BACS2063 Data Structures and Algorithms

This is a note for subject BACS2063 that has been compiled and arranged by a salted fish (?

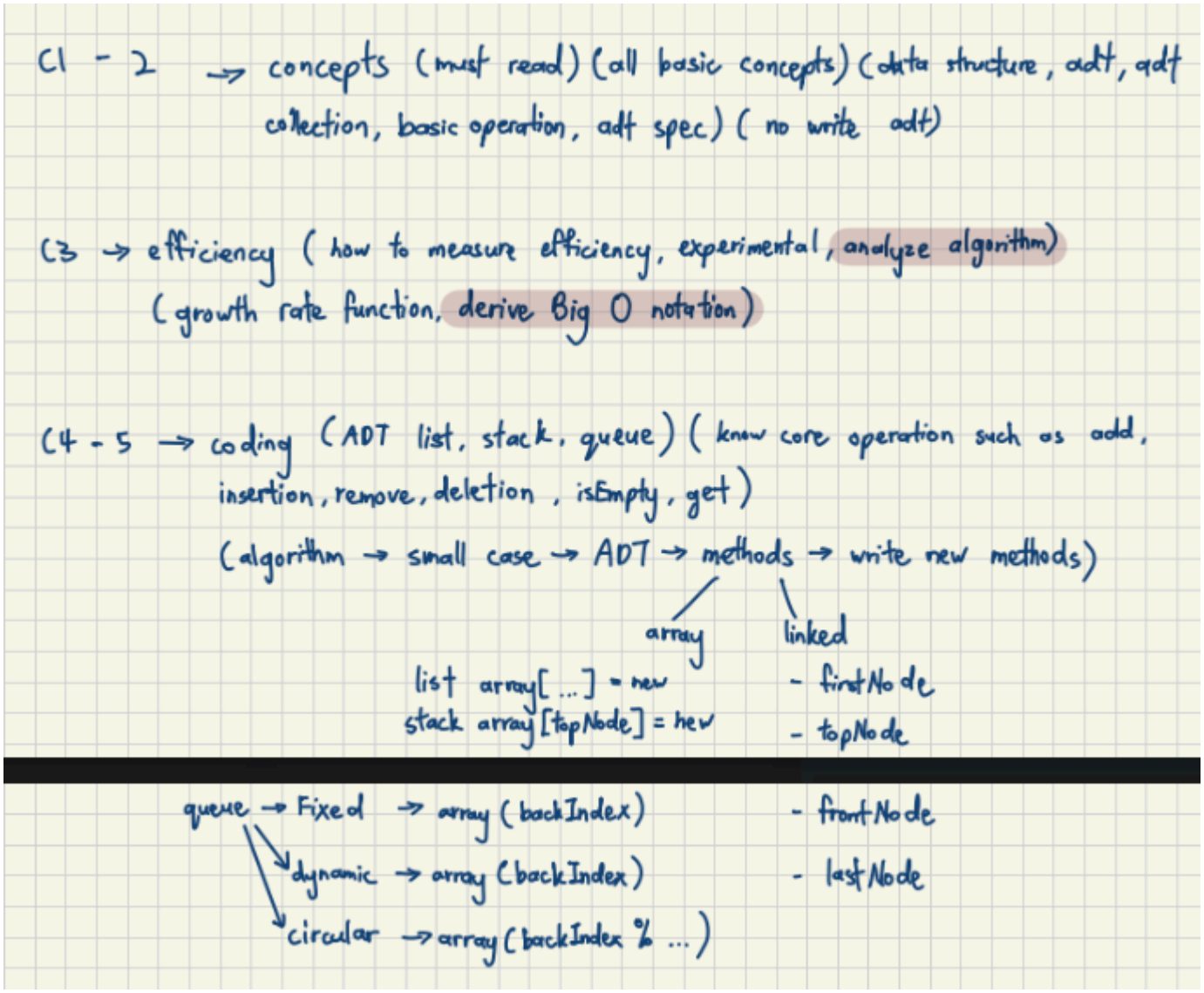
In addition to the lecture slide content, the notes also include internet references or personal additions made by the compiler.

Please note that while every effort has been made to ensure accuracy, this note is intended for personal use and may not fully represent the entirety of the subject matter. Users are encouraged to consult primary sources and official course materials for comprehensive understanding and verification🙂.

Table of Contents

Midterm

Week8 C1-C5



C1: Applications of Abstract Data Types (ADTs)

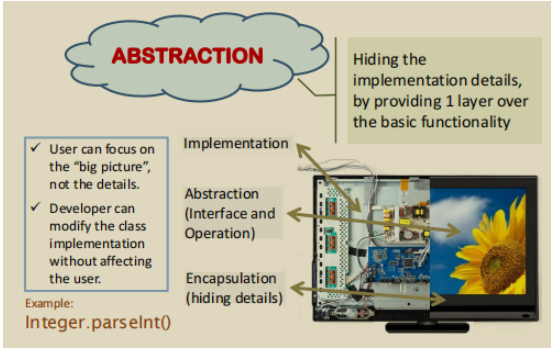
Recap of Programming Paradigms, Terminology

Procedural Paradigm:	Object-Oriented Paradigm:	Main differences
Came first. Focuses on procedures or functions. Data and functions are separate.	Introduced later. Centers around objects, which combine data and methods. Emphasizes reusability and modularity.	Data Handling: Procedural separates data; object-oriented encapsulates it within objects. Modularity: Object-oriented code is more modular. Reusability: Object-oriented supports reusability through inheritance.

process of simplifying complex systems by focusing on essential features and hiding the intricate details. It allows for the creation of models that are easier to understand and work with.

For example, if we abstract the concept of Takoyaki, we could think of it as a "FoodItem" object. This abstraction would emphasize the essential features of Takoyaki, such as it being a Japanese snack made of batter and octopus.

The detailed recipe and cooking process would be hidden behind this simple representation, allowing us to discuss and use Takoyaki in a general sense without needing to know the full complexity of its preparation.



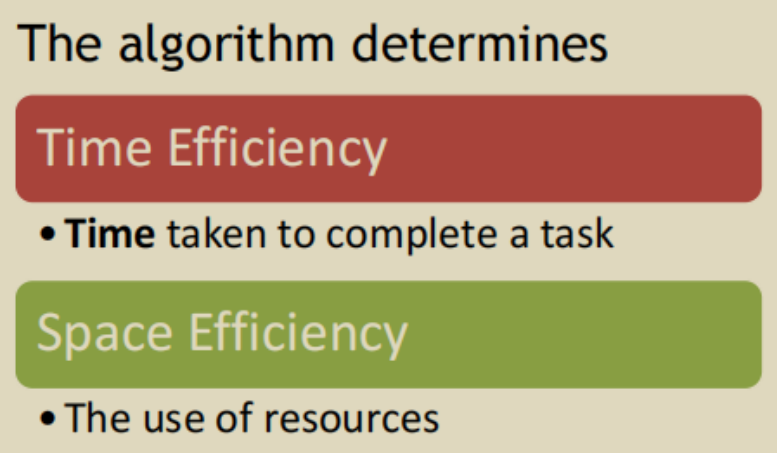
Design Principles(3)

🟩Encapsulation hides the complexity behind a simple interface	Modularity organizes a program into manageable parts	🟩...Abstraction
Imagine Felix as an instance of a Performer class. The class encapsulates Felix's attributes (like name, age, and singing ability) and methods (like perform() or interactWithFans()). As a fan, you don't need to know the details of how these methods work internally; you just enjoy the performance.	If you think of a concert as a software program, the concert could be modularized into different sections like opening act, main performance (where Stray Kids, including Felix, others), and encore. Each module has its own set of songs and activities, but they all work together to create the full concert experience.	simplifying complex reality by focusing on the essential features and ignoring the details that are not important for the current purpose

Abstract Data Type (ADT)s , Data Structure, Collection ADTs

ADTs provide a blueprint for what data and operations are needed	Data Structures provide a concrete implementation of that blueprint	Collection ADTs organize multiple data elements together in a specific way
like a "recipe" for Takoyaki. It tells you what ingredients (data) you need and the steps (operations) to make it, without specifying the exact cooking process. Recall: Benefits of ADTs • Reusability • Maintainability	is the physical "kitchen" where you make Takoyaki. It's the actual implementation of the ADT, detailing how the data is stored and manipulated.	like a "Takoyaki party," where you have a group of different Takoyaki recipes (a collection). Each recipe (object) follows the same interface (ADT) but may have different specific ingredients (data) and preparation methods (operations).

Algorithm



Collection ADTs (intro)

Definition: A collection is an abstract data type (ADT) that holds a group of objects.
Linear Collections: Collections that store entries in a linear sequence.
Examples: List, Stack, Queue.... The use of collection ADTs (C4-...) DSA_N_C1-5

Collection:	🟩Lists	🟩Stacks	🟩Queues
Definition	A linear collection where items have positions. Items can be added, removed, and searched without restriction.	A linear collection where objects follow the Last-In-First-Out (LIFO) principle. Items are added and removed in reverse order.	A linear collection following the First-In-First-Out (FIFO) principle. The first element added is the first to be removed.
Basic Operations	add(e): Insert element e at the end. remove(i): Remove the element at position i and return it. isEmpty(): Return true if the list is empty. get(i): Return the element at position i without removing it. clear(): Remove all elements from the list.	push(e): Add element e to the top of the stack. pop(): Remove and return the top element. isEmpty(): Return true if the stack is empty. peek(): Return the top element without removing it. clear(): Remove all elements from the stack.	enqueue(e): Add element e to the rear of the queue. dequeue(): Remove and return the front element. isEmpty(): Return true if the queue is empty. getFront(): Retrieve the front element without removing it. size(): Return the number of elements in the queue.
Applications DSA_N_...	Task Lists: To-do lists. High-Precision Arithmetic: Represent very long integer values	(Undo) Program Stack: Tracking method calls in programs.	Customer Service: Handling calls or service requests in order. Round-Robin Scheduling: Allocating CPU time slices to running applications.

Problem: Matching Brackets - Checking for Balanced (), [], {}

Traverse the expression to process each character

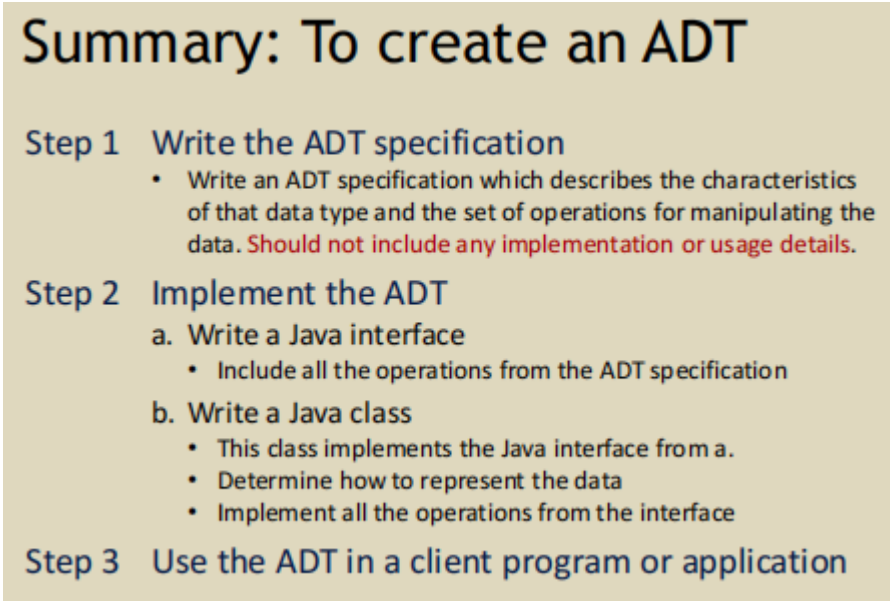
- If it is a left/open bracket, push it onto the stack.
- If it is a right/close bracket, pop a bracket from the stack and compare whether the two bracket types match

Infix, Prefix and Postfix Expressions

(a) Infix: a + b * c Postfix: a b c * + Prefix: + a * b c	(b) Infix: a * b / (c − d) Postfix: a b * c d - / Prefix: / * a b - c d	(c) Infix: a / b + (c − d) Postfix: a b / c d - + Prefix: + / a b - c d	(d) Infix: a / b + c − d Postfix: a b / c + d - Prefix: - + / a b c d

C2: ADTs

Abstract Data Types (ADTs)



🟩Benefits of Abstract Data Types (ADTs)

- **Achieving Reuse**
  - Abstraction: ADT specification. - Specifying the data type.
    Can focus on the abstract properties without worrying about how it is going to be implemented.
  - Encapsulation: ADT implementation.
- **Achieving Maintainability**
  - Encapsulation / Information Hiding: ADT implementation.

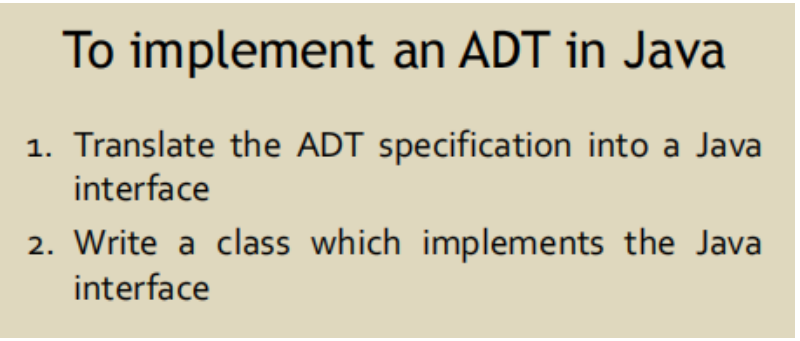
Encapsulation: Using the data type
Can use the components without knowing the implementation details.

🟩Write ADT Specifications -- 3 parts

An ADT specification is like a contract that defines what the ADT is and what operations can be performed on it. It includes:

ADT Title:	A name that describes the type, like Stack or Queue.	Example: DSA_N_C1
Description:	A high-level explanation of the ADT's purpose and the kind of data it will hold.	
Operations:	A list of methods that can be called on the ADT • Operation header: return type (if any), **operation name**, ***parameters (if any)*** • **Brief description** of what the operation does • ***Precondition (if any)*** • **Postcondition** ◦ What is returned by the operation (if any)	

Implement ADTs Using Java Interfaces and Classes



Example: DSA_N_C1-5

In Java, ADTs are implemented using interfaces and classes:

Interfaces: Define the methods that are part of the ADT without providing the implementation. This is like creating a blueprint for what the ADT should be able to do.	**Classes:** Implement the interfaces to provide the concrete behavior for the ADT. This is where you define how the data is stored and the operations are carried out.
Example: **public interface ADTInterface<T> {** boolean isEmpty(); T get(int index); // Other methods... **}**	Example: **public class ADTImplementation<T> implements** ADTInterface<T> **{** **private T[ ] data;** private int size; // Constructor, isEmpty(), get(), and other methods... **}**

By separating the specification (interface) from the implementation (class), Java allows for flexible and maintainable code design.

C3: Efficiency of Algorithms

Efficiency impacts overall performance, response time, and user satisfaction.

It's important to measure efficiency to compare different solutions to the same problem. We can visualize the results by plotting the performance of each run of the algorithm as a point with x-coordinate equal to the input size n and y- coordinate equal to the running time t.

X = n(Y)

Ways to Measure Efficiency of algorithms(2)

Experimental analysis can be influenced by hardware and software environments.

Theoretical analysis relies on identifying the worst-case scenario, which may not always reflect average performance.

Experimental Studies	🟩Analysis of Algorithms...
• Implement the algorithm. • Run it on various test inputs and record the time spent during each execution. • Use methods like System.currentTimeMillis() in Java for timing.	• A more theoretical approach to measure the complexity of an algorithm. • Consider space complexity (memory required) and time complexity (time to execute).

...Goal for Algorithm Analysis

➔ evaluate algorithms independently of hardware and software environments.

➔ analyze high-level descriptions without needing full implementation.

➔ account for all possible inputs.

Time Complexity	Counting Primitive Operations 计算原始操作
Focuses on the running time of an algorithm. Actual time computation is difficult, so time efficiency is estimated for best, average, and worst cases.	analyze the running time of an algorithm without performing experiments, analyze the algorithm by counting primitive operations. **Example of primitive operations:** • Assigning a value to a variable • Following an object reference • Performing an arithmetic operation • Comparing two numbers • Accessing a single element of an array by index • Calling a method

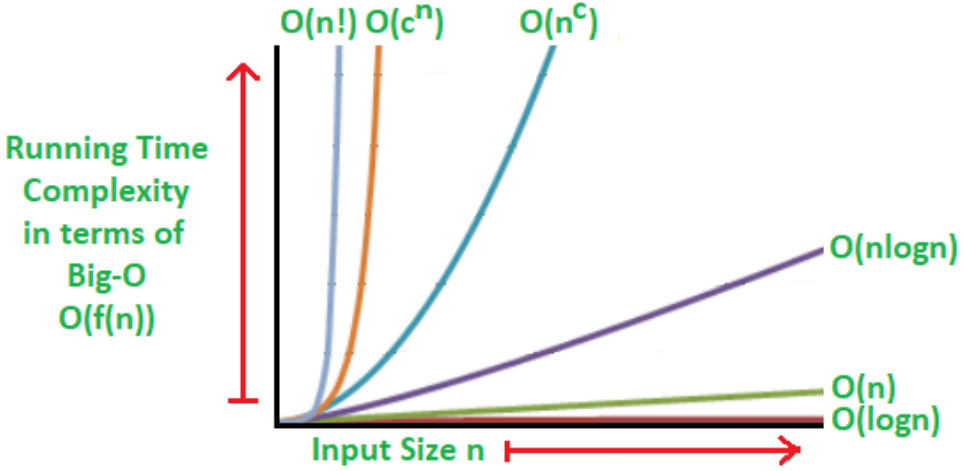
	• Returning from a method

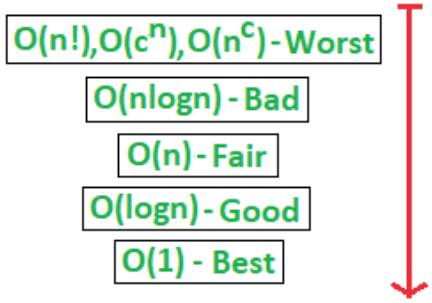
Measuring Operations as a Function of Input Size

associate a function f(n) with the algorithm - capture the order of growth of an algorithm's running time
f(n) will characterize the number of primitive operations that are performed as a function of the **input size n**

🟩Big O Notation - represent an algorithms efficiency/complexity

A mathematical notation to represent an algorithm's efficiency.
Expresses the upper bound of the growth rate of the number of operations.





Common Big O Classes

O(1)	• Constant time • Eg: Accessing an element in an array • Time complexity remains constant regardless of input size. • Clue: No loops or recursion; just simple arithmetic, assignment, or accessing array elements.	`// Accessing an element in an array` `int[] arr = {1, 2, 3, 4, 5};` `int element = arr[2];  // O(1) operation`
O(log n)	• Logarithmic time • Eg: Binary search • Time complexity increases logarithmically as input size grows. • Clue: A loop or recursive call that halves the input size in each step (e.g., left = mid + 1).	`// Binary search in a sorted array` `int binarySearch(int[] arr, int target) {` `  int left = 0, right = arr.length - 1;` `  while (left <= right) {` `  int mid = left + (right - left) / 2;` `  if (arr[mid] == target) {` `  return mid; // Found target` `  }` `  if (arr[mid] < target) {`

		`left = mid + 1; // Search right half` `} else {` `right = mid - 1; // Search left half` `}` `}` `return -1; // Target not found` `}`
O(n)	• Linear time • Eg: Linear search (sequential search) • Time complexity increases directly proportional to input size. • Clue: A single loop that iterates through the input from start to finish.	`// Finding the maximum element in an array` `int findMax(int[] arr) {` `int max = arr[0];` `for (int i = 1; i < arr.length; i++) {` `if (arr[i] > max) {` `max = arr[i];` `}` `}` `return max;` `}`
O(n log n)	• Linearithmic time • Eg: Merge sort, Heap sort • Time complexity increases slightly faster than linear time. • Clue: A divide-and-conquer approach combined with a linear operation (like merging or sorting).	`// Merge Sort algorithm` `void mergeSort(int[] arr, int left, int right) {` `if (left < right) {` `int mid = left + (right - left) / 2;` `mergeSort(arr, left, mid);  // Sort left half` `mergeSort(arr, mid + 1, right);  // Sort right half` `merge(arr, left, mid, right);  // Merge both halves` `}` `}...`https://prnt.sc/SI6fednFwOtv
O(n^2)	• Quadratic time • Eg: Bubble sort, Insertion sort • Time complexity increases quadratically as input size grows. • Clue: Two nested loops, each iterating over the input size.	`// Bubble Sort algorithm` `void bubbleSort(int[] arr) {` `int n = arr.length;` `for (int i = 0; i < n - 1; i++) {` `for (int j = 0; j < n - i - 1; j++) {` `if (arr[j] > arr[j + 1]) {` `// Swap arr[j] and arr[j + 1]` `int temp = arr[j];` `arr[j] = arr[j + 1];` `arr[j + 1] = temp;` `}` `}` `}` `}`
O(n^3)	• Cubic time • Eg: Matrix multiplication (naive) • Time complexity increases cubically as input size grows. • Clue: Three nested loops, each iterating over the input size.	`// Naive matrix multiplication` `int[][] multiplyMatrices(int[][] A, int[][] B) {` `int n = A.length;` `int[][] C = new int[n][n];` `for (int i = 0; i < n; i++) {` `for (int j = 0; j < n; j++) {` `C[i][j] = 0;`

		`` for (int k = 0; k < n; k++) { C[i][j] += A[i][k] * B[k][j]; } } } return C; } ```
O(2^n)	• Exponential time • Eg: Recursive Fibonacci, Subset sum • Time complexity doubles with each addition to the input size. • Clue: Recursive algorithms where the number of recursive calls grows exponentially with the input size.	``` // Recursive Fibonacci calculation int fibonacci(int n) { if (n <= 1) { return n; } return fibonacci(n - 1) + fibonacci(n - 2); } ```

growth rates indicated by their Big O classes to determine which is more efficient.

Growth Rate

Big O Notation	Growth Rate (as n increases)	Efficiency (for large n)
O(1)	No growth	**Most efficient**
O(log n)	Very slow growth	Very efficient
O(n)	Linear growth	Efficient
O(n log n)	Faster than linear but slower than quadratic	Moderately efficient
O(n^2)	Rapid growth (quadratic)	Less efficient
O(n^3)	Very rapid growth (cubic)	Inefficient
O(2^n)	Extremely rapid growth (exponential)	**Least efficient**

Comparing Specific Examples

Let's say we want to compare O(n) and O(n^2) to determine which is more efficient:

For small n

n: The difference between O(n) and O(n^2) might not be significant.

For large n

n: The O(n^2) algorithm will take much more time to complete compared to O(n), as quadratic growth is much faster than linear growth.

Let's consider a few different Big O notations and compare their growth rates by substituting values for n:

n = 10	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(n^3)	O(2^n)
Time Steps	1	3	10	30	100	1,000	1,024

n = 100	O(1)	O(log n)	O(n)	O(n log n)	O(n^2)	O(n^3)	O(2^n)
Time Steps	1	7	100	700	10,000	1,000,000	1.27E+30

Conclusion:

Lower Big O notations (like O(1), O(log n), O(n)) are generally more efficient, especially for large input sizes.

Higher Big O notations (like O(n^2), O(n^3), O(2^n)) indicate less efficiency, particularly as the input size increases.

The best choice of algorithm depends on the problem size and whether the extra computational cost is justified by other factors (like simplicity or accuracy).

How to calculate Big O🟩

c) Big O notation is used to determine the time efficiency of an algorithm. Calculate the Big O for the following algorithms in the Figure 3.

Algorithm A	Algorithm B	Algorithm C
sum = 0 for i = 1 to n sum = sum + i	sum = 0 for i = 1 to n { for j = 1 to i sum = sum + 1 }	sum = n * (n + 1) / 2

Figure 3: Algorithms A, B and C

(5 marks)

1. Understanding Big O Notation

Big O notation describes the upper bound of an algorithm's runtime, which tells us how the runtime grows as the input size (often denoted as `n`) increases. It focuses on the most significant factors and ignores smaller constants and lower-order terms.

3. Examples Using the Provided Algorithms (062023-Q2c)

Algorithm A

```
sum = 0
for i = 1 to n
    sum = sum + i
```

- **Analysis**:
  - The loop runs from 1 to `n` → O(n).
  - Inside the loop, `sum = sum + i` is an O(1) operation.
- **Big O**: The overall complexity is dominated by the loop, so it's O(n).

Algorithm B

```
sum = 0
for i = 1 to n
    for j = 1 to i
```

2. How to Calculate Big O

When analyzing an algorithm, consider these steps:

Step 1: Identify the Basic Operations

- A basic operation is something that takes constant time, like a simple addition or comparison. This is considered O(1).

Step 2: Look at Loops

- **Single Loop:** If an algorithm has a loop that runs from 1 to n, that loop contributes O(n) because it performs the operation n times.
- **Nested Loops:** If there's a loop inside another loop, you multiply their complexities. For example,
  two nested loops where each runs `n` times results in $O(n) \times O(n) = O(n^2)$.

Step 3: Ignore Constants and Lower-Order Terms

- When calculating Big O, we ignore constants (like 2, 5, etc.) and lower-order terms because they become insignificant as n grows large.
  - For example, $O(2n)$ simplifies to $O(n)$, and $O(n^2 + n)$ simplifies to $O(n^2)$.

```
        sum = sum + 1
```

- **Analysis**:
  - The outer loop runs n times → O(n)).
  - The inner loop runs `i` times, and since `i` varies from 1 to `n`, you sum up all these iterations: $1 + 2 + 3 + ... + n$, which equals $\frac{n(n+1)}{2}$.
  - The sum $\frac{n(n+1)}{2}$ grows quadratically → $O(n^2)$.
- **Big O**: The overall complexity is O(n^2).

Algorithm C

```
sum = n * (n + 1) / 2
```

- **Analysis**:
  - This is a direct mathematical formula, with no loops.
  - Each operation is a simple arithmetic calculation → O(1).
- **Big O**: The overall complexity is O(1).

4. General Guidelines

- **Linear Time O(n)**: If the runtime increases proportionally with the input size, it's linear.
- **Quadratic Time O(n^2)**: If the runtime grows with the square of the input size, it's quadratic, often due to nested loops.
- **Constant Time O(1)**: If the runtime does not depend on the input size, it's constant.

C4: Array Implementations of Collection ADTs

Collection ADTs ?

Collection ADTs

- An ADT that can store a collection of objects.
- A *linear collection* is a collection that stores its entries in a linear sequence, *e.g.*:
  - List
  - Stack
  - Queue

  They differ in the restrictions they place on how these entries may be added, removed, or accessed

Generic Types

A generic type must be a reference type, not a primitive type.

Used in Java interfaces and classes that implement collection ADTs to specify and constrain the type of objects being stored in the collection.

Syntax	Using Generic Types
Defining a Generic Type:	Supplying a Type Argument:
The interface or class name is followed by an identifier enclosed in angle brackets:	When using the class, you supply an actual type argument to replace T. For example:
public interface ListInterface<T>	ListInterface<String> taskList;
The identifier T can be any identifier but is usually a single capital letter representing the data type within the class definition.	Now, whenever T appears as a data type in the definition of ListInterface, String will be used.

https://www.geeksforgeeks.org/generics-in-java/

Collection ADTs recap: DSA_N_C1-5

Lists: A list is like a concert setlist. It can have items added or removed in any order, just like songs can be added or removed from a setlist.	Stacks: A stack is like a pile of plates, where you can only take the top plate off or add a new plate to the top. It follows the Last-In-First-Out (LIFO) principle.	Queues: A queue is like a line of fans waiting to meet Felix. The first person in line is the first to meet him, following the First-In-First-Out (FIFO) principle.
Operations: Add, remove, replace, get entry, count entries, check if empty/full.	Operations: Push (add to top), pop (remove from top), peek (look at the top), isEmpty, clear.	Operations: Enqueue (add to back), dequeue (remove from front), getFront, isEmpty, clear.

Array Implementations of ADTs

Use an array to store list items. Adding an item might require expanding the array if it's full.	Use an array where the top of the stack is either the end or the beginning of the array, depending on the implementation.	Use an array with two pointers, one for the front and one for the back of the queue.

Strengths and Weaknesses of Array Implementations

Strengths:	Weaknesses:
Fast access to any element since arrays provide direct access to elements via indexing. Efficient implementation for adding elements to the end of a list or pushing to a stack.	Inefficient for adding or removing elements in the middle of a list, as it requires shifting elements. Fixed size limits flexibility; dynamic resizing (expanding the array) is costly.

Efficiency Analysis

Time Complexity: Operations like adding to the end of a list or pushing to a stack are O(1). However, adding to the beginning or middle of a list, or expanding an array, can be O(n).

Space Complexity: Arrays may waste space if their size is much larger than the number of elements currently stored.

Iterators

Iterators allow for traversing through the elements of a collection one by one, which is useful for operations like displaying all elements or searching for a specific element.

Sample code for C4🥲

List

- add(T element): Adds an element to the end of the list.
- remove(int index): Removes the element at the specified index.
- replace(int index, T element): Replaces the element at the specified index with a new element.
- getEntry(int index): Returns the element at the specified index.
- countEntries(): Returns the number of elements in the list.
- isEmpty(): Checks if the list is empty.
- isFull(): (Only for FixedSizeArrayList) Checks if the list has reached its maximum capacity.
- makeRoom(int newPosition): Shifts elements one position to the right, starting from the last element down to the position where a new element will be inserted. This creates space at the specified position for the new element.
- removeGap(int givenPosition): Shifts elements one position to the left, starting from the position after the one that was removed, up to the last element in the list. This closes the gap left by the removed element.

Fixed-Size Array List

```
public class FixedSizeArrayList<T> implements ListInterface<T> {
    private Object[] array;
    private int size;
    private final int capacity;

    public FixedSizeArrayList(int capacity) {
        this.capacity = capacity;
        array = new Object[capacity];
        size = 0;
    }

    public void add(T element) {
        if (size < capacity) {
            array[size++] = element;
        } else {
            throw new IllegalStateException("List is full");
        }
    }

    public T remove(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        T removed = (T) array[index];
        for (int i = index; i < size - 1; i++) {
            array[i] = array[i + 1];
        }
        array[--size] = null; // Help GC
        return removed;
    }

    public T replace(int index, T element) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        T oldElement = (T) array[index];
        array[index] = element;
        return oldElement;
    }

    public T getEntry(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        return (T) array[index];
    }
    public int countEntries() {
        return size;
    }
    public boolean isEmpty() {
        return size == 0;
```

Dynamic Array List

```
public class DynamicArrayList<T> implements ListInterface{
    private Object[] array;
    private int size;

    public DynamicArrayList() {
        array = new Object[10]; // Default capacity
        size = 0;
    }

    public void add(T element) {
        ensureCapacity();
        array[size++] = element;
    }

    private void ensureCapacity() {
        if (size == array.length) {
            Object[] newArray = new Object[size * 2];
            System.arraycopy(array, 0, newArray, 0, size);
            array = newArray;
        }
    }

...
```

```
    }
    public boolean isFull() {
        return size == capacity;
    }
}
```

Stack

- push(T element): Adds an element to the top of the stack.
- pop(): Removes and returns the top element of the stack. Throws EmptyStackException if the stack is empty.
- peek(): Returns the top element without removing it. Throws EmptyStackException if the stack is empty.
- isEmpty(): Returns true if the stack is empty, false otherwise.
- clear(): Resets the stack to an empty state. For the dynamic array, this also optionally involves clearing references to help with garbage collection and can potentially reduce the size of the array.
- The ensureCapacity() method in the DynamicStack is called by push() to check if the stack needs to grow. If the top index is at the last position of the array, a new, larger array is created, and the elements are copied over. This ensures that there is always space for new elements.

Fixed-Size Array Stack

```
public class FixedSizeStack<T> {
    private T[] stackArray;
    private int top;
    private final int capacity;

    public FixedSizeStack(int capacity) {
        this.capacity = capacity;
        stackArray = (T[]) new Object[capacity];
        top = -1;
    }

    public void push(T element) {
        if (top < capacity - 1) {
            stackArray[++top] = element;
        } else {
            throw new IllegalStateException("Stack is full");
        }
    }

    public T pop() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return stackArray[top--];
    }

    public T peek() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
```

Dynamic Array Stack

```
public class DynamicStack<T> implements StackInterface<T> {
  private T[] stackArray;
  private int topIndex; // index of top entry
  private static final int DEFAULT_CAPACITY = 50;

    public DynamicStack() {
        this(DEFAULT_CAPACITY);
    }


    public DynamicStack(int initialCapacity) {
        array = (T[]) new Object[initialCapacity];
        topIndex = -1;
    }


    public void push(T element) {
        ensureCapacity();
        stackArray[++top] = element;
    }

    private void ensureCapacity() {
        if (top >= stackArray.length - 1) {
            T[] newArray = (T[]) new Object[stackArray.length * 2];
            System.arraycopy(stackArray, 0, newArray, 0,
stackArray.length);
            stackArray = newArray;
        }
    }
```

```
        return stackArray[top];
    }

    public boolean isEmpty() {
        return top == -1;
    }

    public void clear() {
        top = -1;
        // Optionally, clear the reference to the array to help with
garbage collection
        // stackArray = null;
    }
}
```

```
    public T pop() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        T element = stackArray[top];
        stackArray[top--] = null; // Help GC
        return element;
    }

    public T peek() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return stackArray[top];
    }

    public boolean isEmpty() {
        return top == -1;
    }

    public void clear() {
        top = -1;
        // Clear the array and help with garbage collection
        for (int i = 0; i <= top; i++) {
            stackArray[i] = null;
        }
        // Optionally, reduce the size of the array if it's too large
        // int newCapacity = ...; // Calculate new capacity
        // T[] newArray = (T[]) new Object[newCapacity];
        // System.arraycopy(stackArray, 0, newArray, 0, size);
        // stackArray = newArray;
    }
}
```

Queue

- enqueue(T element): Adds an element to the rear of the queue.
- dequeue(): Removes and returns the element from the front of the queue. Throws NoSuchElementException if the queue is empty.
- getFront(): Returns the element at the front without removing it. Throws NoSuchElementException if the queue is empty.
- isEmpty(): Returns true if the queue is empty, false otherwise.
- clear(): Resets the queue to an empty state. For the dynamic array, this also optionally involves clearing references to help with garbage collection.
- The ensureCapacity() method in the DynamicQueue is called by enqueue() to check if the queue needs to grow. If adding a new element would exceed the current array's capacity, a new, larger array is created, and the elements are copied over. This ensures that there is always space for new elements. The resizing process also takes care of wrapping the front and rear indices to simulate a circular queue within the linear array.

Fixed-Size Array Queue	Dynamic Array Queue (Circular)
`public class FixedSizeQueue<T> {`	`public class DynamicQueue<T> implements QueueInterface<T>{`

```
    private T[] queueArray;
    private int front;
    private int rear;
    private final int capacity;

    public FixedSizeQueue(int capacity) {
        this.capacity = capacity;
        queueArray = (T[]) new Object[capacity];
        front = 0;
        rear = -1;
    }

    public void enqueue(T element) {
        if (isFull()) {
            throw new IllegalStateException("Queue is full");
        }
        rear = (rear + 1) % capacity;
        queueArray[rear] = element;
    }

    public T dequeue() {
        if (isEmpty()) {
            throw new NoSuchElementException("Queue is empty");
        }
        T element = queueArray[front];
        front = (front + 1) % capacity;
        return element;
    }

    public T getFront() {
        if (isEmpty()) {
            throw new NoSuchElementException("Queue is empty");
        }
        return queueArray[front];
    }

    public boolean isEmpty() {
        return front == rear;
    }

    public boolean isFull() {
        return (rear + 1) % capacity == front;
    }

    public void clear() {
        front = 0;
        rear = -1;
        // Optionally clear the array to help with garbage collection
        for (int i = 0; i < capacity; i++) {
            queueArray[i] = null;
        }
    }
}
```

```
    private T[] queueArray;
    private int front;
    private int rear;
    private int size;

    public DynamicQueue() {
        queueArray = (T[]) new Object[10]; // Default capacity
        front = 0;
        rear = -1;
        size = 0;
    }

    public void enqueue(T element) {
        ensureCapacity();
        if (rear == -1) {
            rear = 0;
        }
        rear = (rear + 1) % queueArray.length;
        queueArray[rear] = element;
        size++;
    }

    private void ensureCapacity() {
        if (size >= queueArray.length) {
            T[] newArray = (T[]) new Object[queueArray.length * 2];
            int k = 0;
            for (int i = front; i != rear + 1; i = (i + 1) %
queueArray.length) {
                newArray[k++] = queueArray[i];
            }
            queueArray = newArray;
            front = 0;
            rear = k - 1;
        }
    }

    public T dequeue() {
        if (isEmpty()) {
            throw new NoSuchElementException("Queue is empty");
        }
        T element = queueArray[front];
        front = (front + 1) % queueArray.length;
        size--;
        // Optionally shrink the array if it's too large and the queue
is not full
        return element;
    }

    public T getFront() {
        if (isEmpty()) {
            throw new NoSuchElementException("Queue is empty");
        }
        return queueArray[front];
    }
```

```
    public boolean isEmpty() {
        return size == 0;
    }

    public void clear() {
        front = 0;
        rear = -1;
        size = 0;
        // Optionally clear the array to help with garbage collection
        for (int i = 0; i < queueArray.length; i++) {
            queueArray[i] = null;
        }
    }
}
```

C5: Linked Implementations of Collection ADTs

Linked Structures Overview

Limitations of Arrays: Fixed size and the overhead of shifting elements during add and remove operations.

Linked Structures: Offer flexibility by allocating space for each entry as needed and deallocate space when an entry is removed.

Examples:

Felix and Performances: If Felix's performances were managed by a list, each performance could be a node in a linked list, allowing for flexible scheduling (additions or removals) without affecting the entire list.

Takoyaki Orders: A queue could be used to manage orders of Takoyaki, where each order is a node. New orders are added to the back (enqueue), and completed orders are removed from the front (dequeue), ensuring that the first order placed is the first to be served.

Linked Lists

A linked list is a collection of nodes where each node contains data and a reference (link) to the next node.

Nodes: Comprise a data part and a link part, with the link pointing to the next node in the sequence.

Linked Implementation of ADTs

List	Stack	Queue
Inner Class Node: Used to encapsulate the node structure within the list implementation. Traversing a Linked List: Involves moving from the head node to the desired node by following the links. Adding to the List: Can be done at the beginning, end, or within the list by adjusting the links of the nodes.	Stack Characteristics: LIFO (Last-In-First-Out) behavior. Operations: Push (add to top), pop (remove from top), and peek (view the top without removing).	Queue Characteristics: FIFO (First-In-First-Out) behavior. Operations: Enqueue (add to back), dequeue (remove from front), and getFront (view the front without removing).

Strengths and Weaknesses of Array Implementations

Strengths:	Weaknesses:
Dynamic size adjustment without the need for shifting elements. Efficient use of memory as nodes are created and destroyed as needed.	Slower access time compared to arrays since nodes must be traversed. Additional memory overhead for storing links.

Variations of Linked Structures

Linear vs. Circular: Linear lists have a terminal node with a null link, while circular lists connect the last node back to the first.

Singly vs. Doubly Linked Lists: Singly linked lists have nodes that reference the next node only, while doubly linked lists have nodes that reference both the next and previous nodes.

Iterators in Linked Implementation

Iterators provide a way to traverse linked collections, offering methods to check for the next entry, advance to the next entry, and return the current entry.

Sample code for C5🥲

List

- In the LinkedList class, the add method appends elements to the end of the list, remove deletes an element at a given index, replace substitutes an element at a given index with a new value, get retrieves an element at a given index, isEmpty checks if the list is empty, and size returns the number of elements in the list. The clear method removes all elements from the list.
- The CircularLinkedList class extends LinkedList and adds the functionality to make the list circular by pointing the last node to the first node. The add and remove methods are overridden to maintain the circular property of the list. When removing the only element in a circular list, the list becomes empty. When removing the last node, the second-to-last node's next pointer is updated to point to the head of the list, maintaining the circular linkage.
- Please note that this implementation of a circular linked list assumes that the add method will be used to add elements, and the list will not be modified in a way that disrupts the circular property (e.g., directly manipulating next pointers). Additional error checking and handling can be added as needed for a more robust implementation.

Linked

```
public class LinkedList<T> {
    private Node<T> firstNode; // Head node of the list
    private int size; // Current size of the list

    // Node inner class
    private static class Node<T> {
        T data;
        Node<T> next;

        Node(T data) {
            this.data = data;
            this.next = null;
        }
    }

    // Add element to the end of the list
    public void add(T element) {
        Node<T> newNode = new Node<>(element);
        if (fistNode == null) {
```

Circular Linked

```
public class CircularLinkedList<T> extends LinkedList<T> {
    // Make the last node point to the first node to complete the
circle
    public void makeCircular() {
        if (head != null) {
            Node<T> current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = head;
        }
    }

    // Add element to the end of the list and complete the circle
    @Override
    public void add(T element) {
        super.add(element);
        if (size == 1) { // If there's only one element, make it
circular
```

```
            firstNode= newNode;
        } else {
            Node<T> currentNode = fistNote;
            while (currentNode.next != null) {
                currentNode = currentNode.next;
            }//find the last node
            currentNode.next = newNode;  //newNode link to the .next
        }
        size++;
    }

    // Remove element at a specific position
    public T remove(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        Node<T> currentNode = firstNode;
        if (index == 0) {
            firstNode = currentNode.next;
        } else {
            for (int i = 0; i < index - 1; i++) {
                currentNode = currentNode.next;
            }
            Node<T> toRemove = currentNode.next;
            currentNode.next = toRemove.next;
        }
        size--;
        return toRemove.data;
    }

    // Replace element at a specific position
    public T replace(int index, T element) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        Node<T> current = head;
        for (int i = 0; i < index; i++) {
            current = current.next;
        }
        T oldData = current.data;
        current.data = element;
        return oldData;
    }

    // Get element at a specific position
    public T get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        Node<T> current = head;
        for (int i = 0; i < index; i++) {
            current = current.next;
        }
        return current.data;
```

```
            head.next = head;
        } else if (size > 1) {
            Node<T> last = head;
            // Traverse to the current last node
            for (int i = 0; i < size - 2; i++) {
                last = last.next;
            }
            // Make the new last node point to the head
            last.next = head;
        }
    }

    // Remove element at a specific position in a circular list
    @Override
    public T remove(int index) {
        if (isEmpty()) {
            throw new NoSuchElementException("List is empty");
        }
        if (size == 1) {
            T data = head.data;
            head = null;
            size = 0;
            return data;
        } else {
            // Adjust the last node's next to skip the node to be
removed
            Node<T> current = head;
            if (index == 0) {
                Node<T> second = head.next;
                second.next = head.next.next; // Make the second node
point to the third node
                head = second;
            } else {
                for (int i = 0; i < index - 1; i++) {
                    current = current.next;
                }
                Node<T> toRemove = current.next;
                current.next = toRemove.next;
                if (index == size - 1) { // If removing the last node,
update the last reference
                    Node<T> newLast = head;
                    for (int i = 0; i < size - 2; i++) {
                        newLast = newLast.next;
                    }
                    newLast.next = head;
                }
            }
            size--;
            return toRemove.data;
        }
    }

    // Additional methods for CircularLinkedList can be overridden or
extended as needed.
}
```

```
    }

    // Check if the list is empty
    public boolean isEmpty() {
        return size == 0;
    }

    // Get the number of elements in the list
    public int size() {
        return size;
    }

    // Clear all elements from the list
    public void clear() {
        head = null;
        size = 0;
    }
}
```

Stack

- In the LinkedStack class, the push method adds elements to the top of the stack, pop removes and returns the top element, peek retrieves the top element without removing it, isEmpty checks if the stack is empty, and clear removes all elements from the stack. The size method returns the current number of elements in the stack.
- The CircularLinkedStack class extends LinkedStack and modifies the push and pop methods to maintain the circular property of the stack. After pushing a new element, the new top node is made to point to the original top node to create a circular link. When popping an element, the top is adjusted to point to the next node in the circular sequence.
- Please note that the circular linked stack implementation provided here is a basic example and may require additional logic to handle certain edge cases or to ensure the circular property is maintained correctly in all operations. The size() method from the superclass is used to get the current size of the stack, which can be helpful in understanding the stack's state after modifications.

Linked

```
public class LinkedStack<T> {
    private Node<T> top; // Top node of the stack
    private int size; // Current size of the stack

    // Node inner class
    private static class Node<T> {
        T data;
        Node<T> next;

        Node(T data) {
            this.data = data;
            this.next = null;
        }
    }

    // Push element to the top of the stack
```

Circular Linked

```
public class CircularLinkedStack<T> extends LinkedStack<T> {
    public CircularLinkedStack() {
        super();
    }

    // Push element to the top of the circular stack
    @Override
    public void push(T element) {
        super.push(element);
        // Make the new top point to the original top to maintain
circularity
        if (size > 1) {
            Node<T> newTop = (Node<T>) top.next;
            newTop.next = top;
        }
    }
```

```
    public void push(T element) {
        Node<T> newNode = new Node<>(element);
        newNode.next = top;
        top = newNode;
        size++;
    }

    // Pop element from the top of the stack
    public T pop() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        T element = top.data;
        top = top.next;
        size--;
        return element;
    }

    // Peek at the top element of the stack
    public T peek() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return top.data;
    }

    // Check if the stack is empty
    public boolean isEmpty() {
        return size == 0;
    }

    // Clear all elements from the stack
    public void clear() {
        top = null;
        size = 0;
    }

    // Get the current size of the stack
    public int size() {
        return size;
    }
}
```

```
    // Pop element from the top of the circular stack
    @Override
    public T pop() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        T element = super.pop();
        // Adjust the top to point to the next node after the popped
element
        if (size() > 0) {
            Node<T> newTop = top;
            while (newTop.next != null && newTop.next != top) {
                newTop = newTop.next;
            }
            top = newTop;
        }
        return element;
    }

    // Additional methods for CircularLinkedStack can be overridden or
extended as needed.
}
```

Queue

- In the LinkedQueue class, the enqueue method adds elements to the rear of the queue, dequeue removes and returns the element from the front, getFront retrieves the front element without removing it, isEmpty checks if the queue is empty, and clear removes all elements from the queue. The size method returns the current number of elements in the queue.
- The CircularLinkedQueue class extends LinkedQueue and modifies the enqueue and dequeue methods to maintain the circular property of the queue. After enqueueing a new element, the rear node is made to point to the front node to create a circular link. When dequeuing an element, if the queue becomes empty, the rear pointer is also set to null.

- Please note that this implementation of a circular linked queue assumes that the enqueue and dequeue methods are used to modify the queue and that the circular property is maintained accordingly. Additional error checking and handling can be added as needed for a more robust implementation.

Linked

```
public class LinkedQueue<T> {
    private Node<T> front; // Front node of the queue
    private Node<T> rear;  // Rear node of the queue
    private int size;       // Current size of the queue

    // Node inner class
    private static class Node<T> {
        T data;
        Node<T> next;

        Node(T data) {
            this.data = data;
            this.next = null;
        }
    }

    // Enqueue element to the rear of the queue
    public void enqueue(T element) {
        Node<T> newNode = new Node<>(element);
        if (rear == null) {
            front = rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }
        size++;
    }

    // Dequeue element from the front of the queue
    public T dequeue() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
        T element = front.data;
        front = front.next;
        if (front == null) { // If the queue is now empty
            rear = null;
        }
        size--;
        return element;
    }

    // Peek at the front element of the queue
    public T getFront() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
```

Circular Linked

```
public class CircularLinkedQueue<T> extends LinkedQueue<T> {
    // Enqueue element to the rear of the circular queue
    @Override
    public void enqueue(T element) {
        super.enqueue(element);
        // Make the rear node point to the front node to maintain
circularity
        if (size == 1) { // If there's only one element, make it
circular
            rear.next = front;
        } else if (size > 1) {
            rear.next = front; // Make the new rear point to the front
        }
    }

    // Dequeue element from the front of the circular queue
    @Override
    public T dequeue() {
        if (isEmpty()) {
            throw new NoSuchElementException();
        }
        T element = super.dequeue();
        // If the queue is empty after dequeue, update the rear pointer
        if (size == 0) {
            rear = null;
        }
        return element;
    }

    // Additional methods for CircularLinkedQueue can be overridden or
extended as needed.
}
```

```
        return front.data;
    }

    // Check if the queue is empty
    public boolean isEmpty() {
        return size == 0;
    }

    // Clear all elements from the queue
    public void clear() {
        front = rear = null;
        size = 0;
    }

    // Get the current size of the queue
    public int size() {
        return size;
    }
}
```

C6:

C10: Hashing

Hash Code

- A hash code is an **integer value** generated from an object.
- It acts as a **unique identifier** or index when storing data in structures like hash tables.
- Java's Object class provides a method hashCode() which returns a default hash code based on the object's memory address.

ADT Dictionary

- A Dictionary, also called a map or associative array, is a **data structure that stores entries as key-value pairs**. The key is used to search for, insert, or delete associated values.

Operation:
- Add: Insert a new key-value pair.
- Remove: Delete a pair based on the key.
- Retrieve/Get: Fetch the value associated with a given key.
- Contains: Check if a key exists in the dictionary.

Applications:
- A Directory of Telephone Numbers
- The Frequency of Words
- A Concordance of Words

Efficiency:
- Enable fast lookups and updates.
- The efficiency of dictionary operations is crucial in applications that require fast searching
- Example: databases.

Hashing

🟩Techniques for determining **where** an item in a data structure **is indexed or stored without searching** for it.

🟩Hash Function
- Mapping entry search keywords to locations containing items
- Returns the hash index of an entry in the hash table.
- **Characteristics of Good Hash Function**
  - Collision Minimization
    - Minimize the number of collision

Hashing Process

Step 1:
Generate Hash Code - Convert the search key (e.g., a string or number) into an integer using a hash function.

- Uniform Distribution
  - Evenly distribute the key values in the hash table to avoid clustering or overloading certain regions.
- Fast Computation
  - Functions are called frequently, so must be able to be computed quickly. Complex hash function will slow down dictionary operations.
- Key Types
  - Functions can be used for both primitive types (eg. integers) and object instances.

Perfect Hash Function:
- **Each** unique search keyword is mapped to a unique position in the table, so no collisions occur.
- But in practice, collisions still occur. So some technique is needed to handle them.

Step 2:
Compress Hash Code - The hash code is then "compressed" into a valid index for the hash table (typically by using modulo operation to fit the table's size).

🟩🟩Collision Resolution Techniques

Open Addressing		
Linear Probing	- If collision occurs at hashTable[k], look successively at location k + 1, k + 2, ... - When probing reaches the end of the table, it continues - at the table's beginning Example: Following the working step, the 23 have collided with 67. So we need to solve the collision. **Working step** | **Resulting hash table**	✅ - Easy implement - Can reach every location in the hash table. It guarantees the success of the add operation when the hash table is not full. ❌ - Prone to principal clustering (主聚类) increases probe length and search time when the table fills up and lead to hash

Key	h1
65	10
42	9
85	8
67	1
23	1
51	7

0	1	2	3	4	5	6	7	8	9	10
	67					23	51	85	42	65

Actually, before solving the collision, the 23 now is at index 1. So now we set k as 1. Then, we look successively at k + 1, k + 2….

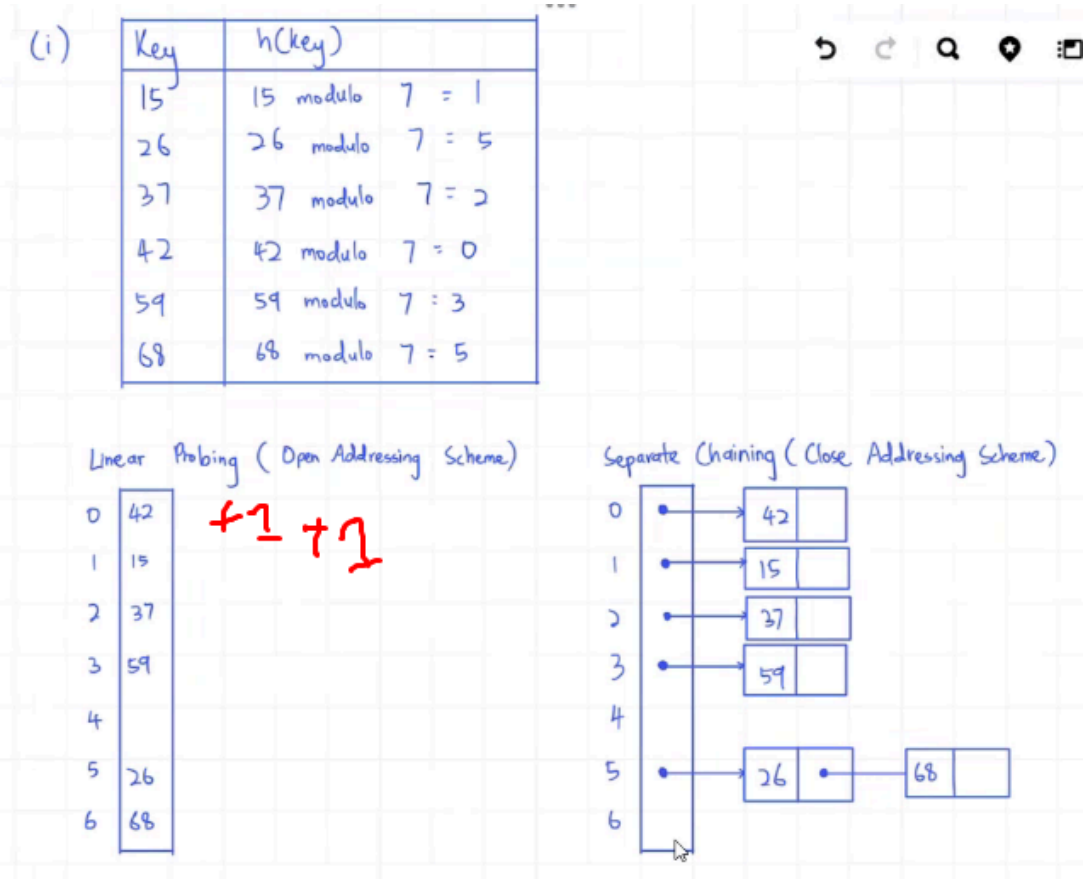
k + 1 = walk one step become index 2, if the location are empty, then direct store the key at the location.

If after one step the location are not empty, then we back to the original place which is index 1, then continue k + 2.

(Eg. k + 2 = walk two steps become index 3.)

And just keep add up the step and repeat the way until success find for a location which is empty to store the key.

table efficiency decreases.

<table>
<tr>
<td></td>
<td>


</td>
<td></td>
</tr>
<tr>
<td>Quadratic Probing</td>
<td>
- Given search key k, the step is the square of the step number
- Probe to k + 1, k + 2^2, k + 3^2, ... k + n^2
- Little similar concept with linear but change walk by one step and one step to power of step (n^2)

Example:
</td>
<td>
✅
- Reduces primary clustering by spreading out probe attempts further apart.

❌
- Requires more efforts to compute the indices.
- Remains vulnerable to secondary clustering, where multiple keys are hashed to the same index and follow the same probe sequence.
</td>
</tr>
</table>

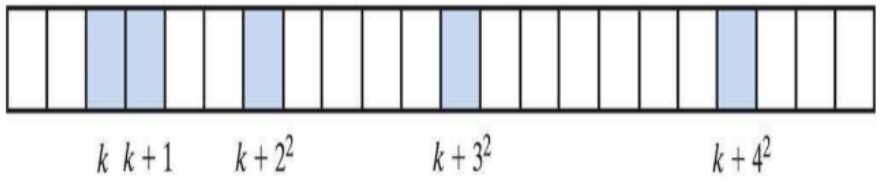


Fig. 18-7 A probe sequence of length five using quadratic probing.

Key	h(key)
15	15 modulo 7 = 1
26	26 modulo 7 = 5
37	37 modulo 7 = 2
42	42 modulo 7 = 0
59	59 modulo 7 = 3
68	68 modulo 7 = 5

→ $5+1, 5+2^2, 5+3^2, 5+4^2$

Double Hashing

Example:

Hashing is used to index and retrieve items from the hashed table. The following keys are to be hashed into a hash table of size 11:

65	42	85	67	23	51

Given the primary hash function:

h1 (key) = key modulo 11

and that collisions are resolved by double hashing with the secondary hash function:

h2 (key) = 7 - key modulo 7

Working step	Resulting hash table

✅

- Reaches every location in the hash table if table size is prime.
- Avoid primary and secondary clustering by using two independent hash functions, resulting in a more even distribution of entries.

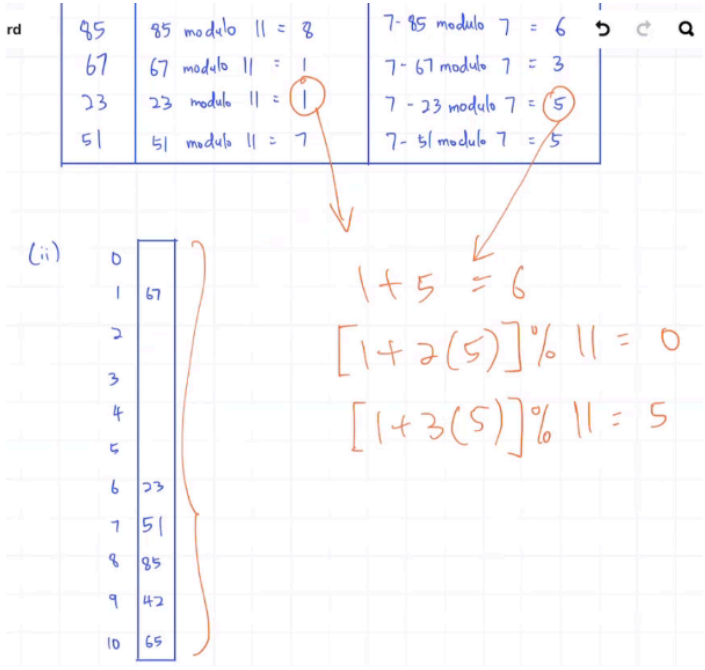
❌

- Time consuming
- More computationally expensive because two hash functions need to be computed

Key	h1	h2
65	10	2
42	9	7
85	8	6
67	1	4
23	1	5
51	7	2

0	1	2	3	4	5	6	7	8	9	10
	67					23	51	85	42	65

#记得先做了modulo ；modulo: xx/ 11 >(-整数)> * 11= h1



for each operation.

Separate Chaining (Close Addressing)

- Each index in the hash table points to a list (or chain) of entries.
- When collisions occur, new entries are simply appended to the list at that index.

✅
- No need to search for empty slots as in open addressing.

Example:

You are intending to input the keys (15, 26, 37, 42, 59, 68) into a hash table of size 7 by using the hashing function below:-

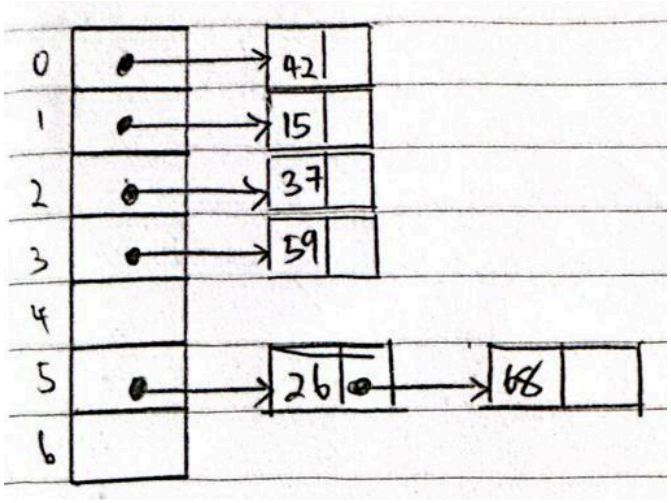
```
h(key)= key modulo 7
```

(i) Compute hash indexes for each key and represent them in the hash table by using the following collision resolution techniques respectively.

- **Separate Chaining** (Close Addressing Scheme) (5 marks)

Working Step	Resulting Hash table

Key	h1
15	1
26	5
37	2
42	0
59	3
68	5



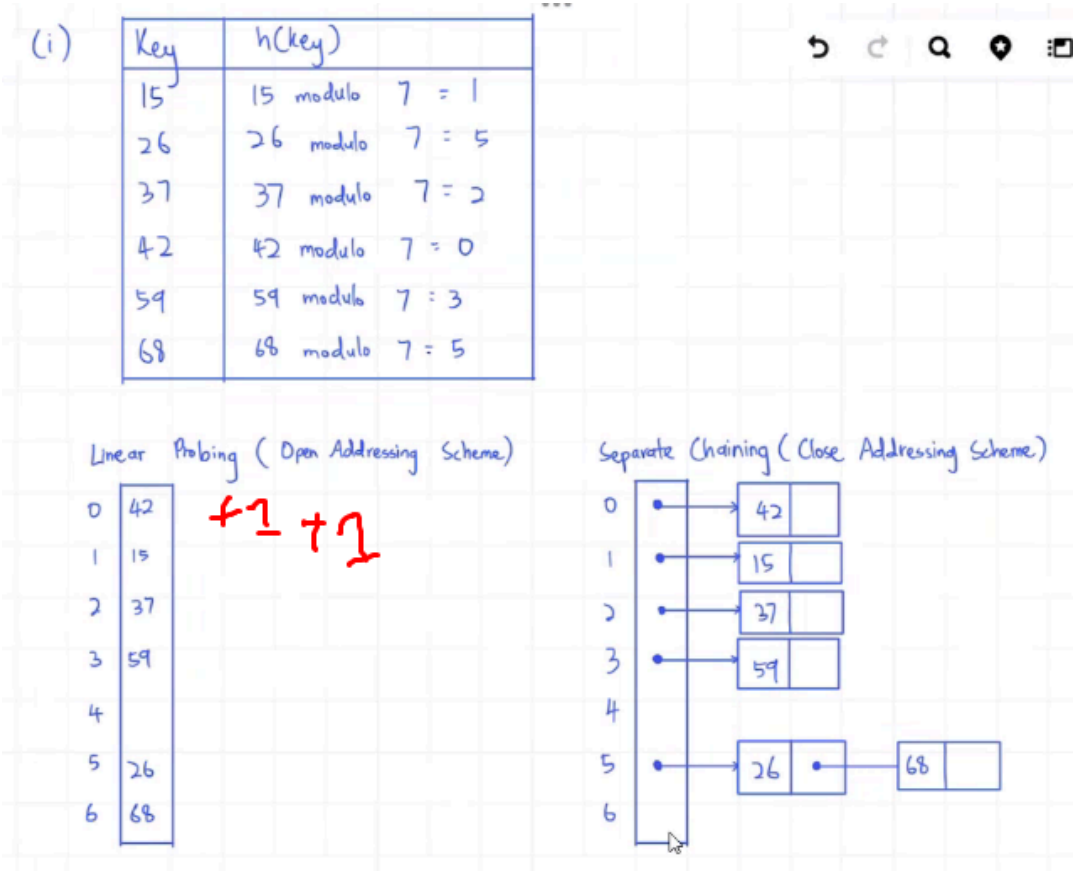
- Efficient insertion, as each bucket can hold multiple entries.
- The table size does not need to be a prime number.

❌

- Uses more memory due to the linked lists or chains stored at each index.

Separate Chaining (Close Addressing) vs. Open Addressing

Separate Chaining (Close Addressing)	Open Addressing
✅: - Provides faster dictionary operations on average - Can use smaller hash table - Less frequent rehashing If the same array size for a hash table, use more memory due to linked chains.	If the same array size for a hash table, use less memory due to no linked chains.



PYQ

C2-ADTs: DSA_N_C1-5

Question 1

A list is a collection of entries where each entry has a position. The first position of the list is 1.

a) Write a complete abstract data type (ADT) specification for a list which has the following operations:

- Add a new entry at the back of the list,
- Remove an entry from a specified position in the list,
- Check if the list contains a specified entry, and
- Check if the list is empty.

b) Write the Java interface based on your ADT specification in Question 1 a).

(20 marks)

(5 marks)

[Total: 25 marks]

042021-Q1a: Abstract Data Type (ADT) Specification for a List

ADT Name: List

Description:

The List ADT represents a collection of entries where each entry is associated with a unique position, starting at position 1. The list allows for dynamic addition and removal of entries while providing the capability to check for the presence of specific entries and the emptiness of the list.

Operations:

1. add(T entry): Adds a new entry to the back of the list.
   Postcondition: The entry is appended to the end of the list.

2. T remove(int position): Removes the entry at the specified position from the list.
   Precondition: The position must be valid (1 to the current size of the list).
   Postcondition: The entry at the specified position is removed. If the position is invalid, an error is raised.

Return: A reference to the removed entry; 'null' if the list was empty or if 'givenPosition' is invalid.

3. boolean contains(T entry): Checks if the list contains the specified entry.
   Postcondition: Returns true if the entry is found, otherwise false.

4. boolean isEmpty(): Checks if the list is empty.
   Postcondition: Returns true if the list has no entries, otherwise false.

Q1b: Java Interface Based on ADT Specification

```
public interface ListInterface<T> {
    void add(T entry); // Add a new entry at the back of the list
    T remove(int position) throws IndexOutOfBoundsException; // Remove an entry
from a specified position
    boolean contains(T entry); // Check if the list contains a specified entry
    boolean isEmpty(); // Check if the list is empty
}
```

Question 2
Write a Java class which implements your Java interface from Question 1 b) and provides an array- based implementation of the list ADT according to your specification. (25 marks)
DSA_N_C1-5

[Total: 25 marks]

042021-Q2: Java Class Implementing the List ADT

```
public class ArrayList<T> implements ListInterface<T> {
    private T[] array;
    private int size;
    private static final int DEFAULT_CAPACITY = 10;

    public ArrayList() {
        this.array = (T[]) new Object[DEFAULT_CAPACITY];
        this.size = 0;
    }

    @Override
    public void add(T entry) {
        ensureCapacity();
        array[size++] = entry;
    }

    @Override
    public T remove(int position) throws IndexOutOfBoundsException {
        if (position < 1 || position > size) {
            throw new IndexOutOfBoundsException("Position out of bounds.");
        }
        T removedEntry = array[position - 1];
        for (int i = position - 1; i < size - 1; i++) {
            array[i] = array[i + 1];
        }
        size--;
        return removedEntry;
    }

    @Override
    public boolean contains(T entry) {
        for (int i = 0; i < size; i++) {
            if (array[i].equals(entry)) {
                return true;
            }
        }
        return false;
    }

    @Override
    public boolean isEmpty() {
```

```
        return size == 0;
    }

    private void ensureCapacity() {
        if (size >= array.length) {
            T[] newArray = (T[]) new Object[array.length * 2];
            System.arraycopy(array, 0, newArray, 0, array.length);
            array = newArray;
        }
    }
}
```

Question 1
Consider a scenario where you are designing an application for TAR UMT student record system. The system should store information about students, including their names, IDs, and grades. You are required to write an Abstract Data Type (ADT) specification which consists of operations for manipulating the student data.

a)Choose an appropriate Abstract Data Type (ADT) that can be applied in TAR UMT student record system. Justify your answer for the selection and explanation on what object would be stored in this ADT. (5 marks)

b)Write an ADT specification based on the selected ADT collection from Question 1 a), include THREE (3) operations such as add, remove and getEntry methods with ADT specification format as shown below:
ADT Title:
Description:
For each operation, state the following information:

- The operation's name, description,
- The operation's preconditions (if any) and postconditions
- The return value of the operation (if any)

(10 marks)

c)Write a class which implements TWO (2) operations of the ADT specification from Question 1 b).

102023-Q1a: choose adt and explain object stored

Selected ADT: List

List ADT is suitable for the TARUMT student record system because

- it allows dynamic storage. Student records can be added or removed efficiently when students enroll or leave
- it supports quick access to records at any position, allowing easy retrieval, update or deletion based on their index or position in the list

Stored Object: each entry in the List will represent a Student object contains field such as name, ID, grades

102023-Q1b: ADT Specification

ADT Title: StudentList

Description: The StudentList ADT manages a collection of student records, providing operations to add, remove, and retrieve student information.

Operations:

Operation Name: addStudent

(10 marks) [Total: 25 marks]	Description: Adds a new Student object to the end of the list. Preconditions: The Student object must not be null. Postconditions: The Student object is added to the end of the list, and the size of the list is incremented by one. Return Value: None. **Operation Name: removeStudent** Description: Removes the Student object from the list at a specified position. Preconditions: The index must be within the bounds of the list (0 ≤ index < size). Postconditions: The Student object at the specified index is removed, and the size of the list is decremented by one. Return Value: The Student object that was removed. **Operation Name: getStudent** Description: Retrieves the Student object at a specified position in the list. Preconditions: The index must be within the bounds of the list (0 ≤ index < size). Postconditions: The Student object at the specified index is returned. Return Value: The Student object at the specified index. 102023

BigO?

012024 - Question 2

a)

The efficiency of an algorithm is gauged through its execution time and the associated memory usage. Comparing two algorithms by executing the code is not easy. Explain TWO (2) difficulties associated with the comparison of two algorithms based on their execution time. (8 marks)

b)

```
public int findMax(int[] array) {
    int max = Integer.MIN_VALUE;

    for (int i = 0; i < array.length; i++) {
        if (array[i] > max) {
            max = array[i];
        }
    }
}
```

Figure 1: findMax method

Figure 1: findMax method

Figure 1 shows the findMax method. Examine the given algorithm and determine its time complexity using Big O notation. Provide a brief explanation of your answer. (5 marks)

(ii)Illustrate whether the time complexity of this algorithm would change if the input array is sorted in ascending order. If yes, provide the new time complexity; if no, explain why. (5 marks)

c) A collision occurs in a hash table when the hash function produces the same hash value. Show how separate chaining and open addressing are used to resolve hashing collision issues.
(7 marks)[Total: 25 marks]

a) Difficulties in Algorithm Execution Time Comparison

Environmental Factors: Different systems (CPU, memory) can skew execution time results.
Scalability: Execution doesn't show how algorithms perform with growing data sizes.

b) findMax Method Time Complexity

Complexity: O(n), as it checks each element once.
The method's loop takes time that grows linearly with the array's size, as it must examine every element to find the maximum value.

c) Hash Collision Resolution

Separate Chaining: Use linked lists at each index to store items with the same hash value.
Open Addressing: Find the next free index using patterns like linear, quadratic probing, or double hashing.