C949v3 Study Guide

Competencies and Topics

Explains Algorithms - 29% of assessment

Introduction to Algorithms

Factors of an algorithm (not found in textbook)
Characteristics of an algorithm (not found in textbook)
Data Flow Diagramming (not found in textbook)
Classes (Chapter 7)
Recursive algorithms (Chapter 8, 11)
Linear search (Chapter 9)
Binary search (Chapter 9)
Sorting algorithms Chapter (9.5 – 9.9, 12)
Big O - Time complexity, worst case, efficiency analysis (Chapter 10)
Data structures (Chapter 10)

Tips:

You will need to be able to read pseudocode. For these problems, practice carefully writing out the steps on your whiteboard. One misstep will give you the incorrect choice! This is the hardest section. First focus on Data Types and Data Structures.

Example:

48 What is displayed when n = 2 in this pseudocode?

```
for(int i = 2; i <= n; i++){
for(j = 0; j <= n;){
   display j;
   j = j + n/2; the division is integer division, decimal part neglected
}
}
```

- 1, 0, 2
- 1, 2, 0
- 0, 1, 2
- 0, 2, 1

1st for loop	2nd for loop	Display
i=2: i=2<=2 → i=3	**j=0<=n=2**	display **0**; **j=1**
	j=1<=n=2	display **1**; **j=2**
	j=2<=n=2	display **2**; **j=3**
	j=3!<=n=2 → EXIT loop	

i=2: i=3!<=2→ EXIT loop		

✅ **0, 1, 2**

Determines Data Structure Impact - 31% of assessment

Implementation of Data Structures

Array (Chapter 10.1, 10.4, 13.16)
Linked list (doubly & singly linked) (Chapter 13.1,13.2,13.7,13.11,13.12,13.15,13.16)
Stack operations (Chapter 14.1)
Queue operations (push, pop, peek, enqueue, dequeue) (Chapter 14.4)
Deque operations (Chapter 14.8)
Hash table (Chapter 15)
Trees (leaf nodes, root, height) (Chapter 16)
Heaps (Chapter 18)
Sets (including: intersection) (Chapter 19.1 & 19.2)
Graph (vertices, adjacency) (Chapter 20)

Tips

Data structure functions. From the pre-assessment, it appears this section will focus mostly on the basic functions of stacks, queues, and dictionaries with a focus on .pop() and .push()

Know what .pop() does for:

- stacks
- queues
- priority queues

Be familiar with tree traversal

- inorder traversal
- preorder traversal

Know the major dictionary functions

Hash tables

- hashing
- chaining
- hash key
- modulo operator (Chapter 2.7 & 15.1)

Design of Data Structures

ADT characteristics (Chapter 10.4)
Dictionary (Chapter 3.5, 6)
Lists (Chapter 6.1)

Tips

Design properties. Know the principal design properties of the listed data structures, e.g., how things are compared, inserted, deleted, which have indexing, which has hierarchies, etc.

Linear Data Structures

- Arrays
- Linked list
- Doubly linked list
- Queue
- Stack

Hierarchical Data Structures

- Tree (binary)
- Heap
  - child and parent
  - heapList
  - right/left child
  - min-heap
  - max-heap

Graph data structures

- weight and direction
- vertices/nodes and edges/node pairs
- directed/undirected graph
- Indexing of data structures

Applies Algorithms - 40% of assessment

Sorting Algorithms

Variable declaration (dynamic vs static) (Chapter 2)
Assignment operators (Chapter 2)
Types / List basics (Chapter 3.2)
Order of operations / precedence rules (Chapter 2 & 4)
Loops (Chapter 24)
Conditional statements / branching (Chapter 4)
Classes (Chapter 7)
Linear search (Chapter 9)
Binary search (Chapter 9)
Big O growth rate O(1), O(n), O(log n), O(n log n), O(n^2) (Chapter 9.3)
Sorting algorithms (Chapter 9.5 – 9.9, 12)
Data structures – arrays, linked list, heap, etc (Chapter 10)
ADTs - stack, deque, list, queue, etc. (Chapter 10)
Time complexity, worst case, efficiency analysis (Chapter 10)
Methods for list (Chapter 13.1)
Stack operations (Chapter 14.1)
Deque operations (Chapter 14.8)

Tips:

Big- O - Introductory video

Know the Big-$\mathcal{O}$ time-complexity for the major sorting algorithms. Note that these time complexities represent the worst-case scenario. Some sorting algorithms, like quicksort, may perform much faster in practice due to clever optimizations and partitioning techniques.

Selection Sort: O(n^2)
Insertion Sort: O(n^2)
Bubble Sort: O(n^2)
Quick Sort: O(n^2)
Merge Sort: O(n log n)
Heap Sort: O(n log n)
Radix Sort: O(nk), where k is the # of digits in the largest number in the array.

You will need to identify the major sorting algorithms from code.

Bubble: Look for something that swaps so the result can "bubble" to the top.
Bucket: Look for something that distributes the vales into "buckets" where they are individually sorted.
Merge: Look for something that continually splits a list in half.
Quicksort: Look for the keywords "pivot" and/or "split".

General Knowledge for Assessment

Data Structure Definitions:

Array: A data structure that stores a fixed-size sequential collection of elements of the same type. Elements in an array can be accessed using their index, which starts from 0.

Linked List: A data structure consisting of a sequence of nodes, each containing an element and a reference to the next node. Linked lists can be used to implement dynamic data structures, where the size of the data changes frequently.

Binary Search Tree: A binary tree data structure where each node has at most two children, and the left child is always less than its parent, while the right child is always greater than its parent. Binary search trees are commonly used for searching and sorting operations.

Hash Table: A data structure that uses a hash function to map keys to indices of an array, where values associated with the keys can be stored. Hash tables provide constant-time average case for basic operations such as insert, delete, and search.

Heap: A binary tree data structure where the parent nodes are always greater (or less) than their children. Heaps are used to implement priority queues, where the highest (or lowest) priority element is always at the root of the heap.

Abstract Data Types

List (and Array in Java)
Function: An ordered collection of elements of the same or different data types.
Distinctions: Elements are ordered and can be accessed using an index.
Underlying Data Structure: Dynamic array.
Basic Commands and Syntax: **[]** is used to create a list. **append()** adds an element to the end of the list. **pop()** removes and returns the last element.

Tuple
Function: An ordered collection of elements of the same or different data types.
Distinctions: Tuples are immutable, meaning their values cannot be changed.
Underlying Data Structure: Dynamic array.
Basic Commands and Syntax: **()** is used to create a tuple. Indexing and slicing are used to access tuple elements.

Stack
Function: A collection of elements that follows the Last-In-First-Out (LIFO) principle.
Distinctions: Elements are added and removed from the top of the stack.
Underlying Data Structure: Dynamic array or linked list.
Basic Commands and Syntax: **append()** adds an element to the top of the stack. **pop()** removes and returns the last element.

Queue
Function: A collection of elements that follows the First-In-First-Out (FIFO) principle.
Distinctions: Elements are added to the rear and removed from the front of the queue.
Underlying Data Structure: Linked list.
Basic Commands and Syntax: **append()** adds an element to the rear of the queue. **pop(0)** removes and returns the first element.

Deque
Function: A collection of elements that supports adding and removing elements from both ends.
Distinctions: Deque stands for "double-ended queue".
Underlying Data Structure: Linked list.
Basic Commands and Syntax: **append()** adds an element to the rear of the deque. **appendleft()** adds an element to the front of the deque. **pop()** removes and returns the last element. **popleft()** removes and returns the first element.

Bag
Function: An unordered collection of elements which may include duplicates.
Distinctions: Elements can be added and removed, but the order is not preserved.
Underlying Data Structure: Hash table.
Basic Commands and Syntax: **add()** adds an element to the bag. **remove()** removes an element from the bag.

Set
Function: An unordered collection of unique elements.
Distinctions: Sets do not allow duplicate elements.
Underlying Data Structure: Hash table.

Basic Commands and Syntax: **set()** creates a set. **add()** adds an element to the set. **remove()** removes an element from the set.

Priority Queue

Function: A collection of elements where each element has a priority associated with it.
Distinctions: Elements are ordered based on priority, not insertion order.
Underlying Data Structure: Heap.
Basic Commands and Syntax: **heapq** module is used to implement priority queue. **heappush()** adds an element to the heap. **heappop()** removes and returns the element with the highest priority.

Dictionary (Map)

Function: A collection of key-value pairs.
Distinctions: Keys are unique and used to access the corresponding value.
Underlying Data Structure: Hash table.
Basic Commands and Syntax: **{}** is used to create a dictionary. **keys()** returns a list of all the keys. **values()** returns a list of all the values. **items()** returns a list of all the key-value pairs.

Data Types

Boolean

Description: A data type with only two possible values, true or false.
Java Syntax: **boolean x = true;**
Python Syntax: **x = True**

Byte

Description: A data type that stores an 8-bit signed two's complement integer.
Java Syntax: **byte x = 10;**
Python Syntax: Not available in Python. The closest equivalent is the **int** data type.

Int

Description: A data type that stores a 32-bit signed two's complement integer.
Java Syntax: **int x = 10;**
Python Syntax: **x = 10**

Char

Description: A data type that stores a single Unicode character.
Java Syntax: **char x = 'a';**
Python Syntax: **x = 'a'**

Float

Description: A data type that stores a single-precision 32-bit floating point number.
Java Syntax: **float x = 3.14f;**
Python Syntax: **x = 3.14**

Double

Description: A data type that stores a double-precision 64-bit floating point number.
Java Syntax: **double x = 3.14;**
Python Syntax: **x = 3.14**

String

Description: A data type that stores a sequence of characters.
Java Syntax: **String x = "Hello, world!";**
Python Syntax: **x = "Hello, world!**

Intro to Programming Concepts

Assignment vs. Comparison

Description: In programming, we use the = sign to assign a value to a variable. We use the == sign to compare two values.
Example: **x = 5** assigns the value 5 to the variable x. **if (x == 5):** checks if the value of x is equal to 5.

Garbage Collection

Description: A process by which a computer's memory is automatically managed. It frees up memory that is no longer being used by the program.
Example: In Java, garbage collection is done automatically by the JVM. In Python, garbage collection is done by the Python interpreter.

Reference Count

Description: A count of the number of times an object is referred to in a program.
Example: In Python, every object has a reference count. When the reference count of an object becomes zero, the object is deleted by the garbage collector.

Memory Allocation

Description: The process of reserving memory space for an object in a program.
Example: In Java, memory is allocated using the new keyword. **MyClass obj = new MyClass();** allocates memory for an object of the MyClass class.

Linked Allocation

Description: A memory allocation technique where memory is allocated in linked nodes. Each node contains a pointer to the next node.
Example: Linked lists are a data structure that use linked allocation. Each node in the linked list contains a pointer to the next node.

Sequential Allocation

Description: A memory allocation technique where memory is allocated in a sequential manner.
Example: Arrays are a data structure that use sequential allocation. Memory is allocated for all the elements of the array in a sequential manner.

Pointer

Description: A variable that stores the memory address of another variable.
Example: In C++, we use pointers to access memory directly. **int *ptr** declares a pointer variable that can store the memory address of an integer.

Binary Search

Description: A search algorithm that finds the position of a target value in a sorted array by repeatedly dividing the search interval in half.
Example: To conduct a binary search on a list, we start by finding the middle element of the list. If the middle element is equal to the target value, we return its position. Otherwise, we repeat the search in the appropriate half of the list.

Null

Description: A value that represents the absence of a value or a null pointer.
Example: In Java, **null** is a special value that represents the absence of a value. It can be assigned to reference types.

Object Constructor

Description: A special method that is used to initialize an object when it is created.
Example: In Java, the constructor method has the same name as the class. It is called when a new object is created. **MyClass obj = new MyClass();** calls the constructor method of the MyClass class.

Stack

Description: A linear data structure that follows the Last In First Out (LIFO) principle.
Push: Adds an element to the top of the stack.
Pop: Removes the top element from the stack and returns it.
Peek: Returns the top element of the stack without removing it.

Queue

Description: A linear data structure that follows the First In First Out (FIFO) principle.
Enqueue: Adds an element to the back of the queue.
Dequeue: Removes the front element from the queue and returns it.
Peek: Returns the front element of the queue without removing it.

Priority Queue

Description: A specialized queue where elements are dequeued based on their priority.
Enqueue: Adds an element to the priority queue based on its priority.
Dequeue: Removes the highest-priority element from the priority queue and returns it.
Peek: Returns the highest-priority element of the priority queue without removing it.

Tree Traversal

Description: The process of visiting each node in a tree in a specific order.

Inorder Traversal: Visits the left subtree, then the current node, and finally the right subtree.
Preorder Traversal: Visits the current node, then the left subtree, and finally the right subtree.
Postorder Traversal: Visits the left subtree, then the right subtree, and finally the current node.

Dictionary

Description: A collection of key-value pairs where each key maps to a value.
Get: Returns the value associated with a specified key.
Set: Sets the value associated with a specified key.
Delete: Removes a key-value pair from the dictionary.

Hash Table

Description: A data structure that uses hashing to store and retrieve key-value pairs efficiently.
Hashing: The process of converting a key into an index that can be used to access a value in the hash table.
Chaining: A method for handling collisions in a hash table by storing multiple key-value pairs in the same index.
Hash Key: The result of hashing a key to determine its index in the hash table.
Modular Arithmetic: A method for hashing keys by taking the remainder of the key divided by the size of the hash table.

Arrays

Description: A data structure that stores a fixed-size sequence of elements of the same type in contiguous memory locations.
Comparison: Elements are compared by their index.
Insertion: Elements can be inserted at the end of the array in O(1) time. Inserting at the beginning or middle of the array takes O(n) time.
Deletion: Elements can be deleted in O(1) time from the end of the array. Deleting from the beginning or middle of the array takes O(n) time.
Indexing: Elements are accessed using their index.
Hierarchy: Arrays have a flat structure.

Linked List

Description: A data structure that consists of a sequence of nodes, each containing an element and a reference to the next node.
Comparison: Elements are compared by their value.
Insertion: Elements can be inserted at the beginning or end of the linked list in O(1) time. Inserting in the middle of the list takes O(n) time.
Deletion: Elements can be deleted in O(1) time if the node to be deleted is known. Deleting a node requires traversing the list in O(n) time to find the node to be deleted.
Indexing: Elements are accessed by traversing the list from the beginning or end.
Hierarchy: Linked lists have a flat structure.

Doubly Linked List

Description: A data structure that consists of a sequence of nodes, each containing an element and references to the previous and next nodes.
Comparison: Elements are compared by their value.
Insertion: Elements can be inserted at the beginning or end of the doubly linked list in O(1) time. Inserting in the middle of the list takes O(n) time.
Deletion: Elements can be deleted in O(1) time if the node to be deleted is known. Deleting a node requires traversing the list in O(n) time to find the node to be deleted.
Indexing: Elements are accessed by traversing the list from the beginning or end.
Hierarchy: Doubly linked lists have a flat structure.

Queue

Description: A data structure that follows the First-In-First-Out (FIFO) principle. Elements are added to the back of the queue and removed from the front of the queue.
Comparison: Elements are compared by their value.
Insertion: Elements are inserted at the back of the queue in O(1) time.
Deletion: Elements are deleted from the front of the queue in O(1) time.
Indexing: Elements are not accessed by index in a queue.
Hierarchy: Queues have a flat structure.

Stack

Description: A data structure that follows the Last-In-First-Out (LIFO) principle. Elements are added to the top of the stack and removed from the top of the stack.
Comparison: Elements are compared by their value.
Insertion: Elements are inserted at the top of the stack in O(1) time.
Deletion: Elements are deleted from the top of the stack in O(1) time.
Indexing: Elements are not accessed by index in a stack.
Hierarchy: Stacks have a flat structure.

Hierarchal Data Structures

Tree (Binary)

Description: A hierarchical data structure that consists of nodes connected by edges. Each node has at most two children.
Comparison: Nodes are compared by their value.
Insertion: Nodes are inserted by finding the appropriate location based on their value and adding a new node as a child.
Deletion: Nodes are deleted by finding the appropriate node based on their value and removing it along with its children.
Indexing: Nodes are not accessed by index in a tree.
Hierarchy: Trees have a hierarchical structure.

Heap

Description: A specialized tree-based data structure that is used to maintain the maximum or minimum element in a collection.
Child and Parent: Each node has at most two children and one parent. The parent of a node can be found by taking the floor of the node index divided by 2.

HeapList: A binary heap can be represented as a list, where the root node is at index 1 and the children of a node at index i are at indices 2i and 2i+1.
Right/Left Child: The left child of a node is at index 2i and the right child is at index 2i+1.
Min-Heap: A binary heap where the value of each parent node is less than or equal to its children.
Max-Heap: A binary heap where the value of each parent node is greater than or equal to its children.
Comparison: Nodes are compared by their value.
Insertion: New nodes are added to the heap by inserting them as a leaf node and then swapping them with their parent until the heap property is restored.
Deletion: The root node (which is the maximum or minimum element depending on the type of heap) is removed and replaced with the last leaf node. The new root is then swapped with its children until the heap property is restored.
Indexing: Nodes are not accessed by index in a heap.
Hierarchy: Heaps have a hierarchical structure.

Graph Data Structures

Graph Data Structures

Description: A collection of nodes (vertices) connected by edges.
Weight and Direction: Edges can have weights to represent the strength or cost of the connection between two vertices. Edges can also be directed, meaning they have a specific start and end point, or undirected, meaning they have no specific start or end point.
Vertices/Nodes and Edges/Node Pairs: The nodes in a graph are also called vertices, and the edges are the connections between them. Edges are often represented as pairs of nodes, such as (u, v) to indicate an edge between vertices u and v.
Directed/Undirected Graph: In a directed graph, edges have a direction and can only be traversed in one direction. In an undirected graph, edges have no direction and can be traversed in either direction.
Comparison: Nodes are compared by their value.
Insertion: Nodes and edges are inserted by adding them to the graph as new vertices or edges, respectively.
Deletion: Nodes and edges are deleted by removing them from the graph along with any connections to other vertices or edges.
Indexing: Nodes are not accessed by index in a graph.
Hierarchy: Graphs have a hierarchical structure.

Indexing of Data Structures

Description: The method used to access individual elements or nodes within a data structure.
Comparison: Elements or nodes within a data structure are compared by their value.

Insertion: New elements or nodes are inserted at specific positions within the data structure based on their value or other relevant factors.
Deletion: Elements or nodes are deleted from the data structure by removing them from their specific positions.
Indexing: Elements or nodes within the data structure can be accessed by their index or position within the structure.
Hierarchy: The hierarchy of the data structure can impact the indexing method used. For example, linear data structures like arrays and linked lists are often accessed by index, while hierarchical data structures like trees and graphs may be accessed using traversal methods.