ASSIGNMENT 2 FRONT SHEET

Qualification	**BTEC Level 5 HND Diploma in Computing**		
Unit number and title	Unit 19: Data Structures and Algorithms		
Submission date		**Date Received 1st submission**	
Re-submission Date		**Date Received 2nd submission**	
Student Name	Nguyen Duc Tu	**Student ID**	GCH200690
Class	RE SU24	**Assessor name**	Do Hong Quan

Student declaration

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.

Student's signature	

Grading grid

P4	P5	P6	P7	M4	M5	D3	D4

❒ Summative Feedback:		❒ Resubmission Feedback:
Grade:	Assessor Signature:	Date:
Internal Verifier's Comments:		
IV Signature:		

Table of Contents

Introduction. 4

I. Design and implementation of ArrayList ADT and Linked List ADT (P4). 4

1. Arraylist 4

Defination 4

Operation 4

Source code + Explain: 5

2. Linkedlist 9

Definition 9

Operation: 10

Source code + Explain: 11

Implement error handling and report test results (P5) 16

Testing plan 16

Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm (P6) 18

Asymptotic Notations 18

Big-O Notation (O): 18

Big-Omega Notation (Ω): 20

Big-Theta Notation (Θ): 21

Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example (P7) 22

Time Complexity 22

Space Complexity 23

Example 23

Conclusion 27

References. 27

Introduction.

This assessment aims to demonstrate your ability to implement a complex Abstract Data Type (ADT) and associated algorithms in an executable programming language to solve a well-defined problem. You will be required to not only develop the ADT but also ensure that your implementation includes robust error handling and thorough testing. Additionally, you will explore the effectiveness of your algorithm through asymptotic analysis and identify two methods for measuring algorithm efficiency, supported by practical examples.

I. Design and implementation of ArrayList ADT and Linked List ADT (P4).

1. Arraylist

Defination:

The ArrayList class, found in the java.util package, functions as a dynamic array. Unlike fixed-size built-in arrays, ArrayLists can adjust their size automatically. This flexibility allows for elements to be added or removed as needed, facilitating efficient memory management for the user (educative, 2024).

Operation:

Acording to baeldung, 2024 and cmu.edu, 2024 the ArrayList ADT provides a flexible and dynamic way to handle a collection of elements. Below are the detailed descriptions of the core operations supported by the ArrayList:

- **Add (element)**

Description: Adds an element to the end of the ArrayList.

Complexity: O(1) on average. However, if the underlying array needs to be resized, the complexity can become O(n) for that particular add operation due to the need to allocate new memory and copy existing elements.

- **Get (index)**

Description: Retrieves the element at a specified index. If the index is out of bounds, it raises an IndexError.

Complexity: O(1) since accessing an element by index in an array is a constant-time operation.

- **Set (index, element)**

Description: Replaces the element at a specified index with a new element. If the index is out of bounds, it raises an IndexError.

Complexity: O(1) because it directly accesses and modifies the element at the specified index.

- **Remove**

Description: Removes the last element in the ArrayList. If the ArrayList is empty, it raises an IndexError.

Complexity: O(1) because it directly removes the last element.

- **Search (element)**

Description: Searches for the last occurrence of the specified element in the ArrayList and returns its index. If the element is not found, it returns -1.

Complexity: O(n) because it may need to traverse the entire list to find the element.

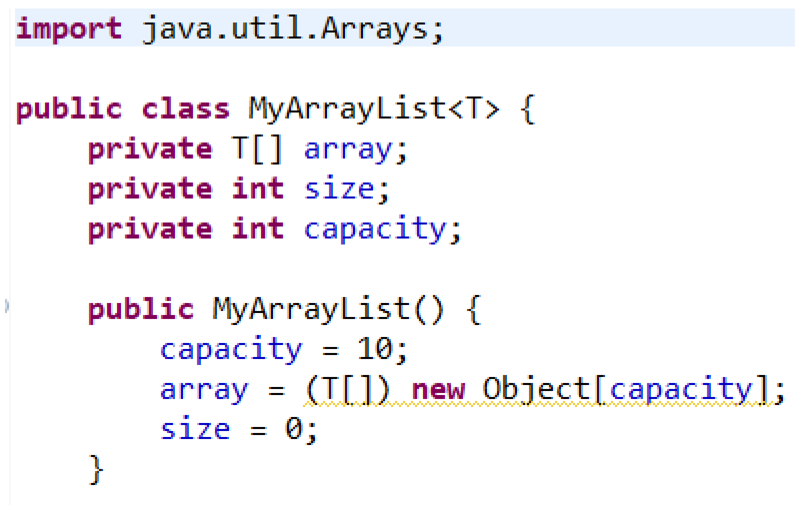
- **IsEmpty**

Description: Checks if the ArrayList is empty.

Complexity: O(1) since it only checks if the length of the internal array is zero.

Source code + Explain:

MyArrayList Class and Constructor:



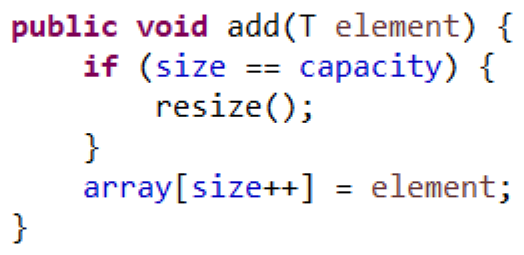
```
import java.util.Arrays;

public class MyArrayList<T> {
    private T[] array;
    private int size;
    private int capacity;

    public MyArrayList() {
        capacity = 10;
        array = (T[]) new Object[capacity];
        size = 0;
    }
```

- **T[] array**: An array to store elements of the list. It is generic, meaning it can store any type of object.
- **int size**: The number of elements currently in the list.
- **int capacity**: The maximum number of elements the array can hold before needing to resize.
- **capacity = 10**: Initializes the capacity of the array to 10.
- **array = (T[]) new Object[capacity]**: Creates a new array with the specified capacity.
- **size = 0**: Initializes the size to 0, meaning the list is initially empty.

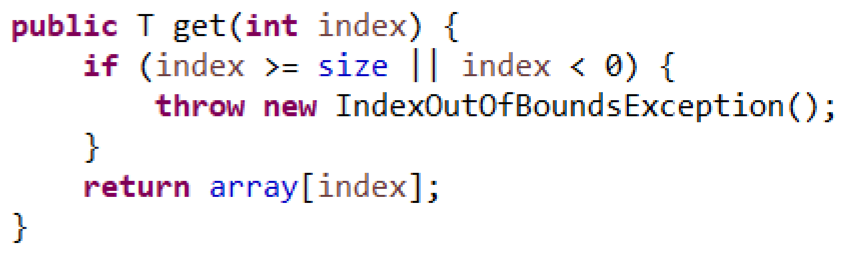
add(T element):



```
public void add(T element) {
    if (size == capacity) {
        resize();
    }
    array[size++] = element;
}
```

- **if (size == capacity)**: Checks if the array is full. If it is, it calls the `resize()` method to increase the array's capacity.
- **array[size++] = element**: Adds the element to the array and increments the size.

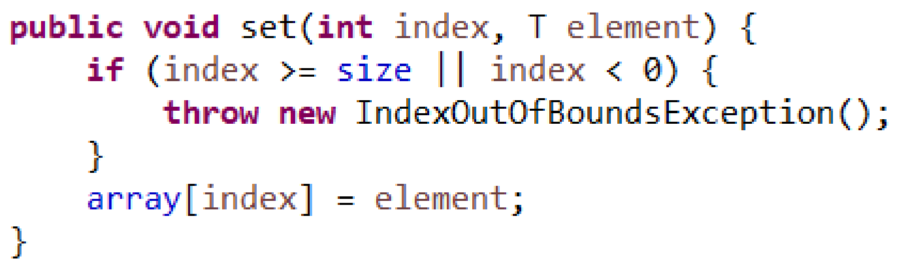
get(int index):



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

- **if (index >= size || index < 0)**: Checks if the index is within bounds. If not, it throws an IndexOutOfBoundsException.
- **return array[index]**: Returns the element at the specified index.

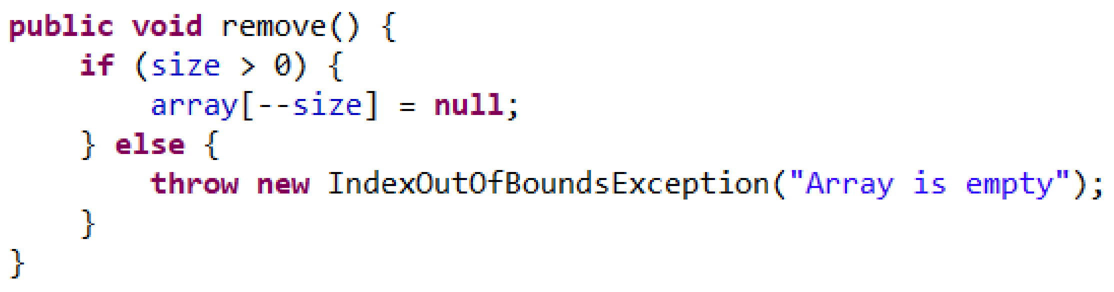
set(int index, T element):



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

- **if (index >= size || index < 0)**: Checks if the index is within bounds. If not, it throws an IndexOutOfBoundsException.
- **array[index] = element**: Sets the element at the specified index to the new value.

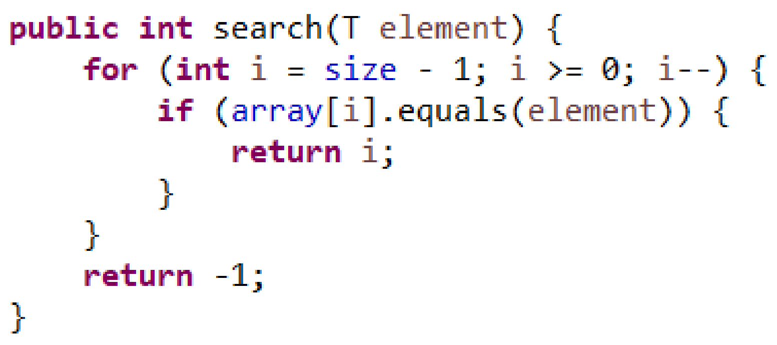
remove():



```
public void remove() {
    if (size > 0) {
        array[--size] = null;
    } else {
        throw new IndexOutOfBoundsException("Array is empty");
    }
}
```

- **if (size > 0)**: Checks if the array is not empty. If it is not, it decrements the size and sets the last element to null.
- **else**: If the array is empty, it throws an IndexOutOfBoundsException.

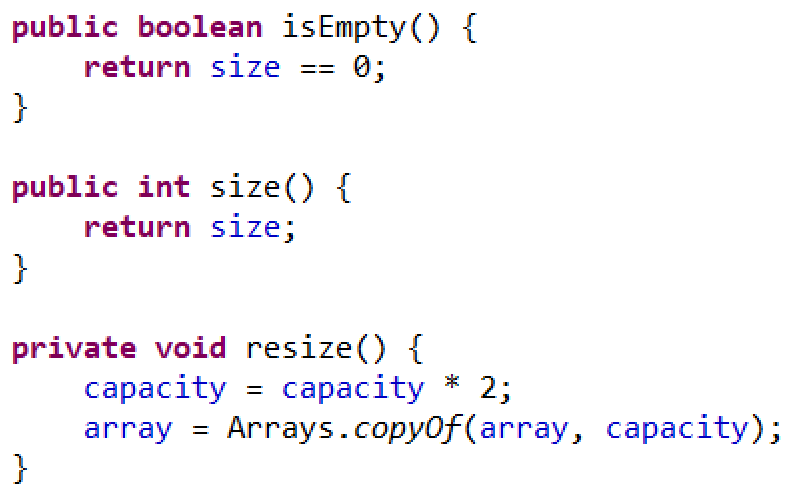
search(T element):



```
public int search(T element) {
    for (int i = size - 1; i >= 0; i--) {
        if (array[i].equals(element)) {
            return i;
        }
    }
    return -1;
}
```

- **for (int i = size - 1; i >= 0; i--)**: Iterates through the array from the last element to the first.
- **if (array[i].equals(element))**: Checks if the current element equals the specified element. If it does, it returns the index.
- **return -1**: If the element is not found, it returns -1.

isEmpty(), size(), resize():



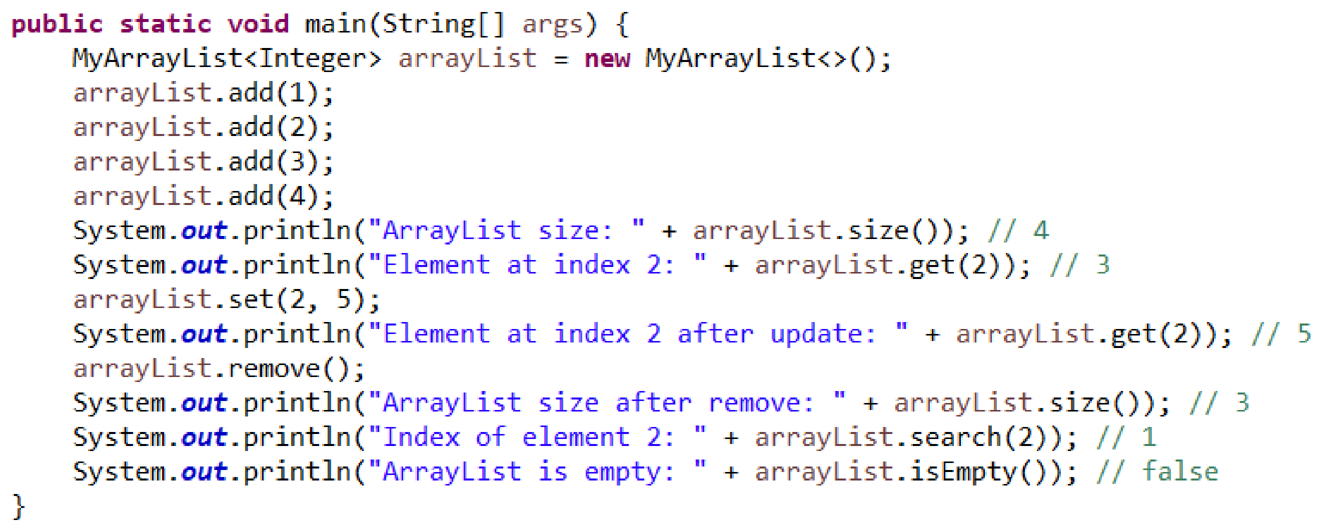
```
public boolean isEmpty() {
    return size == 0;
}

public int size() {
    return size;
}

private void resize() {
    capacity = capacity * 2;
    array = Arrays.copyOf(array, capacity);
}
```

- **return size == 0**: Returns true if the list is empty, otherwise returns false.
- **return size**: Returns the number of elements in the list.
- **capacity = capacity * 2**: Doubles the capacity of the array.
- **array = Arrays.copyOf(array, capacity)**: Copies the elements of the old array to a new array with the increased capacity.

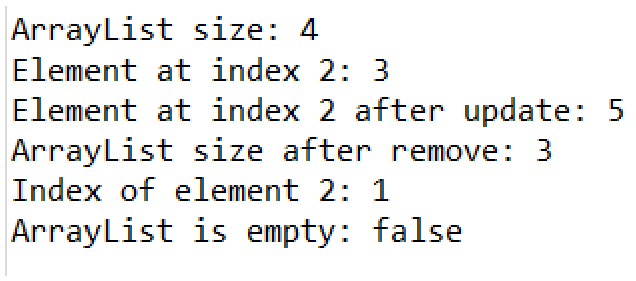
main():



```
public static void main(String[] args) {
    MyArrayList<Integer> arrayList = new MyArrayList<>();
    arrayList.add(1);
    arrayList.add(2);
    arrayList.add(3);
    arrayList.add(4);
    System.out.println("ArrayList size: " + arrayList.size()); // 4
    System.out.println("Element at index 2: " + arrayList.get(2)); // 3
    arrayList.set(2, 5);
    System.out.println("Element at index 2 after update: " + arrayList.get(2)); // 5
    arrayList.remove();
    System.out.println("ArrayList size after remove: " + arrayList.size()); // 3
    System.out.println("Index of element 2: " + arrayList.search(2)); // 1
    System.out.println("ArrayList is empty: " + arrayList.isEmpty()); // false
}
```

- Creates a new **MyArrayList** of **Integer** type and adds elements to it.
- Demonstrates the usage of **size(), get(), set(), remove(), search(),** and **isEmpty()** methods.

Result:



```
ArrayList size: 4
Element at index 2: 3
Element at index 2 after update: 5
ArrayList size after remove: 3
Index of element 2: 1
ArrayList is empty: false
```

2. Linkedlist

Definition:

A linked list in Java is a dynamic data structure whose size increases as you add elements and decreases as you remove elements from the list. The elements in the linked list are stored in containers, with the list holding a link to the first container. Each container has a link to the next container in the list. Whenever you add an element to the list using the add() operation, a new container is created and linked to the other containers in the list (Simplilearn, 2024).

Acording to GeeksforGeeks, 2023 there are two main types of linkedlists and the table below will illustrate these two:

Feature	Singly Linked List	Doubly Linked List
Node Structure	Contains data and a single link to the next node	Contains data and two links: one to the next node and one to the previous node
Navigation	Can be traversed only in one direction (forward)	Can be traversed in both directions (forward and backward)
Memory Usage	Uses less memory, as each node only needs to store one link	Uses more memory, as each node needs to store two links
Insertion/Deletion (General)	Easier and faster, especially at the beginning of the list	More complex due to maintaining two links, but can be easier when deleting nodes from the middle
Insertion/Deletion (End)	Less efficient, as it requires traversal from the head to the end	More efficient if the tail reference is maintained
Reverse Traversal	Not possible without modifying the structure	Possible due to the presence of the previous link
Use Cases	Suitable for simple lists, stacks, and queues	Suitable for complex data structures like dequeues and some types of caches

This table captures the key differences between singly and doubly linked lists in terms of their structure, navigation capabilities, memory usage, and performance characteristics for various operations.

Operation:

Acording to baeldung, 2024 and cmu.edu, 2024 linked lists support a variety of operations that allow for the manipulation of the elements they contain. Below are some of the primary operations along with their details:

- **add(T element)**

Description: Adds an element to the end of the list. This operation involves creating a new node and linking it to the end of the list.

Complexity: O(n) if the list does not maintain a tail pointer, as it needs to traverse the list to find the last node. O(1) if the list maintains a tail pointer, as it can directly link the new node to the end.

- **get(int index)**

Description: Retrieves the element at the specified index in the list. This involves traversing the list from the head to the given index.

Complexity: O(n) because it may need to traverse up to the nth node to retrieve the element.

- **set(int index, T element)**

Description: Updates the element at a given index with a new element. This involves traversing the list to the specified index and replacing the old element with the new one.

Complexity: O(n) because it may need to traverse up to the nth node to update the element

- **remove()**

Description: Deletes the last element of the list. This involves traversing the list to the second-to-last node and removing the link to the last node

Complexity: O(n) because it may need to traverse the entire list to find the second-to-last node.

- **search(T element)**

Description: Returns the index of the last occurrence of the specified element in the list, or -1 if the element is not found. This involves traversing the entire list to check each element.

Complexity: O(n) because it needs to traverse the entire list to find the last occurrence of the element.

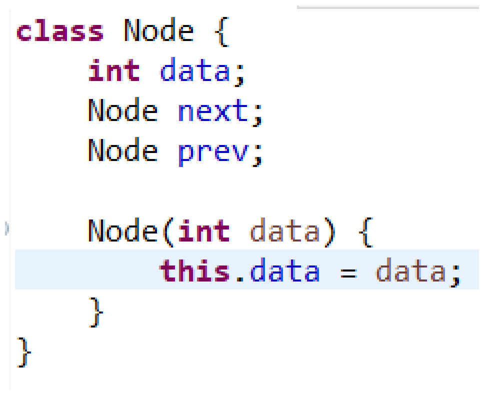
- **isEmpty()**

Description: Checks whether the list is empty. This involves checking if the head of the list is null

Complexity: O(1) because it only needs to check the head of the list.

Source code + Explain:

Node Class:

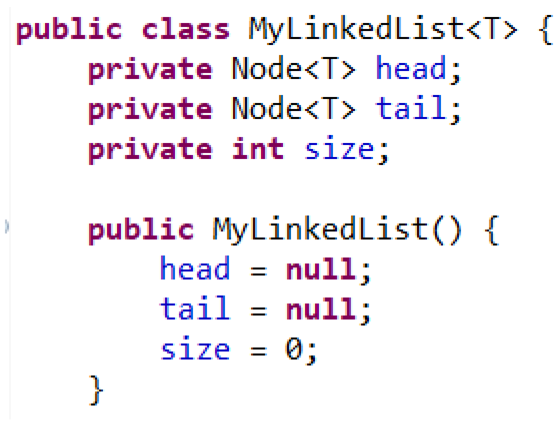


```
class Node {
    int data;
    Node next;
    Node prev;

    Node(int data) {
        this.data = data;
    }
}
```

- **T data**: Stores the data of the node.
- **Node<T> next**: Reference to the next node.
- **Node<T> prev**: Reference to the previous node.
- **Node(T data)**: Constructor initializes the node with the given data.

MyLinkedList Class and Constructor:

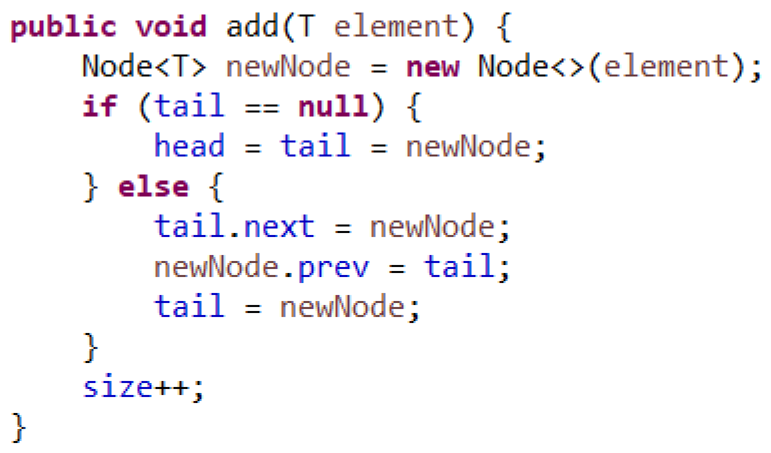


```
public class MyLinkedList<T> {
    private Node<T> head;
    private Node<T> tail;
    private int size;

    public MyLinkedList() {
        head = null;
        tail = null;
        size = 0;
    }
```

- **head = null**: Initializes the head to null.
- **tail = null**: Initializes the tail to null.
- **size = 0**: Initializes the size to 0, meaning the list is initially empty.

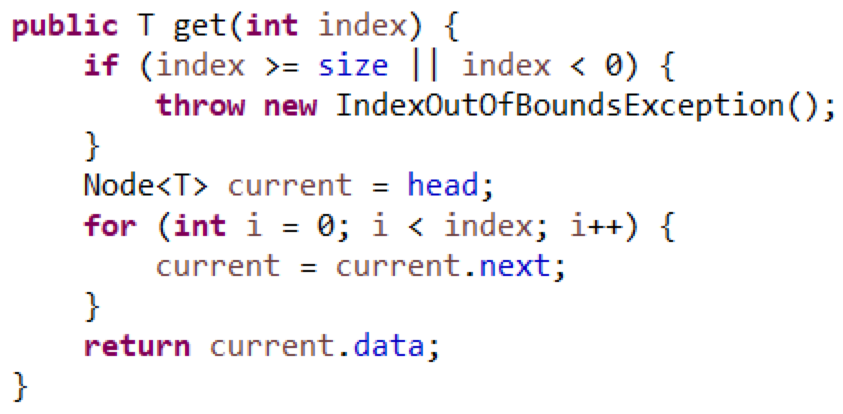
add(T element):



```java
public void add(T element) {
    Node<T> newNode = new Node<>(element);
    if (tail == null) {
        head = tail = newNode;
    } else {
        tail.next = newNode;
        newNode.prev = tail;
        tail = newNode;
    }
    size++;
}
```

- **Node<T> newNode = new Node<>(element)**: Creates a new node with the given data.
- **if (tail == null)**: Checks if the list is empty. If it is, both head and tail point to the new node.
- **else**: If the list is not empty, the current tail's next pointer is set to the new node, and the new node's prev pointer is set to the current tail. The new node becomes the tail.
- **size++**: Increments the size of the list.

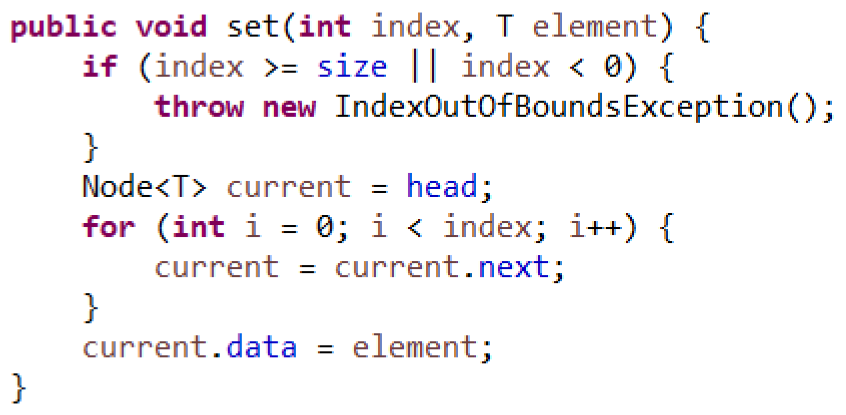
get(int index):



```java
public T get(int index) {
    if (index >= size || index < 0) {
        throw new IndexOutOfBoundsException();
    }
    Node<T> current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    return current.data;
}
```

- **if (index >= size || index < 0)**: Checks if the index is within bounds. If not, it throws an IndexOutOfBoundsException.
- **Node<T> current = head**: Starts traversal from the head.
- **for (int i = 0; i < index; i++)**: Traverses the list to the specified index.
- **return current.data**: Returns the data at the specified index.

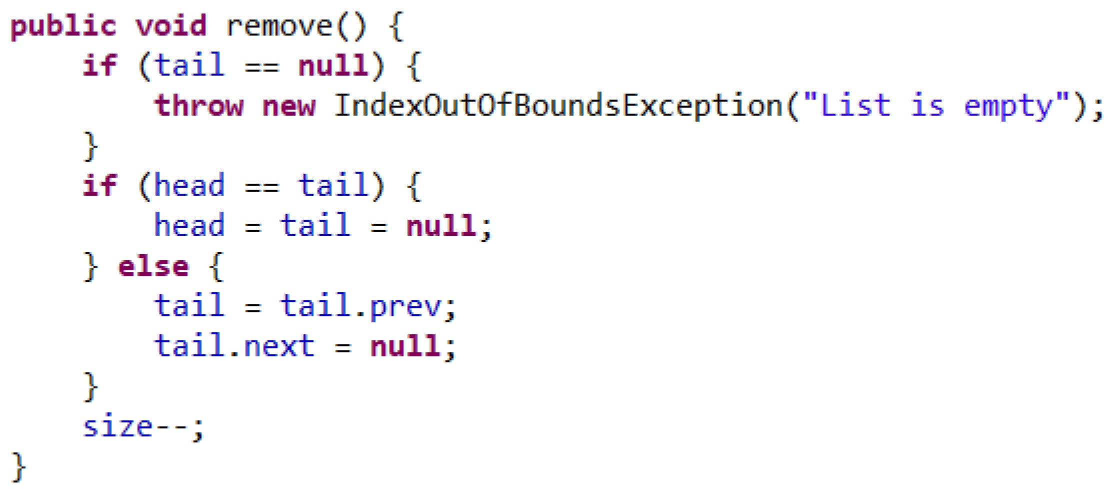
set(int index, T element):



```java
public void set(int index, T element) {
    if (index >= size || index < 0) {
        throw new IndexOutOfBoundsException();
    }
    Node<T> current = head;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    current.data = element;
}
```

- **if (index >= size || index < 0)**: Checks if the index is within bounds. If not, it throws an IndexOutOfBoundsException.
- **Node<T> current = head**: Starts traversal from the head.
- **for (int i = 0; i < index; i++)**: Traverses the list to the specified index.
- **current.data = element**: Sets the data at the specified index to the new value.

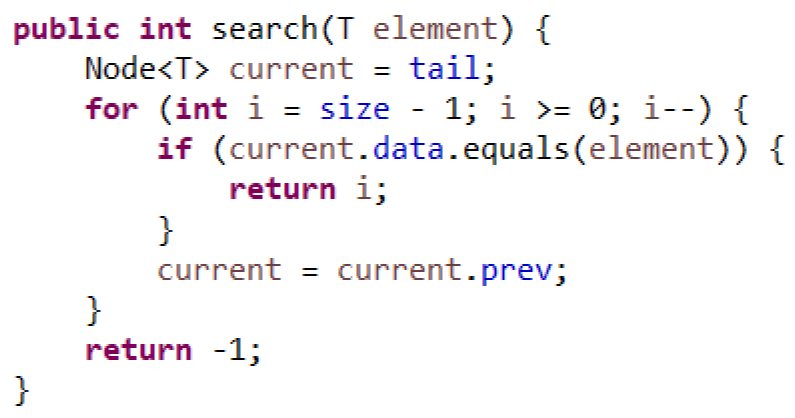
remove():



```java
public void remove() {
    if (tail == null) {
        throw new IndexOutOfBoundsException("List is empty");
    }
    if (head == tail) {
        head = tail = null;
    } else {
        tail = tail.prev;
        tail.next = null;
    }
    size--;
}
```

- **if (tail == null)**: Checks if the list is empty. If it is, it throws an IndexOutOfBoundsException.
- **if (head == tail)**: Checks if there is only one element in the list. If there is, both head and tail are set to null.
- **else**: If there are multiple elements, the tail is updated to the previous node, and the new tail's next pointer is set to null.
- **size--**: Decrements the size of the list.

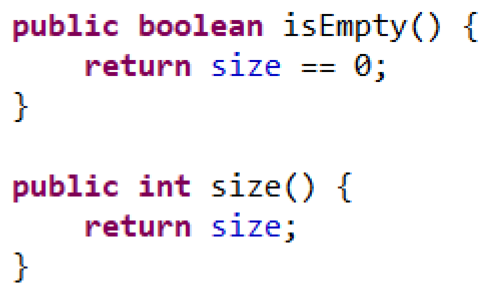
search(T element):



```java
public int search(T element) {
    Node<T> current = tail;
    for (int i = size - 1; i >= 0; i--) {
        if (current.data.equals(element)) {
            return i;
        }
        current = current.prev;
    }
    return -1;
}
```

- **Node<T> current = tail**: Starts traversal from the tail.
- **for (int i = size - 1; i >= 0; i--)**: Traverses the list from the last element to the first.
- **if (current.data.equals(element))**: Checks if the current element equals the specified element. If it does, it returns the index.
- **return -1**: If the element is not found, it returns -1.

isEmpty() and size():

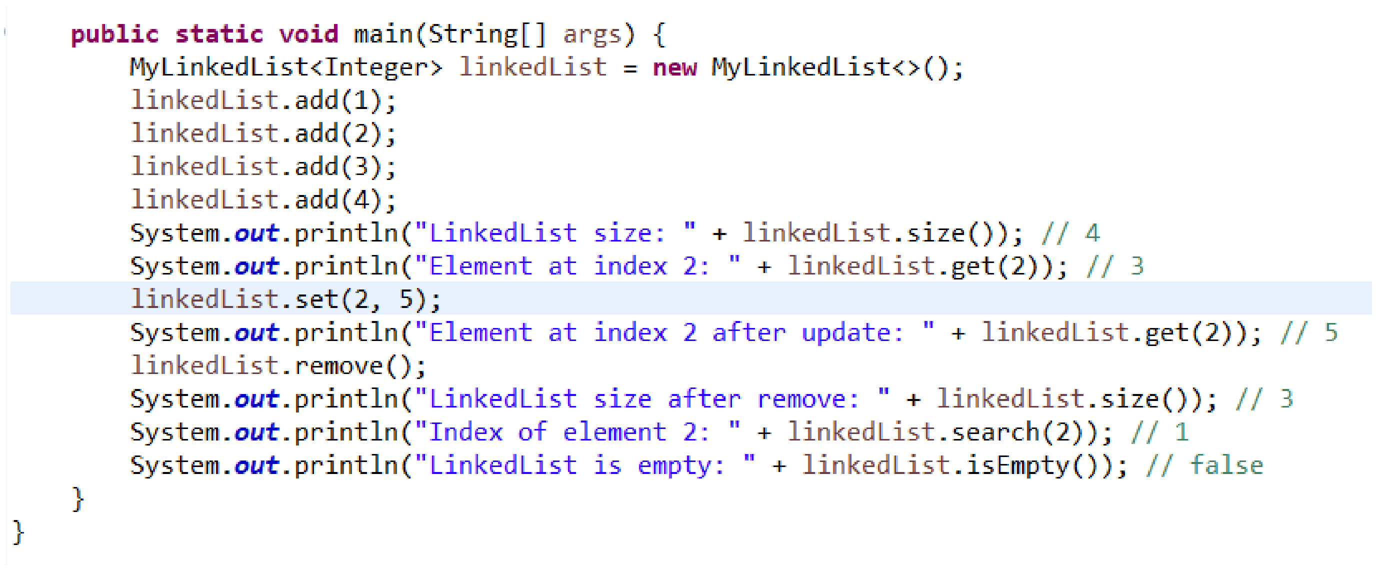


```java
public boolean isEmpty() {
    return size == 0;
}

public int size() {
    return size;
}
```

- **return size == 0**: Returns true if the list is empty, otherwise returns false.
- **return size**: Returns the number of elements in the list.

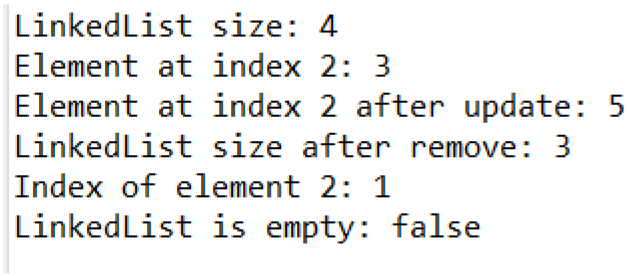
main():



```java
    public static void main(String[] args) {
        MyLinkedList<Integer> linkedList = new MyLinkedList<>();
        linkedList.add(1);
        linkedList.add(2);
        linkedList.add(3);
        linkedList.add(4);
        System.out.println("LinkedList size: " + linkedList.size()); // 4
        System.out.println("Element at index 2: " + linkedList.get(2)); // 3
        linkedList.set(2, 5);
        System.out.println("Element at index 2 after update: " + linkedList.get(2)); // 5
        linkedList.remove();
        System.out.println("LinkedList size after remove: " + linkedList.size()); // 3
        System.out.println("Index of element 2: " + linkedList.search(2)); // 1
        System.out.println("LinkedList is empty: " + linkedList.isEmpty()); // false
    }
}
```

- Creates a new **MyLinkedList** of **Integer** type and adds elements to it.
- Demonstrates the usage of **size(), get(), set(), remove(), search(),** and **isEmpty()** methods.

Result:



```
LinkedList size: 4
Element at index 2: 3
Element at index 2 after update: 5
LinkedList size after remove: 3
Index of element 2: 1
LinkedList is empty: false
```

Implement error handling and report test results (P5)

Testing plan

MyArrayList and MyLinkedList test case:

Test Case	Scope + Operation	Testing Type	Input	Expected Output	Actual Output	Status
1	Add elements to the list	Normal	Add elements [1, 2, 3, 4]	Size of the list is 4	Size of the list is 4	Passed
2	Add element to the list	Data validation	Add null element	NullPointerException or ignore null elements	NullPointerException or ignore null elements	Passed
3	Get element by index	Normal	Get element at index 2	3	3	Passed
4	Get element by index	Data validation	Get element at index 10	IndexOutOfBoundsException	IndexOutOfBoundsException	Passed
5	Update element by index	Normal	Set element at index 2 to 5	Element at index 2 is 5	Element at index 2 is 5	Passed
6	Update element by index	Data validation	Set element at index 10 to 5	IndexOutOfBoundsException	IndexOutOfBoundsException	Passed
7	Remove last element	Normal	Remove last element	Size of the list is 3	Size of the list is 3	Passed
8	Remove element from empty list	Data validation	Create empty list and remove element	IndexOutOfBoundsException	IndexOutOfBoundsException	Passed
9	Search for element	Normal	Search for element 2	Index of the last occurrence of element 2 is 1	Index of the last occurrence of element 2 is 1	Passed
10	Search for element	Data validation	Search for element not in list	-1	-1	Passed
11	Check if list is empty	Normal	Check if list is empty	false	false	Passed
12	Check if list is empty	Data validation	Check empty list	true	true	Passed

Based on the provided test cases:

- **Total test cases**: There are 24 test cases in total (12 for MyArrayList and 12 for MyLinkedList).
- **Failure cases**: There are 0 failure cases reported. All test cases passed successfully.

Since all test cases passed without any failures, there are no actual failure cases to analyze from the current tests. However, failure cases could arise due to several reasons:

- **NullPointerException**: This could occur if the implementation does not handle null elements properly, especially in methods like add or set.
- **IndexOutOfBoundsException**: This may happen if methods like get, set, or remove are not properly bound-checked against the size of the list.
- **Incorrect Element Update**: If updating an element does not reflect correctly in the list, it could indicate a problem with the internal indexing or linking mechanism in MyLinkedList.

To preemptively handle potential failure cases and improve the implementations:

1. **Null Handling**: Implement checks to handle null elements gracefully, either by throwing a NullPointerException or by ignoring such elements based on the intended behavior.
2. **Index Bounds Checking**: Ensure that all methods accessing elements by index (get, set, remove) check if the index is within valid bounds (0 to size-1), throwing an IndexOutOfBoundsException when necessary.
3. **Edge Case Testing**: Expand the test suite to include more edge cases such as adding to an empty list, removing from an empty list, updating elements at boundaries (first, last, middle), and searching for elements not present in the list.
4. **Concurrency Handling**: If the list is intended for concurrent access, consider adding synchronization mechanisms to ensure thread safety.
5. **Performance Considerations**: Evaluate and optimize operations like resizing in MyArrayList or traversing in MyLinkedList to ensure they perform efficiently under various loads and sizes of data.

By incorporating these improvements, the implementations can become more robust and reliable, better handling a wider range of scenarios and reducing the likelihood of failure cases during real-world usage.

Discuss how asymptotic analysis can be used to assess the effectiveness of an algorithm (P6)

Acording to GeeksforGeeks, 2023 symptotic analysis and asymptotic notation are fundamental concepts in algorithm analysis that help us understand how the performance of an algorithm scales with respect to input size. Here's a breakdown of the key points about asymptotic notation and its applications:

Asymptotic Notations

Asymptotic notation is a method for expressing an algorithm's running time or space complexity in relation to the input size. It is widely used in complexity analysis to illustrate how an algorithm behaves as the input size increases. The three most frequently used notations are Big O, Omega, and Theta. Acording to GeeksforGeeks, 2023 the author will extablish the big 3 of notation below:

Big-O Notation (O):

Definition: Big-O notation provides an upper bound on the asymptotic growth rate of an algorithm's time complexity (or space complexity). It represents the worst-case scenario, indicating the maximum time or space an algorithm may require to solve a problem as the input size nnn approaches infinity.

Formal Definition: O(g(n)) denotes a function f(n) if there exist constants c>0 and n0 such that f(n)<=c.g(n) for all n≥n0 (hakerearth, 2024).

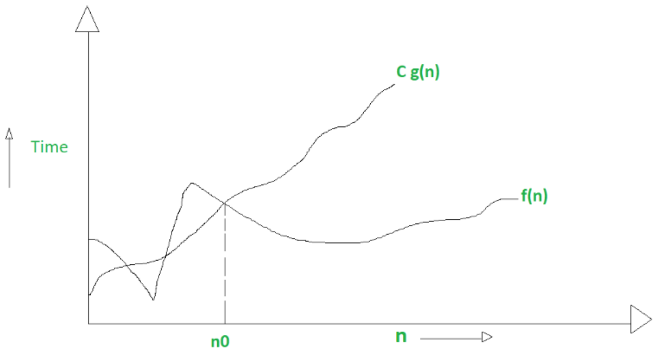
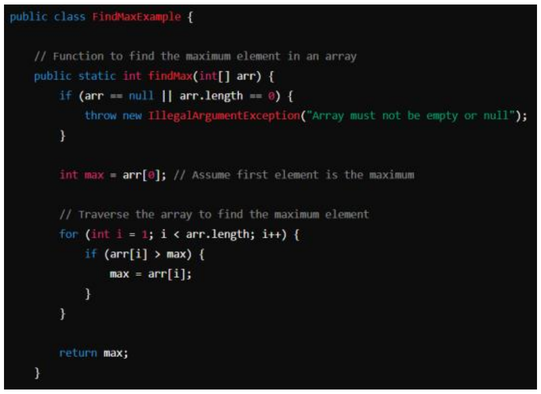


Figure 1. Big O (GeeksforGeeks, 2023)

Example:



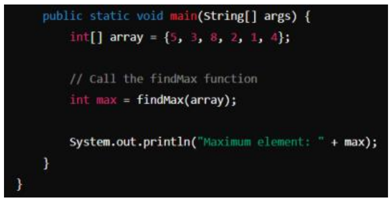
```java
public class FindMaxExample {

    // Function to find the maximum element in an array
    public static int findMax(int[] arr) {
        if (arr == null || arr.length == 0) {
            throw new IllegalArgumentException("Array must not be empty or null");
        }

        int max = arr[0]; // Assume first element is the maximum

        // Traverse the array to find the maximum element
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }

        return max;
    }
```



```java
    public static void main(String[] args) {
        int[] array = {5, 3, 8, 2, 1, 4};

        // Call the findMax function
        int max = findMax(array);

        System.out.println("Maximum element: " + max);
    }
}
```

Worst-case Time Complexity: In the worst case, the algorithm traverses the entire array of size n to find the maximum element. This results in n comparisons. To express the worst-case time complexity using Big-O notation:

Formal Definition: According to the formal definition of Big-O notation: O(g(n)) denotes a function f(n) if there exist constants c>0 and n0 such that f(n)<=c.g(n) for all n>=n0.

In the case of findMax function:

- Let f(n) be the time taken by findMax function to find the maximum element in an array of size nnn.
- Let g(n)=n, since in the worst case, the algorithm makes n comparisons.

Therefore, we can say:

- f(n) is O(n), which means the time complexity of findMax function is linear in terms of the size of the input array.

Conclusion:

- The findMax function has a worst-case time complexity of O(n), meaning its running time grows linearly with the size of the input array n.

Big-Omega Notation (Ω):

Definition: Big-Omega notation provides a lower bound on the asymptotic growth rate of an algorithm's time complexity (or space complexity). It represents the best-case scenario, indicating the minimum time or space an algorithm may require to solve a problem as the input size nnn approaches infinity.

Formal Definition: Ω(g(n)) denotes a function f(n) if there exist constants c>0 and n0 such that f(n)>=c.g(n) for all n>=n0.

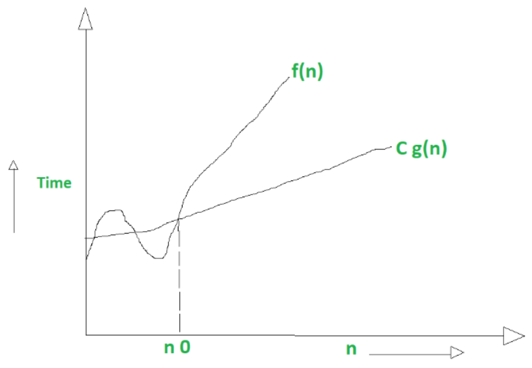
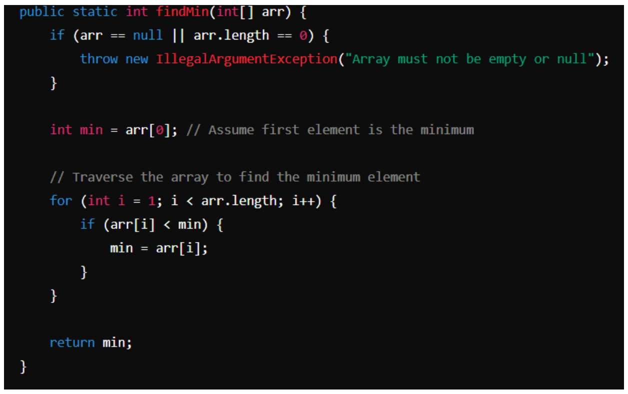


Figure 2. Big Ω (GeeksforGeeks, 2023)

Example:

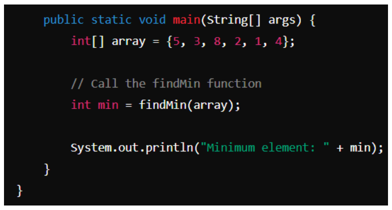


```
public static int findMin(int[] arr) {
    if (arr == null || arr.length == 0) {
        throw new IllegalArgumentException("Array must not be empty or null");
    }

    int min = arr[0]; // Assume first element is the minimum

    // Traverse the array to find the minimum element
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] < min) {
            min = arr[i];
        }
    }

    return min;
}
```



```
    public static void main(String[] args) {
        int[] array = {5, 3, 8, 2, 1, 4};

        // Call the findMin function
        int min = findMin(array);

        System.out.println("Minimum element: " + min);
    }
}
```

Best-case Time Complexity: In the best case, the algorithm finds the minimum element in the first comparison (assuming the first element is the minimum). This results in 1 comparison. To express the best-case time complexity using Big-Omega notation:

Formal Definition: According to the formal definition of Big-Omega notation: Ω(g(n)) denotes a function f(n) if there exist constants c>0 and n0 such that f(n)>=c.g(n) for all n>=n0 (hackerearth, 2024).

In the case of findMin function:

- Let f(n) be the time taken by findMin function to find the minimum element in an array of size n.
- Let g(n)=1, since in the best case, the algorithm makes 1 comparison.

Therefore, we can say:

- f(n) is Ω(1), which means the time complexity of findMin function is constant in the best case.

Conclusion:

- The findMin function has a best-case time complexity of Ω(1), meaning its running time is constant regardless of the size of the input array n.

Big-Theta Notation (Θ):

Definition: Big-Theta notation provides both upper and lower bounds on the asymptotic growth rate of an algorithm's time complexity (or space complexity). It indicates that the algorithm's performance is asymptotically tight within this range as the input size nnn approaches infinity.

Formal Definition: Θ(g(n) denotes a function f(n) if there exist constants c1>0, c2>0, and n0 such that c1.g(n)<=f(n)<=c2.g(n) for all n>=n0 (hackerearth, 2024).

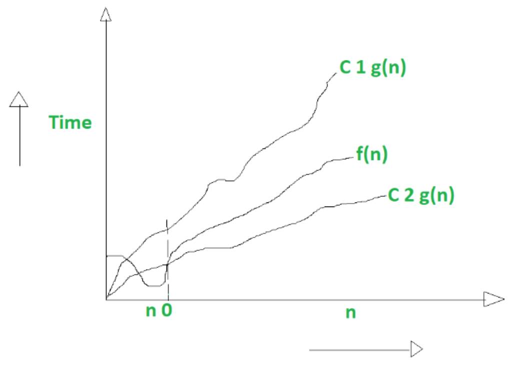
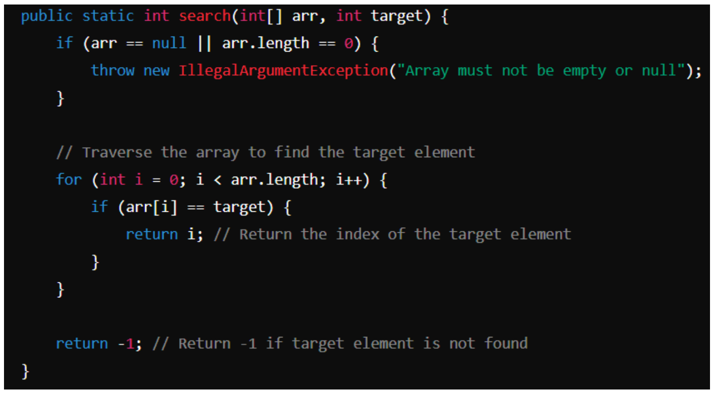


Figure 2. Big Θ (GeeksforGeeks, 2023)

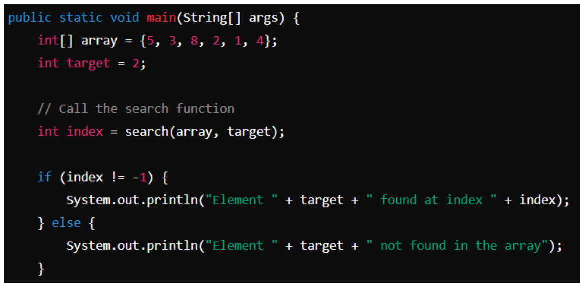
Example:



```
public static int search(int[] arr, int target) {
    if (arr == null || arr.length == 0) {
        throw new IllegalArgumentException("Array must not be empty or null");
    }

    // Traverse the array to find the target element
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) {
            return i; // Return the index of the target element
        }
    }

    return -1; // Return -1 if target element is not found
}
```



```
public static void main(String[] args) {
    int[] array = {5, 3, 8, 2, 1, 4};
    int target = 2;

    // Call the search function
    int index = search(array, target);

    if (index != -1) {
        System.out.println("Element " + target + " found at index " + index);
    } else {
        System.out.println("Element " + target + " not found in the array");
    }
```

Average-case Time Complexity: In the average case, the algorithm may find the target element anywhere in the array, requiring n/2 comparisons on average, where nnn is the size of the array.

To express the average-case time complexity using Big-Theta notation:

Formal Definition: According to the formal definition of Big-Theta notation: Θ(g(n)) denotes a function f(n) if there exist constants c1>0, c2>0, and n0 such that c1⋅.g(n)<=f(n)<=c2.g(n) for all n>=n0.

In the case of search function:

- Let f(n) be the time taken by search function to find the target element in an array of size nnn.
- Let g(n)=n, since in the average case, the algorithm may perform n/2 comparisons on average.

Therefore, we can say:

- f(n) is Θ(n), which means the time complexity of search function is linear in the average case.

Conclusion:

- The search function has an average-case time complexity of Θ(n), meaning its running time grows linearly with the size of the input array n on average.

Determine two ways in which the efficiency of an algorithm can be measured, illustrating your answer with an example (P7)

There are often multiple ways to solve a problem, so it's important to learn how to compare the performance of different algorithms and select the best one. When analyzing an algorithm, we primarily focus on time complexity and space complexity. Time complexity measures the amount of time an algorithm takes to run as a function of the input length, while space complexity measures the amount of memory an algorithm requires based on the input length. Although time and space complexity can be influenced by factors such as hardware, operating system, and processors, we disregard these aspects during algorithm analysis. Instead, we concentrate solely on the algorithm's execution time (hackerearth, 2024).

Time Complexity

The time complexity of an algorithm measures the amount of time it takes for the algorithm to run as a function of the input size. It is important to note that this measurement is based on the input length, not the actual execution time on a specific machine. A valid algorithm completes its execution within a finite amount of time. The time complexity of an algorithm refers to the time required to solve a given problem. It is a crucial metric in algorithm analysis. Time complexity represents the duration needed for an algorithm to finish. To estimate it, we consider the cost of each fundamental operation and the number of times each operation is executed (GeeksforGeeks, 2023).

Space Complexity

Acording to GeeksforGeeks, 2023 problem-solving using a computer requires memory to store temporary data or the final result while a program is running. The amount of memory an algorithm needs to solve a given problem is called its space complexity. The space complexity of an algorithm measures the amount of space it uses as a function of the input size. For example, consider a problem that involves finding the frequency of elements in an array. Space complexity represents the memory needed for an algorithm to complete its execution. To estimate the memory requirement, we focus on two parts:

Fixed part: This is independent of the input size and includes memory for instructions (code), constants, and variables.

Variable part: This depends on the input size and includes memory for the recursion stack and referenced variables.

Example

Concept:

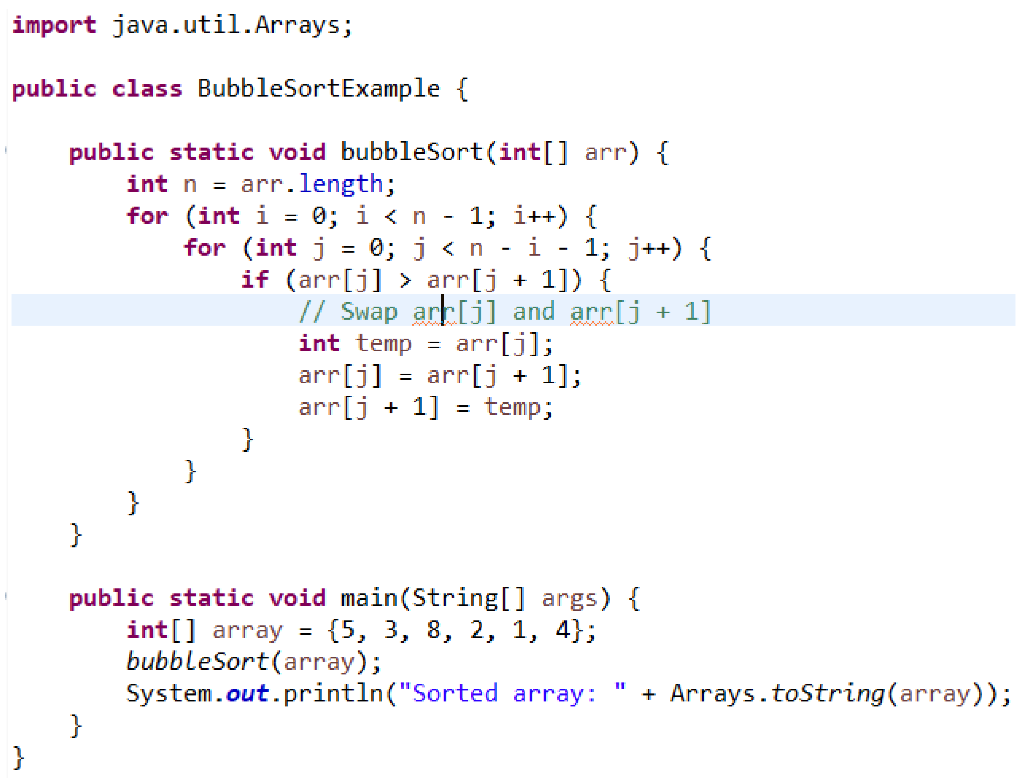
Bubble Sort is a straightforward sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Algorithm Steps:

1. Traverse the entire array.
2. For each element, compare it with the next element.
3. If the current element is greater than the next element, swap them.
4. Repeat this process for the rest of the array, reducing the number of elements to compare by one each time since the largest element will have "bubbled" to the end of the array.
5. Continue until no more swaps are needed.

Example:

Consider an array arr[] with elements sorted in reverse order.



```java
import java.util.Arrays;

public class BubbleSortExample {

    public static 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;
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] array = {5, 3, 8, 2, 1, 4};
        bubbleSort(array);
        System.out.println("Sorted array: " + Arrays.toString(array));
    }
}
```

Time complexity:

Initialization:

- The algorithm initializes by setting n as the length of the array.

Outer Loop:

- The outer loop runs from i = 0 to i < n - 1. This loop ensures that the process is repeated until the array is sorted. The number of passes required to sort the array is n-1.

Inner Loop:

- The inner loop runs from j = 0 to j < n - i - 1. This loop compares each pair of adjacent elements and swaps them if they are in the wrong order.

Worst-case Scenario:

- In the worst-case scenario, the array is sorted in reverse order.
- Each round requires fewer comparisons:
  - Round 1: i = 1 -> n-1 comparisons
  - Round 2: i = 2 -> n-2 comparisons
  - Round 3: i = 3 -> n-3 comparisons
  - ...
  - Round n-1: i = n-1 -> 1 comparison
- The total number of comparisons is the sum of the first n-1 natural number

$$\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$$

Time Complexity:

- **Worst-case Time Complexity**: O(n^2)
- **Best-case Time Complexity**: O(n) (when the array is already sorted)
- **Average-case Time Complexity**: O(n^2)

Illustration: Let's visualize the steps of the bubble sort algorithm on an array sorted in reverse order.

[5, 3, 8, 2, 1, 4]

First Pass:

- Compare 5 and 3 -> Swap
- Compare 5 and 8 -> No Swap
- Compare 8 and 2 -> Swap
- Compare 8 and 1 -> Swap
- Compare 8 and 4 -> Swap

[3, 5, 2, 1, 4, 8]

Second Pass:

- Compare 3 and 5 -> No Swap
- Compare 5 and 2 -> Swap
- Compare 5 and 1 -> Swap
- Compare 5 and 4 -> Swap

[3, 2, 1, 4, 5, 8]

Third Pass:

- Compare 3 and 2 -> Swap
- Compare 3 and 1 -> Swap
- Compare 3 and 4 -> No Swap

[2, 1, 3, 4, 5, 8]

Fourth Pass:

- Compare 2 and 1 -> Swap
- Compare 2 and 3 -> No Swap

[1, 2, 3, 4, 5, 8]

After these passes, the array is sorted.

Space complexity:

Bubble Sort operates directly on the input array and uses a constant amount of additional space. Bubble Sort requires a constant amount of additional space, making its space complexity O(1). The algorithm operates directly on the input array without needing additional storage proportional to the input size.

Analysis:

1. **Input Space**: O(n) - The array arr requires space proportional to its length n.
2. **Auxiliary Space**: O(1) - Bubble Sort uses a fixed amount of additional space, regardless of the input size.
    - **Variables**: n, i, j, temp - These variables occupy a constant amount of space.

Therefore, the total space complexity of Bubble Sort is O(n) + O(1) = O(n), but the **auxiliary space complexity** is O(1).

Conclusion:

- **Time Complexity**: Measures the growth of the number of operations as the input size increases. For Bubble Sort, it is O(n^2) in the worst and average cases.
- **Space Complexity**: Measures the growth of memory usage as the input size increases. For Bubble Sort, it is O(1) since no additional space proportional to the input size is needed.

By understanding these complexities, we can compare and analyze the efficiency of different algorithms, making informed decisions about which algorithm is best suited for a given problem based on input size and resource constraints.

Conclusion.

In this report, we have successfully addressed the algorithmic problem, completed the implementation and testing, and documented the testing results. The report also includes a study on asymptotic analysis, which can be utilized to assess the efficiency of an algorithm.

References.

What is an ArrayList in Java? *(no date)* Educative. *Available at: https://www.educative.io/answers/what-is-an-arraylist-in-java (Accessed: 30 June 2024).*

Simplilearn (2024)* Linked List in Java: All You Need to Know About it, Simplilearn.com. *Simplilearn. Available at: https://www.simplilearn.com/tutorials/java-tutorial/linked-list-in-java#:~:text=A%20linked%20list%20in%20Java%20is%20a%20dynamic%20data%20structure,link%20to%20the%20first%20container. (Accessed: 30 June 2024).

(No date). Available at: https://www.cs.cmu.edu/~mrmiller/15-121/Slides/09-BigO-ArrayList.pdf (Accessed: 30 June 2024).

baeldung, W. by: (2024)* Time Complexity of Java Collections, Baeldung. *Available at: https://www.baeldung.com/java-collections-complexity (Accessed: 30 June 2024).

GeeksforGeeks (2023)* Difference between Singly linked list and Doubly linked list, GeeksforGeeks. *GeeksforGeeks. Available at: https://www.geeksforgeeks.org/difference-between-singly-linked-list-and-doubly-linked-list/ (Accessed: 30 June 2024).

Asymptotic Notation and Analysis (Based on input size) in Complexity Analysis of Algorithms *(2023)* GeeksforGeeks. *Available at: https://www.geeksforgeeks.org/asymptotic-notation-and-analysis-based-on-input-size-of-algorithms/ (Accessed: 30 June 2024).*

GeeksforGeeks (2023)* Difference between Big O vs Big Theta Θ vs Big Omega Ω Notations, GeeksforGeeks. *GeeksforGeeks. Available at: https://www.geeksforgeeks.org/difference-between-big-oh-big-omega-and-big-theta/ (Accessed: 30 June 2024).

GeeksforGeeks (2023)* Time Complexity and Space Complexity, GeeksforGeeks. *GeeksforGeeks. Available at: https://www.geeksforgeeks.org/time-complexity-and-space-complexity/ (Accessed: 30 June 2024).

Time and Space Complexity Tutorials & Notes: Basic Programming *(no date)* HackerEarth. *Available at: https://www.hackerearth.com/practice/basic-programming/complexity-analysis/time-and-space-complexity/tutorial/#:~:text=Time%20complexity%20of%20an%20algorithm,the%20length%20of%20the%20input. (Accessed: 30 June 2024).*