Table of contents

C1

(d) Infix: a / b + c − d	Postfix: x= ab/ y= x c + z= y d - ab/ c+ d-	Prefix: x= /ab y= + x c z= - y d - + /ab cd

Abstraction and Encapsulation

Encapsulation hides the complexity behind a simple interface	**Modularity** organizes a program into manageable parts	**Abstraction** simplifying complex reality by focusing on the essential features and ignoring the details that are not important for the current purpose
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.	

List, Stack, Queue

(Definition, Terminology, Basic Operations, Concepts)

List

Definition	A List is a linear data structure that stores a collection of elements in a sequential manner. Each element in a list can be accessed by its index, which represents its position in the list.
Terminology	Node: An individual element in a list that contains data and a reference (or pointer) to the next node. Head: The first node in the list. Tail: The last node in the list (may point to null). Size: The total number of elements in the list.
Basic Operations	Add (Insertion): Insert an element at a specified position or at the end of the list. Remove (Deletion): Remove an element from a specified position. Get (Access): Retrieve an element from a specified position. Replace: Replace an element at a specified position with a new value. Contains: Check if a specific element exists in the list. Is Empty: Check if the list is empty. Get Size: Retrieve the number of elements in the list.
Concepts	Dynamic Size: Lists can grow or shrink in size as elements are added or removed. Ordered Collection: Elements in a list maintain their order based on their insertion sequence. Linked vs. Array Lists: Lists can be implemented using linked nodes or contiguous memory (arrays), affecting performance for various operations.

Stack

Definition	A Stack is a linear data structure that follows the Last In, First Out (LIFO) principle, where the last element added is the first one to be removed.
Terminology	Node: An element in the stack that contains data and a reference to the next node. Top: The node at the top of the stack from which elements are added or removed. Push: The operation to add an element to the top of the stack. Pop: The operation to remove and return the element from the top of the stack. Peek: The operation to view the top element without removing it.
Basic Operations	Push: Add a new entry onto the stack. Pop: Remove and return the top entry from the stack. Peek: Return the top entry without removing it. Is Empty: Check if the stack is empty.
Concepts	LIFO Structure: Stacks operate in a manner where the most recently added item is the first to be removed. Use Cases: Stacks are commonly used in function calls, expression evaluation, backtracking algorithms, and undo mechanisms in applications.

Queue

Definition	A Queue is a linear data structure that follows the First In, First Out (FIFO) principle, where the first element added is the first one to be removed.
Terminology	Node: An element in the queue that contains data and a reference to the next node. Front: The node at the front of the queue from which elements are removed. Back (or Rear): The node at the back of the queue where elements are added. Enqueue: The operation to add an element at the back of the queue. Dequeue: The operation to remove and return the element from the front of the queue.
Basic Operations	Enqueue: Add a new entry to the back of the queue. Dequeue: Remove and return the front entry from the queue. Get Front: Return the front entry without removing it. Is Empty: Check if the queue is empty.
Concepts	FIFO Structure: Queues operate such that the oldest element is served first. Use Cases: Queues are often used in scheduling algorithms, breadth-first search (BFS) in graphs, and managing requests in order of arrival.

ADT Specification

(Definition, how to write ADT specification, ADT Interface and Java code Implementation on the basic operations)

1. Definition

Written in a natural language (e.g. English) and are independent of any programming language.

• Used as specifications for concrete data types (i.e. the actual data types used in programs)

2. how to write ADT specification

ADT List

A list is a linear collection of entries of a type T, which allows duplicate elements. Entries can be added at a specified position or at the end of the list. The list positions start from 1.

(The position-based operations (e.g., getEntry, replace) in List are not applicable for Queue or Stack, so they are omitted.)

ADT Stack

A stack is a linear collection of entries of type T, which allows duplicate elements and operates on a Last In, First Out (LIFO) principle. Entries are added and removed from the top of the stack.

ADT Queue

A queue is a linear collection of entries of type T, which allows duplicate elements and operates on a First In, First Out (FIFO) principle. Entries are added at the back and removed from the front.

Method Specifications:

end of the list - top of the stack - back、front of the queue

void add(T newEntry) void push(T newEntry) void enqueue(T newEntry)	Description: Adds newEntry to the end of the list| **back** of the queue . Postcondition: newEntry has been added to the end of the list.
boolean add(Integer newPosition, T newEntry)	Description: Adds newEntry at position newPosition within the list. Position 1 indicates the first entry. Precondition: newPosition must be between 1 and total entries + 1. Postcondition: newEntry has been added to the indicated position. Returns: true if newEntry was successfully added, false otherwise.
T remove(Integer givenPosition)	Description: Removes the entry at position givenPosition in the list. Precondition: givenPosition must be between 1 and total entries. Postcondition: The entry at givenPosition has been removed. Returns: The entry that was removed from the list.
T pop() T dequeue()	Description: Removes the entry at the top of the stack| front of the queue . Precondition: The stack is not empty. Postcondition: The top entry has been removed from the stack. Returns: The entry that was removed from the top of the stack.
boolean contains(T anEntry)	Description: Determines whether the list contains anEntry. Postcondition: The list remains unchanged. Returns: true if anEntry is in the list, false otherwise.
T getEntry(Integer givenPosition)	Description: Retrieves the entry at position givenPosition in the list. Precondition: givenPosition must be between 1 and total entries. Postcondition: The list remains unchanged. Returns: The entry at position givenPosition.

T peek() T getFront()	Description: Retrieves the entry at the top of the stack| front of the queue without removing it. Precondition: The stack is not empty. Postcondition: The stack remains unchanged. Returns: The entry at the top of the stack.
boolean replace(Integer givenPosition, T newEntry)	Description: Replaces the entry at position givenPosition with newEntry. Precondition: givenPosition must be between 1 and total entries. Postcondition: The entry at givenPosition has been replaced. Returns: true if the replacement was successful, false otherwise.

void clear()

- Description: Removes all entries from the list.
- Postcondition: The list is now empty.

int getNumberOfEntries()

- Description: Gets the number of entries currently in the list.
- Postcondition: The list remains unchanged.
- Returns: The number of entries in the list.

boolean isEmpty()

- Description: Determines whether the list is empty.
- Postcondition: The list remains unchanged.
- Returns: true if the list is empty, false otherwise.

boolean isFull()

- Description: Determines whether the list is full.
- Postcondition: The list remains unchanged.
- Returns: true if the list is full, false otherwise.

3. ADT Interface and Java code Implementation on the basic operations

Java Interface

```
public interface ADTCollection<T> {
    void add(T newEntry);           // List operation
    void push(T newEntry);          // Stack operation
    void enqueue(T newEntry);       // Queue operation

    T remove(int givenPosition);    // List operation
    T pop();                        // Stack operation
    T dequeue();                    // Queue operation

    boolean contains(T anEntry);    // Common operation

    T getEntry(int givenPosition);  // List operation
    T peek();                       // Stack operation
    T getFront();                   // Queue operation
}
```

Array-based Implementation

```
Boleh guna: (Assume that the class and private variable have been defined)

public class ArrayCollection<T> implements ADTCollection<T> {
    private T[] array;
    private int numberOfEntries; // only list use
    private int frontIndex;  // For Queue front
    private int topIndex;    // For Stack top

private final int capacity;

    // Constructor
    public ArrayCollection(int capacity) {
        T[] tempArray = (T[]) new Object[capacity];
        array = tempArray;
        numberOfEntries = 0;
        frontIndex = 0;
        topIndex = -1;  // For Stack
```

```
    }

    // List operation: Add to the end of the list
    @Override
    public void add(T newEntry) {
        if (numberOfEntries < array.length) {
            array[numberOfEntries] = newEntry;
            numberOfEntries++;
        } else {
            System.out.println("List is full!");
        }
    }

    // Stack operation: Push to the top of the stack
    @Override
    public void push(T newEntry) {
        if (numberOfEntries < array.length) {
            topIndex++;
            array[topIndex] = newEntry;
            numberOfEntries++;
        } else {
            System.out.println("Stack is full!");
        }
    }

    // Queue operation: Enqueue at the back of the queue
    @Override
    public void enqueue(T newEntry) {
        if (numberOfEntries < array.length) {
            backIndex = (backIndex + 1) % array.length;
            array[backIndex] = newEntry;
            numberOfEntries++;
        } else {
            System.out.println("Queue is full!");
        }
    }

    // List operation: Remove at a specified position
    @Override
    public T remove(int givenPosition) {
        if (givenPosition >= 1 && givenPosition <= numberOfEntries) {
            T result = array[givenPosition - 1];
            // Shift elements left
            for (int i = givenPosition - 1; i < numberOfEntries - 1; i++) {
                array[i] = array[i + 1];
            }
            array[numberOfEntries - 1] = null;  // Clear last element
            numberOfEntries--;
            return result;
        } else {
            System.out.println("Invalid position!");
            return null;
        }
    }

    // Stack operation: Pop from the top of the stack
    @Override
    public T pop() {
        if (topIndex >= 0) {
            T result = array[topIndex];
            array[topIndex] = null;
            topIndex--;
            numberOfEntries--;
            return result;
        } else {
            System.out.println("Stack is empty!");
            return null;
        }
    }

    // Queue operation: Dequeue from the front of the queue
    @Override
    public T dequeue() {
```

```
        if (numberOfEntries > 0) {
            T result = array[frontIndex];
            array[frontIndex] = null;
            frontIndex = (frontIndex + 1) % array.length;
            numberOfEntries--;
            return result;
        } else {
            System.out.println("Queue is empty!");
            return null;
        }
    }

    // Common operation: Check if the collection contains an entry
    @Override
    public boolean contains(T anEntry) {
        for (int i = 0; i < numberOfEntries; i++) {
            if (array[i].equals(anEntry)) {
                return true;
            }
        }
        return false;
    }

    // List operation: Get entry at a specific position
    @Override
    public T getEntry(int givenPosition) {
        if (givenPosition >= 1 && givenPosition <= numberOfEntries) {
            return array[givenPosition - 1];
        } else {
            System.out.println("Invalid position!");
            return null;
        }
    }

    // Stack operation: Peek at the top of the stack
    @Override
    public T peek() {
        if (topIndex >= 0) {
            return array[topIndex];
        } else {
            System.out.println("Stack is empty!");
            return null;
        }
    }

    // Queue operation: Get the front of the queue
    @Override
    public T getFront() {
        if (numberOfEntries > 0) {
            return array[frontIndex];
        } else {
            System.out.println("Queue is empty!");
            return null;
        }
    }

    // Method to print array for debugging
    public void printArray() {
        for (int i = 0; i < array.length; i++) {
            System.out.print(array[i] + " ");
        }
        System.out.println();
    }
}
```

Linked-based Implementation

```
class Node<T> {
    T data;
    Node<T> next;
```

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

```
public class LinkedList<T> {
    private Node<T> head;
    private int numberOfEntries;

    public LinkedList() {
        head = null;
        numberOfEntries = 0;
    }

    // List operation: Add to the end of the list
    public void add(T newEntry) {
        Node<T> newNode = new Node<>(newEntry);
        if (head == null) {
            head = newNode;
        } else {
            Node<T> current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
        numberOfEntries++;
    }

    // List operation: Add to a specified position
    public void add(int newPosition, T newEntry) {
        if (newPosition >= 1 && newPosition <= numberOfEntries + 1) {
            Node<T> newNode = new Node<>(newEntry);
            if (newPosition == 1) {
                newNode.next = head;
                head = newNode;
            } else {
                Node<T> current = head;
                for (int i = 1; i < newPosition - 1; i++) {
                    current = current.next;
                }
                newNode.next = current.next;
                current.next = newNode;
            }
            numberOfEntries++;
        } else {
            System.out.println("Invalid position!");
        }
    }

    // List operation: Remove an entry at a given position
    public T remove(int givenPosition) {
        if (givenPosition >= 1 && givenPosition <= numberOfEntries) {
            T result;
            if (givenPosition == 1) {
                result = head.data;
                head = head.next;
            } else {
                Node<T> current = head;
                for (int i = 1; i < givenPosition - 1; i++) {
                    current = current.next;
                }
                result = current.next.data;
                current.next = current.next.next;
            }
            numberOfEntries--;
            return result;
        } else {
            System.out.println("Invalid position!");
            return null;
        }
    }
```

```
    // List operation: Retrieve entry at a given position
    public T getEntry(int givenPosition) {
        if (givenPosition >= 1 && givenPosition <= numberOfEntries) {
            Node<T> current = head;
            for (int i = 1; i < givenPosition; i++) {
                current = current.next;
            }
            return current.data;
        } else {
            System.out.println("Invalid position!");
            return null;
        }
    }

    // Method to replace an entry at a specified position
    public boolean replace(int givenPosition, T newEntry) {
        if (givenPosition < 1 || givenPosition > numberOfEntries) {
            return false; // Invalid position
        }

        Node<T> current = head;
        for (int i = 1; i < givenPosition; i++) {
            current = current.next; // Traverse to the specified position
        }
        current.data = newEntry; // Replace the entry
        return true; // Replacement successful
    }

    // List operation: Check if the list contains an entry
    public boolean contains(T anEntry) {
        Node<T> current = head;
        while (current != null) {
            if (current.data.equals(anEntry)) {
                return true;
            }
            current = current.next;
        }
        return false;
    }

    // Common method to check if the list is empty
    public boolean isEmpty() {
        return numberOfEntries == 0;
    }

    // Get the number of entries in the list
    public int getNumberOfEntries() {
        return numberOfEntries;
    }
}
```

```
public class LinkedStack<T> {
    private Node<T> topNode;

    public LinkedStack() {
        topNode = null;
    }

    // Stack operation: Push a new entry onto the stack
    public void push(T newEntry) {
        Node<T> newNode = new Node<>(newEntry);
        newNode.next = topNode;
        topNode = newNode;
    }

    // Stack operation: Pop an entry from the top of the stack
    public T pop() {
        if (topNode != null) {
            T result = topNode.data;
            topNode = topNode.next;
            return result;
        } else {
```

```
            System.out.println("Stack is empty!");
            return null;
        }
    }

    // Stack operation: Peek at the top entry without removing it
    public T peek() {
        if (topNode != null) {
            return topNode.data;
        } else {
            System.out.println("Stack is empty!");
            return null;
        }
    }

    // Method to replace the top element
    public boolean replace(int givenPosition, T newEntry) {
        if (givenPosition != 1 || topNode == null) {
            return false; // Invalid position or stack is empty
        }
        topNode.data = newEntry; // Replace the top entry
        return true; // Replacement successful
    }

    // Check if the stack is empty
    public boolean isEmpty() {
        return topNode == null;
    }
}
```

```
public class LinkedQueue<T> {
    private Node<T> frontNode;
    private Node<T> backNode;

    public LinkedQueue() {
        frontNode = null;
        backNode = null;
    }

    // Queue operation: Enqueue a new entry at the back of the queue
    public void enqueue(T newEntry) {
        Node<T> newNode = new Node<>(newEntry);
        if (isEmpty()) {
            frontNode = newNode;
        } else {
            backNode.next = newNode;
        }
        backNode = newNode;
    }

    // Queue operation: Dequeue an entry from the front of the queue
    public T dequeue() {
        if (!isEmpty()) {
            T result = frontNode.data;
            frontNode = frontNode.next;
            if (frontNode == null) {
                backNode = null;
            }
            return result;
        } else {
            System.out.println("Queue is empty!");
            return null;
        }
    }

    // Queue operation: Get the front entry without removing it
    public T getFront() {
        if (!isEmpty()) {
            return frontNode.data;
        } else {
            System.out.println("Queue is empty!");
            return null;
```

```
        }
    }

    // Method to replace an entry at a specified position
    public boolean replace(int givenPosition, T newEntry) {
        if (givenPosition < 1 || givenPosition > getNumberOfEntries()) {
            return false; // Invalid position
        }
        Node<T> current = frontNode;
        for (int i = 1; i < givenPosition; i++) {
            current = current.next; // Traverse to the specified position
        }
        current.data = newEntry; // Replace the entry
        return true; // Replacement successful
    }

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

    // Get the number of entries in the queue
    public int getNumberOfEntries() {
        int count = 0;
        Node<T> current = frontNode;
        while (current != null) {
            count++;
            current = current.next;
        }
        return count;
    }
}
```

Chapter 3: Efficiency of Algorithms

Highlights: Why and How - Calculate O(??),

Two ways to measure the efficiency of algorithms:

1. Experimental studies - drawback

2. Analysis of algorithms

Question 2 (2024-JAN)

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)

1. Hardware Variability:

Issue: Different hardware setups (CPU speed, memory, etc.) can significantly affect execution time.

Example: Algorithm A might perform faster on a high-end machine but slower on a lower-end one, making direct comparison unreliable across different systems.

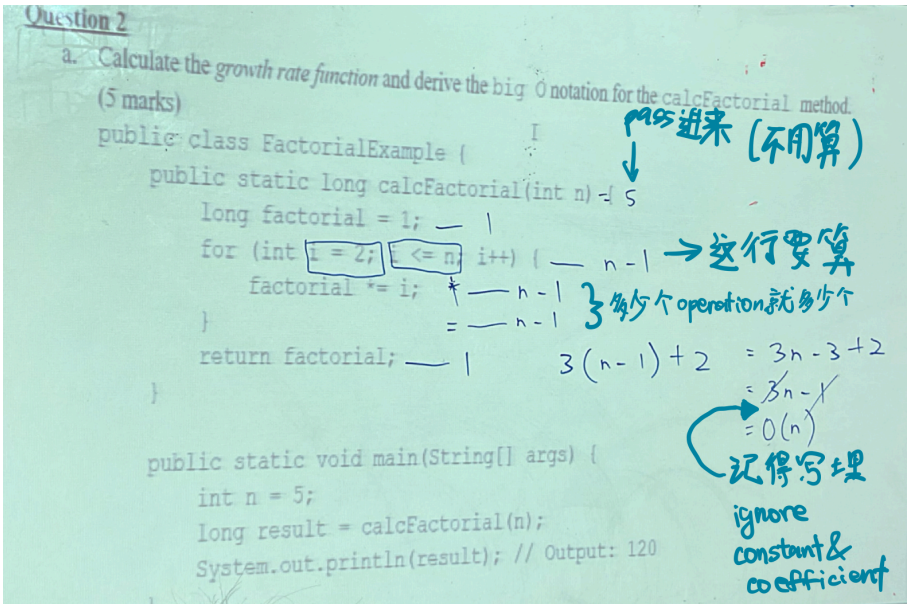
2. Input Size and Nature:

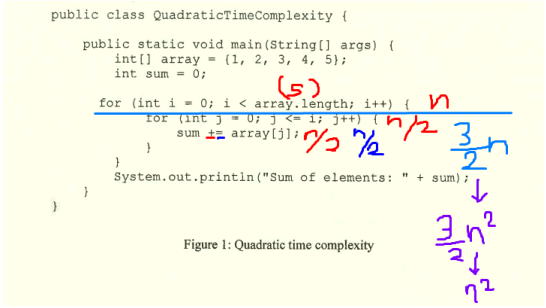
Issue: Execution time can vary greatly based on the size and type of input data.

Example: Algorithm B might be faster for small inputs but slower for larger inputs, or behave differently with sorted versus unsorted data. This makes it difficult to compare algorithms consistently without considering all possible input scenarios.

Big O

(Methods of measurement, calculate growth rate, how to determine the big O)





Chapter 6 : Recursive

Highlights: Definitions, coding

Implementation the recursive methods and changing the iterative to recursive methods

Example-binarysearch

```
public int binarySearchIterative(int[] arr, int target) {
    int low = 0, high = arr.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1; // Target not found
}
```

```
public int binarySearchRecursive(int[] arr, int target,
int low, int high) {
    if (low > high) {
        return -1; // Base case: target not found
    }
    int mid = (low + high) / 2;
    if (arr[mid] == target) {
        return mid;
    } else if (arr[mid] < target) {
        return binarySearchRecursive(arr, target, mid + 1,
high); // Recursive call
    } else {
        return binarySearchRecursive(arr, target, low, mid
- 1); // Recursive call
    }
}
```

（mid 是当前数组的中间位置。如果 arr[mid] 小于目标值 target，说明目标值一定比中间的这个值大。）

Chapter 7 : Sorted Lists

Highlights: Methods

Chapter 8 : Algorithms for Searching and Sorting

Highlights: Sequential and Binary Search

Sorting - bubble sort, insertion and selection sort

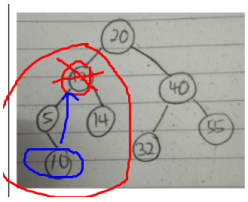
Insertion: 是看下一个index 有没有比”前面全部”的小 就要插队过去
Selection: 是看现在index的“后面全部”有没有比较小 有就互换位置
Bubble 是跟右边一pair一pair轮流交换 才是一轮end of pass，全完对了就不用继续写的

Focus: diagram (Illustration)

Chapter 9 : Binary Trees
Highlights: Constructing binary tree and traversal order.
Focus: Constructing, adding, removing, traversal and sorting and searching

Binary search tree

Pre-order Traversal (Root, Left, Right); In-order Traversal (Left, Root, Right); Post-order Traversal (Left, Right, Root)
General Remove: left right most



(2024-May)

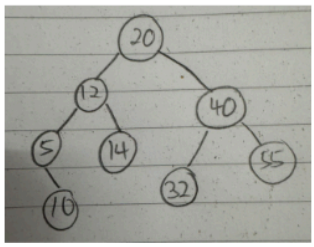
Question 4

Binary Search Tree is a non-linear data structure where each node has at most two children. It is a special data structure used for data storage purposes.

20	40	32	12	5	14	55	10

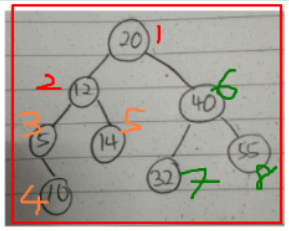
Figure 3: Number array

a) Construct a binary search tree using the given integer value in Figure 3. (8 marks)

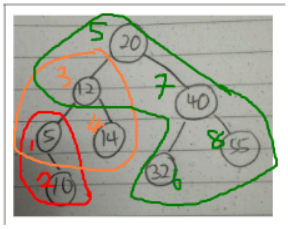


b) Based on answer for binary search tree in **Question 4 a)**, show the results of the **preorder, inorder** and **postorder** traversal outputs. (6 marks)

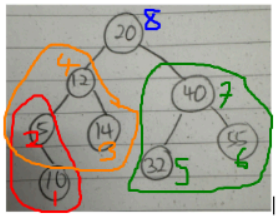
Preorder: 20, 12, 5, 10, 14, 40, 32, 55 (root left right)



Inorder: 5, 10, 12, 14, 20, 32, 40, 55 (left root right)



Postorder: 10, 5, 14, 12, 32, 55, 40, 20 (left right root)



Hashing - collision

(Term, definition, hashing methods and hashing collision methods, calculate hash index for insertion)

Note link: DSA_N_C1-5

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)

Separate Chaining:
Process: When a collision occurs (i.e., two keys have the same hash value), the new key is added to a linked list at the same index.
Example: For keys 10 and 15, if both hash to index 0, a linked list is created at index 0 to store both values.
Result:
Index 0: [10 -> 15]

Open Addressing (Linear Probing):
Process: If a collision happens, the algorithm searches for the next available slot in the hash table (sequentially).
Example: If 10 hashes to index 0 and 15 also hashes to index 0, it checks index 1. If index 1 is empty, 15 is placed there.
Result:
Index 0: 10
Index 1: 15

d) Describe **TWO (2)** hashing methods to solve the collision problem. (4 marks)

d) Describe hashing function by providing relevant examples. (6 marks)

1. Chaining (Separate Chaining)
Collisions are resolved by storing multiple elements in a linked list at the same index.
If a collision occurs, the new element is added to the list at that index.
Example:
Keys: 10, 15, 20, all hash to the same index.
Result: A linked list at that index holding [10, 15, 20].

2. Open Addressing (Linear Probing)
When a collision occurs, it searches for the next empty slot in the table.
Linear probing checks the next index sequentially until an empty slot is found.
Example:
Keys: 10, 15, 20. If 10 is at index 0, 15 moves to index 1, 20 moves to index 2.

**take from chatgpt:

Term and Definition

Hashing is a technique used to uniquely identify a specific object from a group of similar objects. It involves converting input data (often called a "key") into a fixed-size string of characters, which is typically a hash code or hash value. This hash value is used to index data in a hash table, enabling quick data retrieval.

Hashing Methods

Division Method:

This method involves dividing the key by a prime number and using the remainder as the hash index.

Formula:

$$\text{hash index} = \text{key} \mod \text{table size}$$

Multiplication Method:

This method multiplies the key by a constant (usually between 0 and 1), extracts the fractional part, and multiplies it by the table size to get the hash index.

Formula:

$$\text{hash index} = \lfloor \text{table size} \times (\text{key} \times A \mod 1) \rfloor$$

Where A is a constant (e.g., $A = \frac{\sqrt{5}-1}{2}$).

Universal Hashing:

This method uses a family of hash functions to reduce collisions. A hash function is randomly chosen from a set, making it difficult for an adversary to predict the function to be used.

Hashing Collision Methods

A collision occurs when two keys hash to the same index. Various methods are used to handle collisions:

Chaining:

In this method, each index of the hash table points to a linked list of entries that hash to the same index. When a collision occurs, the new entry is simply added to the list.

Pros: Simple implementation, flexible in handling varying numbers of collisions.

Cons: Can lead to increased search times if many collisions occur.

Open Addressing:

In this method, when a collision occurs, the algorithm searches for the next available slot in the table. There are several strategies for finding the next slot:

Linear Probing: Checks the next slot in a sequential manner.

Quadratic Probing: Checks slots using a quadratic function (e.g., i^2).

Double Hashing: Uses a second hash function to determine the step size for finding the next available slot.

Rehashing:

This involves creating a new hash table with a larger size and rehashing all existing entries into the new table. It's often used when the load factor (number of entries divided by table size) exceeds a certain threshold.

Calculate Hash Index for Insertion

To calculate the hash index for inserting a key into a hash table, follow these steps:

1. Choose a Hash Function: Select a hashing method (e.g., division method).
2. Compute the Hash Index:
   Example: Suppose we have a hash table of size 10 and we want to insert the key 25.
   $$\text{hash index} = 25 \mod 10 = 5$$
   Using the division method:
3. Insert the Entry: Place the entry at index 5 of the hash table. If there is a collision, apply the chosen collision resolution method.