STACKS, QUEUES AND DEQUES

DATA STRUCTURES IN JAVA

Sercan Külcü | Data Structures In Java | 10.05.2023

Contents

Stacks

In this chapter, we'll be discussing one of the fundamental data structures in computer science: the stack. A stack is a collection of elements that supports two primary operations: push and pop. The push operation adds an element to the top of the stack, while the pop operation removes the top element from the stack.

One of the key characteristics of a stack is that it follows a Last-In, First-Out (LIFO) ordering. This means that the most recently added element is always the first one to be removed. To visualize this, you can think of a stack as a pile of books or plates, where you can only add or remove items from the top of the pile.

In Java, you can implement a stack using an array or a linked list. Let's take a look at an example implementation using an array:

```
public class Stack {

  private int[] array;

  private int top;


  public Stack(int size) {

    array = new int[size];

    top = -1;

  }


  public void push(int element) {

    if (top == array.length - 1) {

      throw new IllegalStateException("Stack overflow");
    }
    top++;
    array[top] = element;
}

public int pop() {
    if (top == -1) {
        throw new IllegalStateException("Stack underflow");
    }
    int element = array[top];
    top--;
    return element;
}

public int peek() {
    if (top == -1) {
        throw new IllegalStateException("Stack is empty");
    }
    return array[top];
}

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

In this implementation, we're using an array to store the elements of the stack. The top variable keeps track of the index of the top element in the stack. The push method adds an element to the top of the stack by incrementing the top variable and assigning the element to the corresponding index in the array. The pop method removes the top element from the stack by returning the value of the element at the top index and then decrementing the top variable. The peek method returns the value of the top element without removing it, and the isEmpty method checks if the stack is empty.

Stacks are widely used in computer science and software engineering. They are used in programming languages to implement function calls, in web browsers to store the history of visited pages, and in operating systems to store information about processes and their state.

In summary, a stack is a fundamental data structure that follows a Last-In, First-Out (LIFO) ordering. Stacks can be implemented using an array or a linked list in Java, and are useful for a wide range of applications in computer science and software engineering. By understanding how to create and use stacks in Java, you can write more efficient and powerful programs that can solve a wide range of problems.

Queues

In this chapter, we'll be discussing another important data structure in computer science: the queue. A queue is a collection of elements that supports two primary operations: enqueue and dequeue. The enqueue operation adds an element to the back of the queue, while the dequeue operation removes the element from the front of the queue.

One of the key characteristics of a queue is that it follows a First-In, First-Out (FIFO) ordering. This means that the first element added to the queue is always the first one to be removed. To visualize this, you can think of a queue as a line of people waiting for a bus or a rollercoaster, where the person who arrived first is the first one to board.

In Java, you can implement a queue using an array or a linked list. Let's take a look at an example implementation using a linked list:

```
public class Queue {
    private Node front;
    private Node rear;

    public Queue() {
        front = null;
        rear = null;
    }

    public void enqueue(int element) {
        Node newNode = new Node(element);
        if (rear == null) {
```

```
        front = newNode;
    } else {
        rear.next = newNode;
    }
    rear = newNode;
}

public int dequeue() {
    if (front == null) {
        throw new IllegalStateException("Queue underflow");
    }
    int element = front.data;
    front = front.next;
    if (front == null) {
        rear = null;
    }
    return element;
}

public int peek() {
    if (front == null) {
        throw new IllegalStateException("Queue is empty");
        }
        return front.data;
    }

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

    private static class Node {
        int data;
        Node next;

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

In this implementation, we're using a linked list to store the elements of the queue. The front variable keeps track of the front of the queue, while the rear variable keeps track of the back of the queue. The enqueue method adds an element to the back of the queue by creating a new node and setting it as the next node of the rear node. If the queue is empty, the new node becomes the front node. The dequeue method removes the element from the front of the queue by returning the value of the front node's data and updating the front node to its next node. If the front node becomes null, then the queue is empty and the rear node is set to null as well. The peek method returns the value of the front element without removing it, and the isEmpty method checks if the queue is empty.

Queues are widely used in computer science and software engineering. They are used in operating systems to manage processes and their priority levels, in networking to handle packet traffic, and in video and audio streaming to buffer data.

In summary, a queue is a fundamental data structure that follows a First-In, First-Out (FIFO) ordering. Queues can be implemented using an array or a linked list in Java, and are useful for a wide range of applications in computer science and software engineering. By understanding how to create and use queues in Java, you can write more efficient and powerful programs that can solve a wide range of problems.

Deques

In this chapter, we'll be discussing another type of data structure: the deque. A deque, short for double-ended queue, is a data structure that supports insertion and deletion of elements from both ends of the queue. This means that you can add elements to the front or the back of the deque, and remove elements from the front or the back as well.

In Java, you can implement a deque using an array or a linked list. Let's take a look at an example implementation using a linked list:

```
public class Deque {

    private Node front;

    private Node rear;


    public Deque() {

        front = null;

        rear = null;

    }


    public void addFront(int element) {

        Node newNode = new Node(element);

        if (front == null) {

            rear = newNode;

        } else {

            newNode.next = front;
        }
        front = newNode;
    }

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

    public int removeFront() {
        if (front == null) {
            throw new IllegalStateException("Deque underflow");
        }
        int element = front.data;
        front = front.next;
        if (front == null) {
            rear = null;
        }
        return element;
    }

    public int removeRear() {
        if (rear == null) {
            throw new IllegalStateException("Deque underflow");
        }
        int element = rear.data;
        if (front == rear) {
            front = null;
            rear = null;
        } else {
            Node current = front;
            while (current.next != rear) {
                current = current.next;
            }
            current.next = null;
            rear = current;
        }
        return element;
    }
```

```
public int peekFront() {
    if (front == null) {
        throw new IllegalStateException("Deque is empty");
    }
    return front.data;
}

public int peekRear() {
    if (rear == null) {
        throw new IllegalStateException("Deque is empty");
    }
    return rear.data;
}

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

private static class Node {
    int data;
    Node next;
```

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

In this implementation, we're using a linked list to store the elements of the deque. The front variable keeps track of the front of the deque, while the rear variable keeps track of the back of the deque. The addFront method adds an element to the front of the deque by creating a new node and setting it as the next node of the current front node. If the deque is empty, the new node becomes both the front and rear node. The addRear method adds an element to the back of the deque in a similar fashion, creating a new node and setting it as the next node of the current rear node. If the deque is empty, the new node becomes both the front and rear node. The removeFront method removes the element from the front of the deque by returning the value of the front node's data and updating the front node to its next node. If the front node becomes null, then the deque is empty and the rear node is set to null as well. The removeRear method removes the element from the back of the deque by returning the value of the rear node's data and updating the rear node to current node.