Linear Data Structures in Java

Table of Contents

Arrays

Static Arrays

Definition: A static array is a collection of elements of the same type, stored in a contiguous block of memory. Once the size is defined, it cannot be changed.

Characteristics:

- **Fixed Size**: The size is specified at the time of creation and remains constant.
- **Contiguous Memory**: Elements are stored next to each other in memory.
- **Index-Based Access**: Access elements using indices (e.g., `array[0]`).

Example:

```
public class StaticArrayExample {
    public static void main(String[] args) {
        int[] numbers = new int[5]; // Array of size 5
        numbers[0] = 10;
        numbers[1] = 20;
        numbers[2] = 30;
        numbers[3] = 40;
        numbers[4] = 50;

        // Print all elements
        for (int i = 0; i < numbers.length; i++) {
            System.out.println(numbers[i]);
        }
```

```
    }
}
```

Time Complexity:

Operation	Time Complexity
Access	O(1)
Insertion	O(n)
Deletion	O(n)

Advantages:

- Fast access to elements using indices.
- Memory-efficient for fixed-size collections.

Limitations:

- Fixed size may lead to wasted space or insufficient space.
- Resizing requires creating a new array and copying elements.

Dynamic Arrays

Definition: Dynamic arrays can automatically resize themselves when elements are added or removed. In Java, `ArrayList` is a common implementation of a dynamic array.

Characteristics:

- **Resizing**: Automatically grows as needed, typically doubling in size.
- **Additional Methods**: Provides methods to add, remove, and manipulate elements.

Example:

```java
import java.util.ArrayList;

public class DynamicArrayExample {
    public static void main(String[] args) {
        ArrayList<Integer> numbers = new ArrayList<>();
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);

        // Print all elements
        for (Integer number : numbers) {
            System.out.println(number);
        }
    }
}
```

Growth Factor: Dynamic arrays typically resize by increasing the capacity by a factor (e.g., doubling the size). This ensures that most insertions are O(1) on average.

Time Complexity:

Operation	Time Complexity (Amortized)
Access	O(1)
Insertion	O(1) (Amortized)
Deletion	O(n)

Advantages:

- Flexible size, grows automatically.
- Provides built-in methods for manipulation.

Limitations:

- Overhead of resizing and copying elements.
- Additional memory usage due to capacity management.

Linked Lists

- Singly Linked List

Definition: A singly linked list consists of nodes where each node points to the next node in the sequence.

Characteristics:

- **Single Link**: Each node contains a reference to the next node.
- **Dynamic Size**: Can grow and shrink as needed.

Example:

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

public class SinglyLinkedList {
    Node head;

    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node temp = head;
```

```
            while (temp.next != null) temp = temp.next;
            temp.next = newNode;
        }
    }

    public void printList() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.printList();  // Output: 10 20 30
    }
}
```

Time Complexity:

Operation	Time Complexity
Access	O(n)
Insertion	O(1) (at beginning)
Deletion	O(1) (at beginning)

Advantages:

- Dynamic size, no fixed limits.
- Efficient insertion and deletion.

Limitations:

- Slower access time (O(n)) compared to arrays.
- Extra memory for pointers.

- Doubly Linked List

Definition: A doubly linked list allows traversal in both directions as each node has two pointers: one to the next node and one to the previous node.

Characteristics:

- **Bidirectional Links**: Each node contains references to both the next and previous nodes.

- **Dynamic Size**: Like singly linked lists, can grow and shrink as needed.

Example:

```
class Node {
    int data;
    Node next, prev;
    Node(int data) { this.data = data; this.next = this.prev = null; }
}

public class DoublyLinkedList {
    Node head;

    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node temp = head;
            while (temp.next != null) temp = temp.next;
            temp.next = newNode;
            newNode.prev = temp;
        }
    }

    public void printList() {
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        DoublyLinkedList list = new DoublyLinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.printList();  // Output: 10 20 30
    }
}
```

Difference Between Singly and Doubly Linked List:

Feature	Singly Linked List	Doubly Linked List
Direction of Traversal	One direction (forward)	Two directions (forward & back)
Memory Usage	Less (only one pointer)	More (two pointers)

Feature	Singly Linked List	Doubly Linked List
Efficiency of Operations	Insertion and deletion slower	Faster for some operations
Use Cases	Simple traversal and operations	Complex operations requiring bidirectional traversal

- Circular Linked List

Definition: A circular linked list is similar to a singly or doubly linked list but the last node points back to the first node, forming a loop.

Characteristics:

- **Circular Reference**: The last node points back to the first node.
- **Traversal**: Can traverse the list repeatedly without reaching the end.

Example:

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

public class CircularLinkedList {
    Node head;

    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            head.next = head;
        } else {
            Node temp = head;
            while (temp.next != head) temp = temp.next;
            temp.next = newNode;
            newNode.next = head;
        }
    }

    public void printList() {
        if (head == null) return;
        Node temp = head;
        do {
            System.out.print(temp.data + " ");
            temp = temp.next;
        } while (temp != head);
        System.out.println();
    }
```

```
    public static void main(String[] args) {
        CircularLinkedList list = new CircularLinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.printList();  // Output: 10 20 30
    }
}
```

Advantages:

- Useful for circular processes, like round-robin scheduling.
- Eliminates the need to check for the end of the list.

Disadvantages:

- More complex to implement and understand.
- Extra logic needed to handle circular references.

Stacks

Implementation Using Arrays

Definition: A stack follows the Last-In-First-Out (LIFO) principle. Operations are performed at one end called the top of the stack.

Operations:

- `push`: Add an item to the top of the stack.
- `pop`: Remove the item from the top of the stack.
- `peek`: View the item at the top without removing it.

Example:

```
class Stack {
    int top;
    int[] stack = new int[5];

    Stack() { top = -1; }

    public void push(int data) {
        if (top == stack.length - 1) {
            System.out.println("Stack Overflow");
        } else {
            stack[++top] = data;
        }
    }

    public int pop() {
```

```
        if (top == -1) {
            System.out.println("Stack Underflow");
            return -1;
        } else {
            return stack[top--];
        }
    }

    public int peek() {
        if (top == -

1) {
            System.out.println("Stack is Empty");
            return -1;
        } else {
            return stack[top];
        }
    }
}

public class StackExample {
    public static void main(String[] args) {
        Stack stack = new Stack();
        stack.push(10);
        stack.push(20);
        System.out.println(stack.pop()); // Output: 20
        System.out.println(stack.peek()); // Output: 10
    }
}
```

Time Complexity:

Operation	Time Complexity
Push	O(1)
Pop	O(1)
Peek	O(1)

Advantages:

- Simple and efficient.
- Fast operations with constant time complexity.

Limitations:

- Fixed size in array-based implementation.
- Limited flexibility compared to dynamic implementations.

Implementation Using Linked Lists

Definition: A stack can also be implemented using a singly linked list where operations are performed at one end.

Operations:

- `push`: Add an item to the top of the stack.
- `pop`: Remove the item from the top of the stack.
- `peek`: View the item at the top without removing it.

Example:

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

class Stack {
    Node top;

    public void push(int data) {
        Node newNode = new Node(data);
        newNode.next = top;
        top = newNode;
    }

    public int pop() {
        if (top == null) {
            System.out.println("Stack Underflow");
            return -1;
        } else {
            int data = top.data;
            top = top.next;
            return data;
        }
    }

    public int peek() {
        if (top == null) {
            System.out.println("Stack is Empty");
            return -1;
        } else {
            return top.data;
        }
    }
}

public class StackExample {
    public static void main(String[] args) {
        Stack stack = new Stack();
        stack.push(10);
        stack.push(20);
```

```
        System.out.println(stack.pop()); // Output: 20
        System.out.println(stack.peek()); // Output: 10
    }
}
```

Advantages:

- Dynamic size, grows as needed.
- No fixed size limitation.

Disadvantages:

- Additional memory usage for pointers.
- Slightly more complex than array-based implementation.

Queues

Implementation Using Arrays

Definition: A queue follows the First-In-First-Out (FIFO) principle. Elements are added at the rear and removed from the front.

Operations:

- `enqueue`: Add an item to the rear of the queue.
- `dequeue`: Remove an item from the front of the queue.
- `peek`: View the item at the front without removing it.

Example:

```
class Queue {
    int front, rear, size;
    int[] queue;

    Queue(int capacity) {
        queue = new int[capacity];
        front = 0;
        rear = -1;
        size = 0;
    }

    public void enqueue(int data) {
        if (size == queue.length) {
            System.out.println("Queue Overflow");
        } else {
            rear = (rear + 1) % queue.length;
            queue[rear] = data;
            size++;
        }
    }
```

```
    public int dequeue() {
        if (size == 0) {
            System.out.println("Queue Underflow");
            return -1;
        } else {
            int data = queue[front];
            front = (front + 1) % queue.length;
            size--;
            return data;
        }
    }

    public int peek() {
        if (size == 0) {
            System.out.println("Queue is Empty");
            return -1;
        } else {
            return queue[front];
        }
    }
}

public class QueueExample {
    public static void main(String[] args) {
        Queue queue = new Queue(5);
        queue.enqueue(10);
        queue.enqueue(20);
        System.out.println(queue.dequeue()); // Output: 10
        System.out.println(queue.peek()); // Output: 20
    }
}
```

Time Complexity:

Operation	Time Complexity
Enqueue	O(1)
Dequeue	O(1)
Peek	O(1)

Advantages:

- Simple and efficient.
- Constant time complexity for basic operations.

Limitations:

- Fixed size in array-based implementation.
- Resizing is not straightforward.

Implementation Using Linked Lists

Definition: A queue can also be implemented using a singly linked list where operations are performed at both ends.

Operations:

- `enqueue`: Add an item to the rear of the queue.
- `dequeue`: Remove an item from the front of the queue.
- `peek`: View the item at the front without removing it.

Example:

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

class Queue {
    Node front, rear;

    Queue() {
        front = rear = null;
    }

    public void enqueue(int data) {
        Node newNode = new Node(data);
        if (rear == null) {
            front = rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }
    }

    public int dequeue() {
        if (front == null) {
            System.out.println("Queue Underflow");
            return -1;
        } else {
            int data = front.data;
            front = front.next;
            if (front == null) rear = null;
            return data;
        }
    }

    public int peek() {
        if (front == null) {
            System.out.println("Queue is Empty");
            return -1;
```

```
            } else {
                return front.data;
            }
        }
    }

    public class QueueExample {
        public static void main(String[] args) {
            Queue queue = new Queue();
            queue.enqueue(10);
            queue.enqueue(20);
            System.out.println(queue.dequeue()); // Output: 10
            System.out.println(queue.peek()); // Output: 20
        }
    }
```

Advantages:

- Dynamic size, grows as needed.
- Efficient operations with linked list flexibility.

Disadvantages:

- Extra memory for pointers.
- Slightly more complex than array-based implementation.

Practice Programs

1. **Static Array Manipulations**: Write programs to perform common operations like insertion, deletion, searching, and updating elements in static arrays.
2. **Dynamic Array Manipulations**: Use `ArrayList` to practice adding, removing, and manipulating elements dynamically.
3. **Singly Linked List Operations**: Implement operations such as insertion, deletion, and traversal in a singly linked list.
4. **Doubly Linked List Operations**: Write programs to insert, delete, and traverse nodes in both directions in a doubly linked list.
5. **Circular Linked List Operations**: Practice inserting and traversing nodes in a circular linked list where the last node points back to the first node.
6. **Stack Operations**: Implement stack operations (push, pop, peek) using both arrays and linked lists.
7. **Queue Operations**: Implement queue operations (enqueue, dequeue, peek) using both arrays and linked lists.
