes. If a variable is static, it is assigned memory once and all objects of the class access the same variable[8].

```
class Citizen {
    static String country();
    Citizen() {
        country = "Brazil";
    }
    public static void setCountry(String c) {
        country = c;
    }
}
```

5.3 Inheritance

- Super class: The class whose features are inherited is known as super class
- Sub class: The class that inherits the other
- Reusability: Inheritance supports reusability
- In java, use $@override$ and $extends$

[8] https://www.educative.io/edpresso/how-to-use-static-keyword-in-java

5.4 Aggregation, Association and Composition

- Aggregation
  - Relationship in which an object contains one or more other subordinate objects as part as of its state. The subordinate has independent existence separate from their containing object. Has-a relationship[9].
  - "I have an object in which I've borrowed from someone else". When Foo dies, Bar may live on[10].

```
public class Foo {
    private Bar bar;
    Foo(Bar bar) {
        this.bar = bar;
    }
}
```

- Composition
  - "I own an object and I am responsible for its lifetime". When Foo dies, so does Bar.

```
public class Foo {
    private Bar bar = new Bar();
}
```

- Association
  - "I have a relationship with an object". Foo uses Bar.

```
public class Foo {
    void Baz(Bar bar);
}
```

- Dependency
  - One object is dependent on another object for its specification or implementation.
  - Dependency injection is a technique whereby one object supplies the dependencies of another object[11].

5.5 Coupling and Cohesion

- Coupling: Degree of interdependence between software modules
- Cohesion: Degree of which the elements of a module belong together
- Desirable: Less coupling, more cohesion

5.6 Polymorphism

The ability of an object reference to be used as if it referred to an object with different forms. Examples: Class inheritance and interface inheritance. The most common use occurs when a parent class reference is used to refer to a child class object[12].

[9] https://www.cs.kent.ac.uk/people/staff/djb/oop/glossary.html
[10] https://tinyurl.com/y2wjx3f9
[11] https://medium.com/@harivigneshjayapalan/dagger-2-for-android-beginners-di-part-i-f5cc4e5ad878
[12] https://www.tutorialspoint.com/java/java_polymorphism.htm

5.7 Function overloading

Example:

```
int Volume(int s);
int Volume(double r, int h);
int Volume(long l, int b, int h);
```

6 Design Patterns

6.1 Template Method

6.2 Strategy

6.3 State

6.4 Composite

6.5 Singleton

6.6 Builder

6.7 Factory

7 Architecture

7.1 API

- Application Programming Interface
- API allows various software applications to communicate with one another over the internet.
- The most common methods seen in APIs are:
  - GET (asks to retrieve a resource)
  - POST (asks to create a new resource)
  - PUT (asks to edit/update an existing resource)
  - DELETE (asks to delete a resource)
- API requests are made using HTTP
- Endpoint tell the server which resources that the client wants to interact with

7.2 Remote Procedure Call

RPC is a protocol that one program can use to request a service from another program located in another computer on a network. It uses client-server model. RPC is a synchronous operation. Examples: REST, SOAP.

7.3 Caching

An in-memory cache can deliver very rapid results. It is a simple key-value pairing and typically sits between your application layer and your data store. An in-memory cache can deliver very rapid results. It is a simple key-value pairing and typically sits between your application layer and your data store. When an application requests a piece of information, it first tries the cache. If the cache does not contain the key, it will then look up the data in the data store. (At this point, the data might-or might not-be stored in the data store.) When you cache, you might cache a query and its results directly. Or, alternatively, you can cache the specific object (for example, a rendered version of a part of the website, or a list of the most recent blog posts) [1].

Distributed applications typically implement either or both of the following strategies when caching data:

- Using a private cache, where data is held locally on the computer that's running an instance of an application or service.
- Using a shared cache, serving as a common source that can be accessed by multiple processes and machines.

In both cases, caching can be performed client-side and server-side. Client-side caching is done by the process that provides the user interface for a system, such as a web browser or desktop application. Server-side caching is done by the process that provides the business services that are running remotely. Caching typically works well with data that is immutable or that changes infrequently. Examples include reference information such as product and pricing information in an e-commerce application, or shared static resources that are costly to construct. Some or all of this data can be loaded into the cache at application startup to minimize demand on resources and to improve performance. It might also be appropriate to have

a background process that periodically updates reference data in the cache to ensure it is up-to-date, or that refreshes the cache when reference data changes.[13]

7.4 Database Partitioning (Sharding)

Sharding is a database architecture pattern related to horizontal partitioning — the practice of separating one table's rows into multiple different tables, known as partitions. Each partition has the same schema and columns, but also entirely different rows. Likewise, the data held in each is unique and independent of the data held in other partitions.

It can be helpful to think of horizontal partitioning in terms of how it relates to vertical partitioning. In a vertically-partitioned table, entire columns are separated out and put into new, distinct tables. The data held within one vertical partition is independent from the data in all the others, and each holds both distinct rows and columns.

Sharding involves breaking up one's data into two or more smaller chunks, called logical shards. The logical shards are then distributed across separate database nodes, referred to as physical shards, which can hold multiple logical shards. Despite this, the data held within all the shards collectively represent an entire logical dataset.

Database shards exemplify a shared-nothing architecture. This means that the shards are autonomous; they don't share any of the same data or computing resources. In some cases, though, it may make sense to replicate certain tables into each shard to serve as reference tables. For example, let's say there's a database for an application that depends on fixed conversion rates for weight measurements. By replicating a table containing the necessary conversion rate data into each shard, it would help to ensure that all of the data required for queries is held in every shard.

Oftentimes, sharding is implemented at the application level, meaning that the application includes code that defines which shard to transmit reads and writes to. However, some database management systems have sharding capabilities built in, allowing you to implement sharding directly at the database level.

The main appeal of sharding a database is that it can help to facilitate horizontal scaling, also known as scaling out. Horizontal scaling is the practice of adding more machines to an existing stack in order to spread out the load and allow for more traffic and faster processing. This is often contrasted with vertical scaling, otherwise known as scaling up, which involves upgrading the hardware of an existing server, usually by adding more RAM or CPU.

It's relatively simple to have a relational database running on a single machine and scale it up as necessary by upgrading its computing resources. Ultimately, though, any non-distributed database will be limited in terms of storage and compute power, so having the freedom to scale horizontally makes your setup far more flexible.

Another reason why some might choose a sharded database architecture is to speed up query response times. When you submit a query on a database that hasn't been sharded, it may have to search every row in the table you're querying before it can find the result set you're looking for. For an application with a large, monolithic database, queries can become prohibitively slow. By sharding one table into multiple, though, queries have to go over fewer rows and their result sets are returned much more quickly.

Sharding can also help to make an application more reliable by mitigating the impact of outages. If your application or website relies on an unsharded database, an outage has the potential to make the entire application unavailable. With a sharded database, though, an outage is likely to affect only a single shard. Even though this might make some parts of the application or website unavailable to some users, the overall impact would still be less than if the entire database crashed[14]

[13] `https://docs.microsoft.com/en-us/azure/architecture/best-practices/caching`
[14] `https://www.digitalocean.com/community/tutorials/understanding-database-sharding`

7.5 Non Relational Databases

Joins in a relational database such as SQL can get very slow as the system grows bigger. For this reason, you would generally avoid them. Denormalization is one part of this. It means adding redundant information into a database to speed up reads. For example, imagine a database describing projects and tasks (where a project can have multiple tasks). You might need to get the project name and the task information. Rather than doing a join across these tables, you can store the project name within the task table.

Or you can go with a NoSQL database. It does not support joins and might structure data in a different way. It is designed to scale better. The reason why so many NoSQL systems have eventual consistency is that virtually all of them are designed to be distributed.

References

[1] G. L. McDowell. *Cracking the coding interview: 189 programming questions and solutions.* 2015.