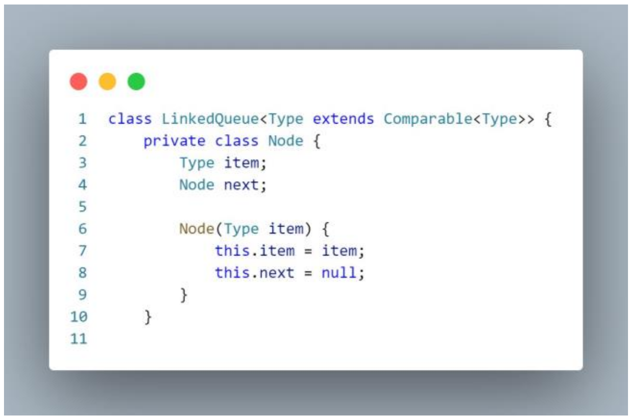
Dequeue

LinkedQueue Class

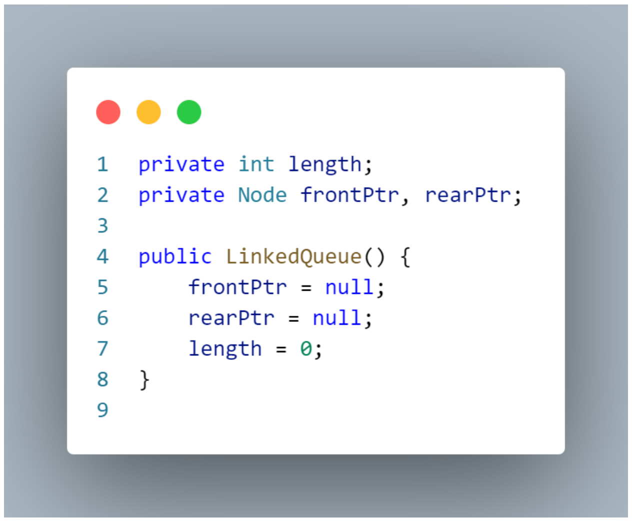


```java
class LinkedQueue<Type extends Comparable<Type>> {
    private class Node {
        Type item;
        Node next;

        Node(Type item) {
            this.item = item;
            this.next = null;
        }
    }

```

Node Class

- Purpose: Inner class representing a node in the linked list.
- Attributes:item: Stores the data of generic type.next: Reference to the next node in the linked list.

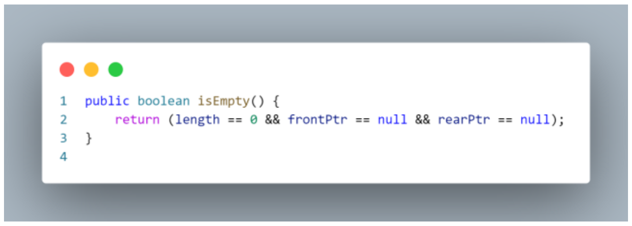
Constructor LinkedQueue()



```java
private int length;
private Node frontPtr, rearPtr;

public LinkedQueue() {
    frontPtr = null;
    rearPtr = null;
    length = 0;
}

```

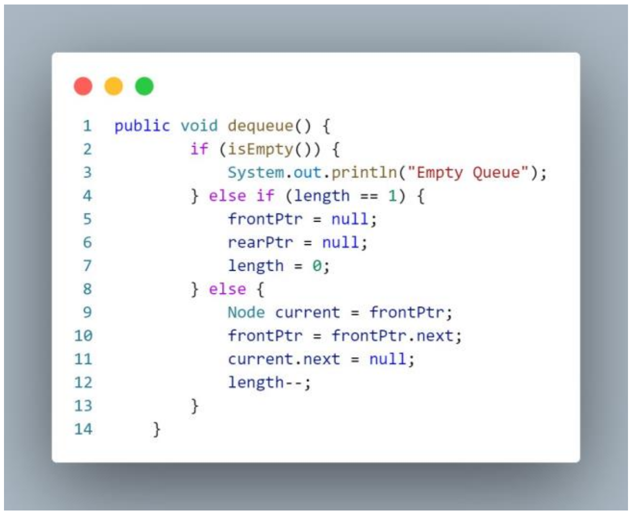
- Purpose: Initializes an empty double-ended queue.
- Behavior:Sets frontPtr and rearPtr to null.Initializes length to 0.

isEmpty()


```java
public boolean isEmpty() {
    return (length == 0 && frontPtr == null && rearPtr == null);
}
```

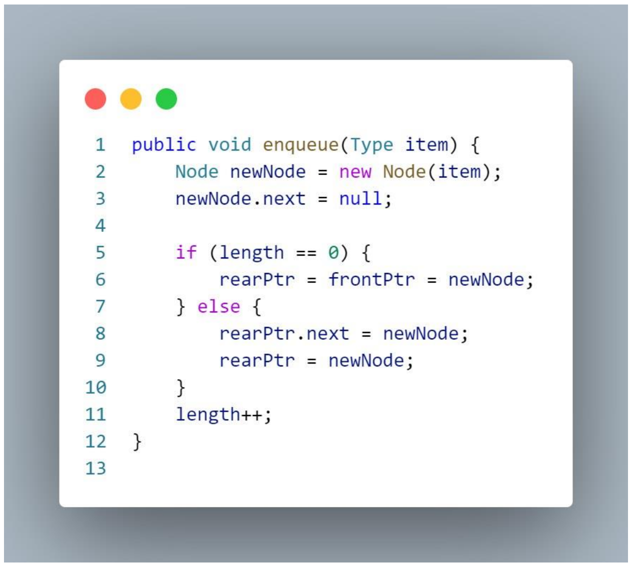
- Purpose: Checks if the queue is empty.
- Returns: true if the queue is empty, false otherwise.

dequeue()


```java
public void dequeue() {
        if (isEmpty()) {
            System.out.println("Empty Queue");
        } else if (length == 1) {
            frontPtr = null;
            rearPtr = null;
            length = 0;
        } else {
            Node current = frontPtr;
            frontPtr = frontPtr.next;
            current.next = null;
            length--;
        }
    }
```

- Purpose: Removes an element from the front of the queue.
- Behavior:Removes the front element by updating frontPtr and length.Handles the case when the queue has only one element or no elements.

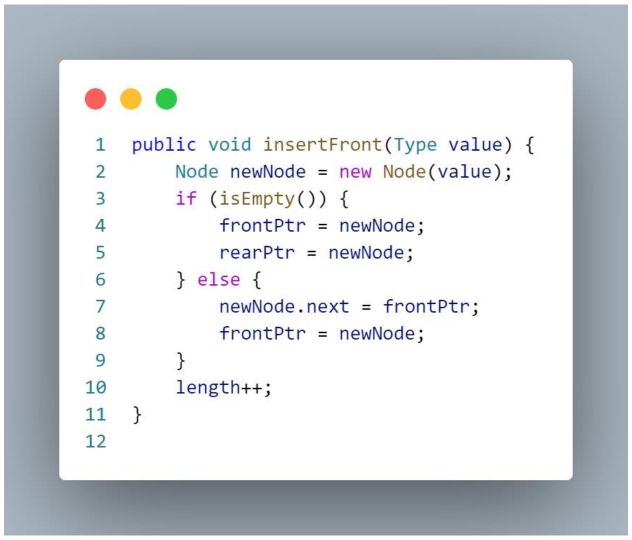
enqueue(Type item)



```java
public void enqueue(Type item) {
    Node newNode = new Node(item);
    newNode.next = null;

    if (length == 0) {
        rearPtr = frontPtr = newNode;
    } else {
        rearPtr.next = newNode;
        rearPtr = newNode;
    }
    length++;
}
```

- Purpose: Adds an element to the rear of the queue.
- Parameters: item - Element to be added.
- Behavior:Creates a new node with the given item.Handles cases when the queue is empty or non-empty.

insertFront(Type value)

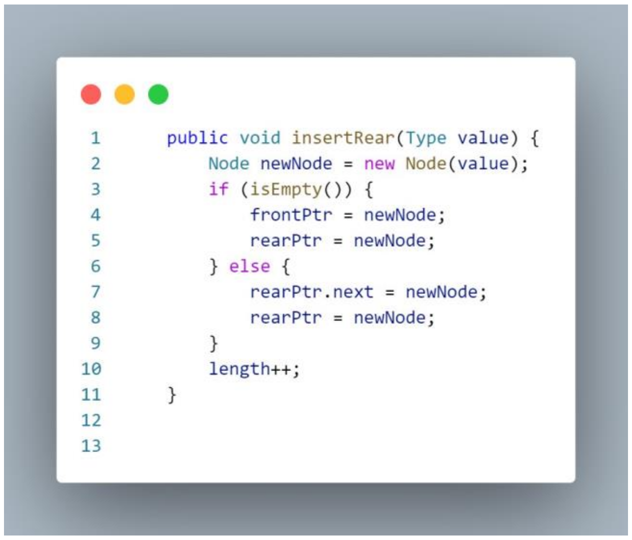


```java
public void insertFront(Type value) {
    Node newNode = new Node(value);
    if (isEmpty()) {
        frontPtr = newNode;
        rearPtr = newNode;
    } else {
        newNode.next = frontPtr;
        frontPtr = newNode;
    }
    length++;
}
```

- Purpose: Inserts an element at the front of the queue.
- Parameters: value - Element to be inserted.

- Behavior:Creates a new node with the given value.Handles cases when the queue is empty or non-empty.

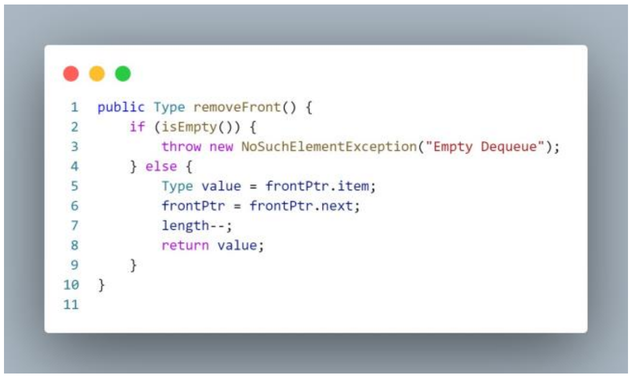
insertRear(Type value)



```
public void insertRear(Type value) {
    Node newNode = new Node(value);
    if (isEmpty()) {
        frontPtr = newNode;
        rearPtr = newNode;
    } else {
        rearPtr.next = newNode;
        rearPtr = newNode;
    }
    length++;
}
```

- Purpose: Inserts an element at the rear of the queue.
- Parameters: value - Element to be inserted.
- Behavior:Creates a new node with the given value.Handles cases when the queue is empty or non-empty.

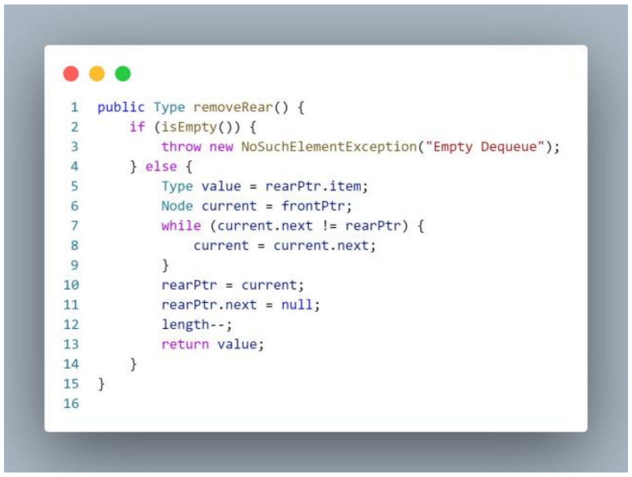
removeFront()



```
public Type removeFront() {
    if (isEmpty()) {
        throw new NoSuchElementException("Empty Dequeue");
    } else {
        Type value = frontPtr.item;
        frontPtr = frontPtr.next;
        length--;
        return value;
    }
}
```

- Purpose: Removes and returns the element from the front of the queue.
- Returns: The element at the front.
- Behavior:Handles cases when the queue is empty or non-empty.

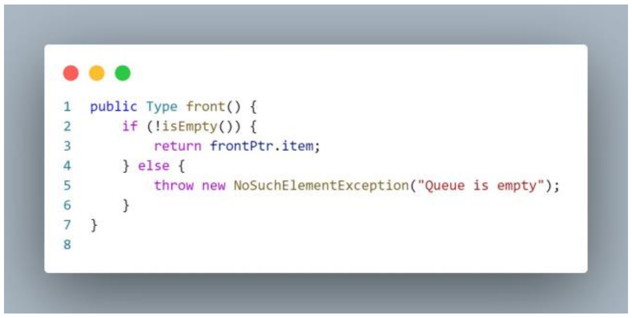
removeRear()



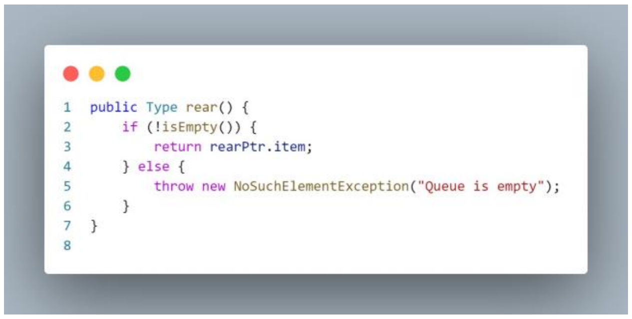
```java
public Type removeRear() {
    if (isEmpty()) {
        throw new NoSuchElementException("Empty Dequeue");
    } else {
        Type value = rearPtr.item;
        Node current = frontPtr;
        while (current.next != rearPtr) {
            current = current.next;
        }
        rearPtr = current;
        rearPtr.next = null;
        length--;
        return value;
    }
}
```

- Purpose: Removes and returns the element from the rear of the queue.
- Returns: The element at the rear.
- Behavior:Handles cases when the queue is empty or non-empty.

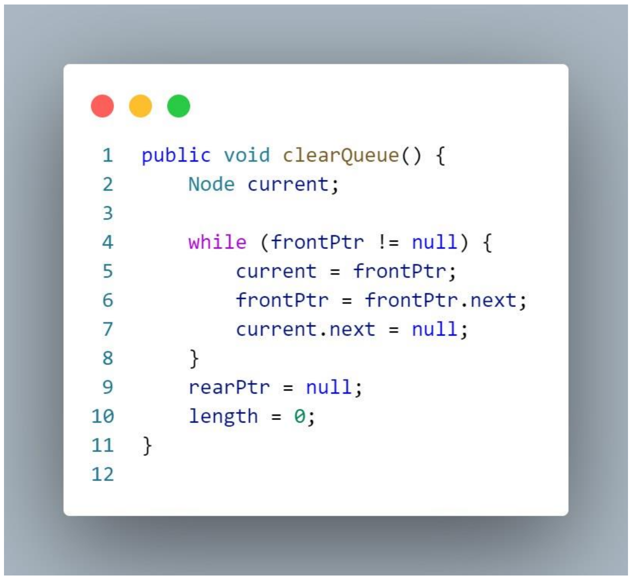
Other Methods



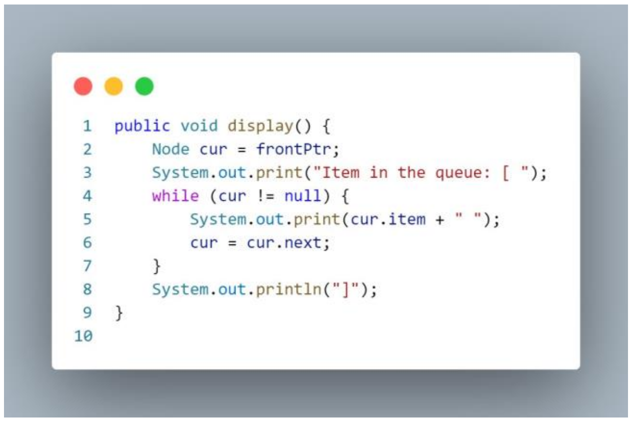
```java
public Type front() {
    if (!isEmpty()) {
        return frontPtr.item;
    } else {
        throw new NoSuchElementException("Queue is empty");
    }
}
```



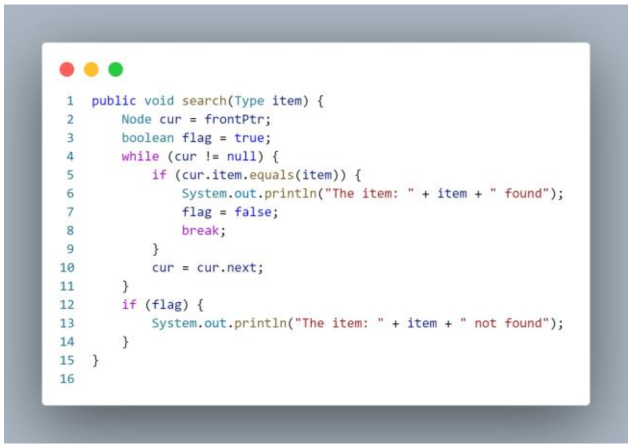
```java
public Type rear() {
    if (!isEmpty()) {
        return rearPtr.item;
    } else {
        throw new NoSuchElementException("Queue is empty");
    }
}
```



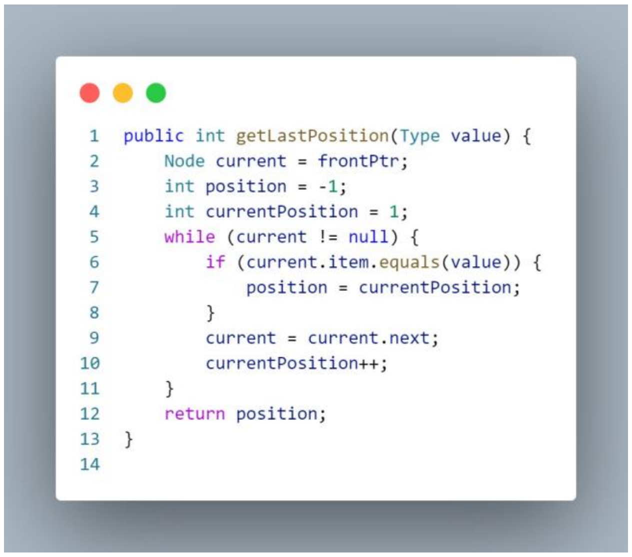
```java
public void clearQueue() {
    Node current;

    while (frontPtr != null) {
        current = frontPtr;
        frontPtr = frontPtr.next;
        current.next = null;
    }
    rearPtr = null;
    length = 0;
}
```



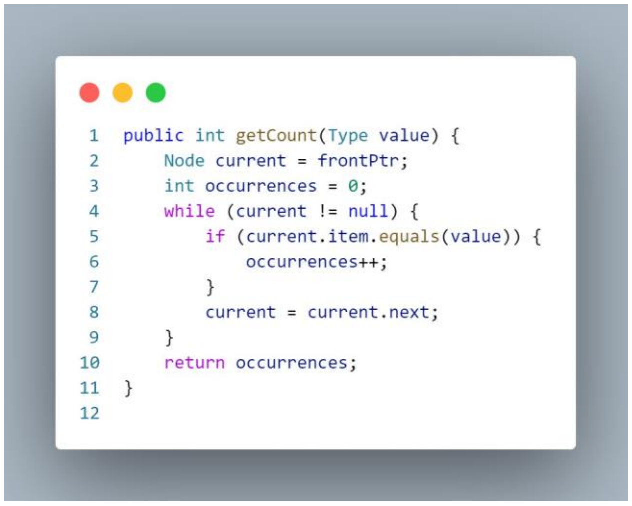
```java
public void display() {
    Node cur = frontPtr;
    System.out.print("Item in the queue: [ ");
    while (cur != null) {
        System.out.print(cur.item + " ");
        cur = cur.next;
    }
    System.out.println("]");
}
```



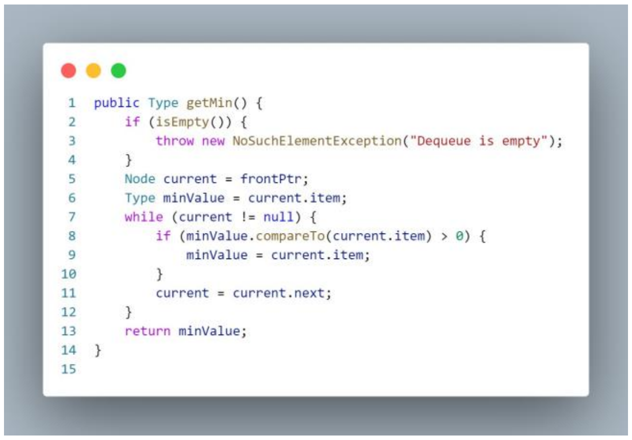
```java
public void search(Type item) {
    Node cur = frontPtr;
    boolean flag = true;
    while (cur != null) {
        if (cur.item.equals(item)) {
            System.out.println("The item: " + item + " found");
            flag = false;
            break;
        }
        cur = cur.next;
    }
    if (flag) {
        System.out.println("The item: " + item + " not found");
    }
}
```



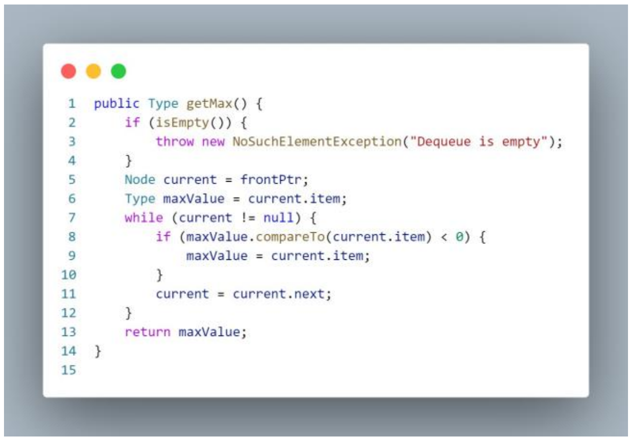
```java
public int getLastPosition(Type value) {
    Node current = frontPtr;
    int position = -1;
    int currentPosition = 1;
    while (current != null) {
        if (current.item.equals(value)) {
            position = currentPosition;
        }
        current = current.next;
        currentPosition++;
    }
    return position;
}
```



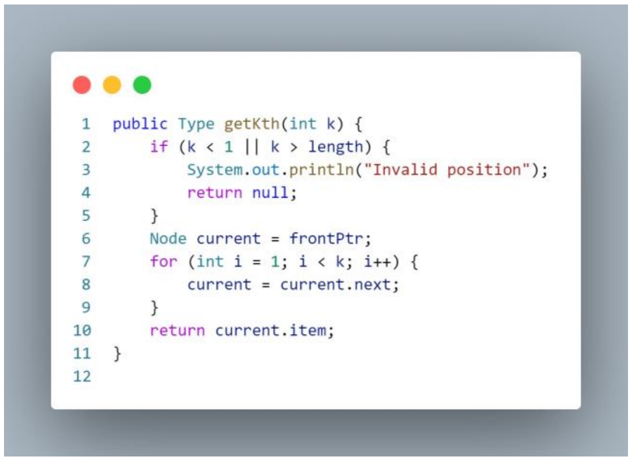
```java
public int getCount(Type value) {
    Node current = frontPtr;
    int occurrences = 0;
    while (current != null) {
        if (current.item.equals(value)) {
            occurrences++;
        }
        current = current.next;
    }
    return occurrences;
}
```



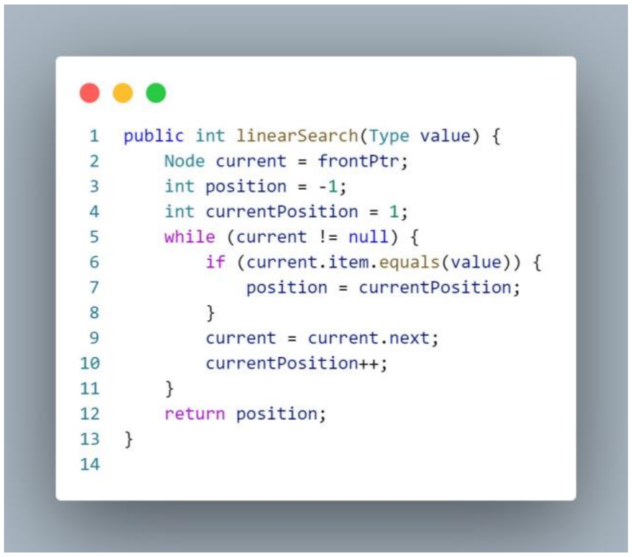
```java
public Type getMin() {
    if (isEmpty()) {
        throw new NoSuchElementException("Dequeue is empty");
    }
    Node current = frontPtr;
    Type minValue = current.item;
    while (current != null) {
        if (minValue.compareTo(current.item) > 0) {
            minValue = current.item;
        }
        current = current.next;
    }
    return minValue;
}
```



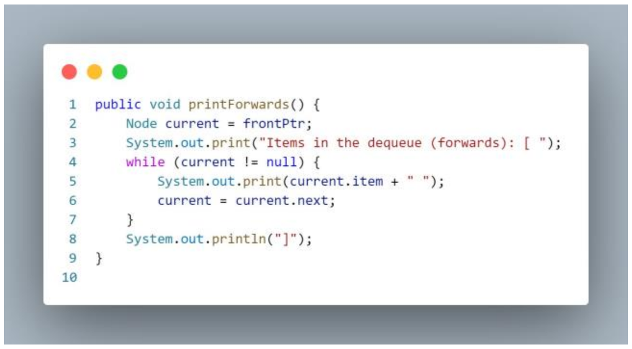
```java
public Type getMax() {
    if (isEmpty()) {
        throw new NoSuchElementException("Dequeue is empty");
    }
    Node current = frontPtr;
    Type maxValue = current.item;
    while (current != null) {
        if (maxValue.compareTo(current.item) < 0) {
            maxValue = current.item;
        }
        current = current.next;
    }
    return maxValue;
}
```



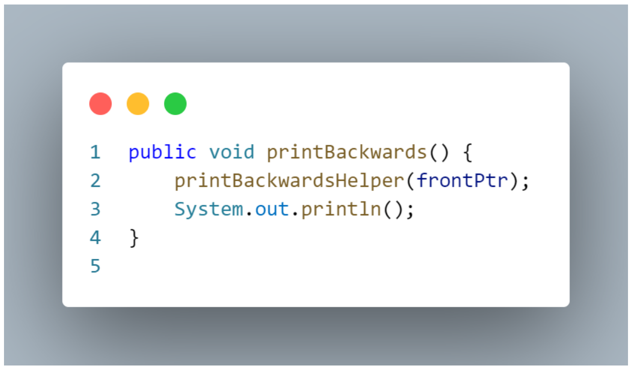
```java
public Type getKth(int k) {
    if (k < 1 || k > length) {
        System.out.println("Invalid position");
        return null;
    }
    Node current = frontPtr;
    for (int i = 1; i < k; i++) {
        current = current.next;
    }
    return current.item;
}
```



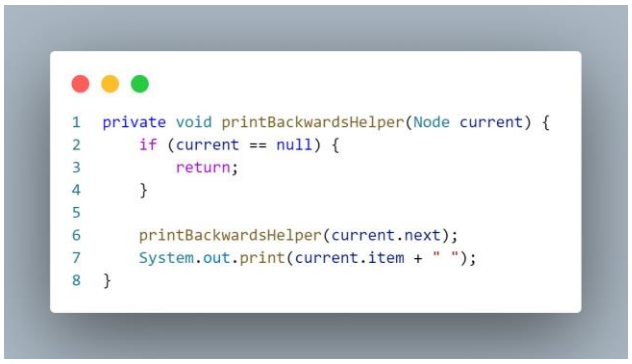
```java
public int linearSearch(Type value) {
    Node current = frontPtr;
    int position = -1;
    int currentPosition = 1;
    while (current != null) {
        if (current.item.equals(value)) {
            position = currentPosition;
        }
        current = current.next;
        currentPosition++;
    }
    return position;
}
```



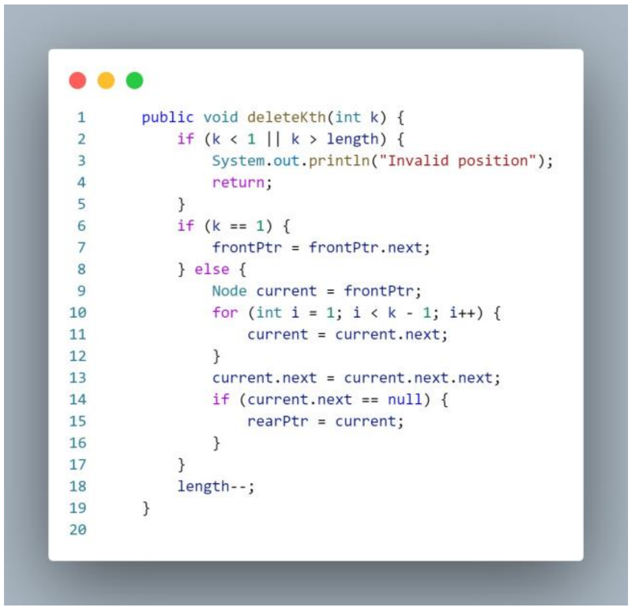
```java
public void printForwards() {
    Node current = frontPtr;
    System.out.print("Items in the dequeue (forwards): [ ");
    while (current != null) {
        System.out.print(current.item + " ");
        current = current.next;
    }
    System.out.println("]");
}
```



```java
public void printBackwards() {
    printBackwardsHelper(frontPtr);
    System.out.println();
}
```



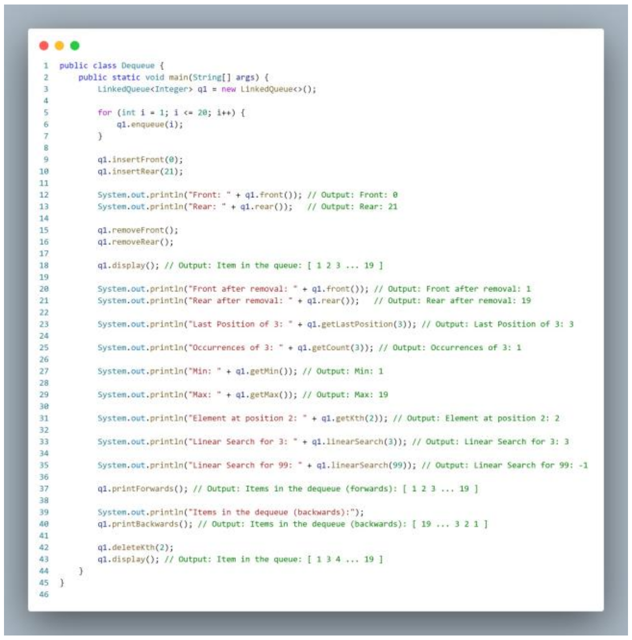
```java
private void printBackwardsHelper(Node current) {
    if (current == null) {
        return;
    }

    printBackwardsHelper(current.next);
    System.out.print(current.item + " ");
}
```



```java
    public void deleteKth(int k) {
        if (k < 1 || k > length) {
            System.out.println("Invalid position");
            return;
        }
        if (k == 1) {
            frontPtr = frontPtr.next;
        } else {
            Node current = frontPtr;
            for (int i = 1; i < k - 1; i++) {
                current = current.next;
            }
            current.next = current.next.next;
            if (current.next == null) {
                rearPtr = current;
            }
        }
        length--;
    }
```

- front(), rear(): Retrieve the front and rear elements of the queue.
- clearQueue(): Clears the entire queue.
- display(): Display all elements in the queue.
- search(Type item): Search for a specific item in the queue.
- getLastPosition(Type value): Find the last position of a given value.
- getCount(Type value): Count occurrences of a given value.
- getMin(), getMax(): Find the minimum and maximum values in the queue.
- getKth(int k): Retrieve the element at the k-th position in the queue.
- linearSearch(Type value): Perform a linear search for a value in the queue.
- printForwards(), printBackwards(): Print elements forwards and backwards in the queue.
- deleteKth(int k): Delete the element at the k-th position in the queue.

Dequeue Class (Main)



```java
public class Dequeue {
    public static void main(String[] args) {
        LinkedQueue<Integer> q1 = new LinkedQueue<>();

        for (int i = 1; i <= 20; i++) {
            q1.enqueue(i);
        }

        q1.insertFront(0);
        q1.insertRear(21);

        System.out.println("Front: " + q1.front()); // Output: Front: 0
        System.out.println("Rear: " + q1.rear());   // Output: Rear: 21

        q1.removeFront();
        q1.removeRear();

        q1.display(); // Output: Item in the queue: [ 1 2 3 ... 19 ]

        System.out.println("Front after removal: " + q1.front()); // Output: Front after removal: 1
        System.out.println("Rear after removal: " + q1.rear());   // Output: Rear after removal: 19

        System.out.println("Last Position of 3: " + q1.getLastPosition(3)); // Output: Last Position of 3: 3

        System.out.println("Occurrences of 3: " + q1.getCount(3)); // Output: Occurrences of 3: 1

        System.out.println("Min: " + q1.getMin()); // Output: Min: 1

        System.out.println("Max: " + q1.getMax()); // Output: Max: 19

        System.out.println("Element at position 2: " + q1.getKth(2)); // Output: Element at position 2: 2

        System.out.println("Linear Search for 3: " + q1.linearSearch(3)); // Output: Linear Search for 3: 3

        System.out.println("Linear Search for 99: " + q1.linearSearch(99)); // Output: Linear Search for 99: -1

        q1.printForwards(); // Output: Items in the dequeue (forwards): [ 1 2 3 ... 19 ]

        System.out.println("Items in the dequeue (backwards):");
        q1.printBackwards(); // Output: Items in the dequeue (backwards): [ 19 ... 3 2 1 ]

        q1.deleteKth(2);
        q1.display(); // Output: Item in the queue: [ 1 3 4 ... 19 ]
    }
}
```

- Purpose: Demonstrates the functionality of the LinkedQueue class by performing various operations on a queue of integers.

The code initializes a queue, performs enqueue and insertion operations, removes elements, searches for values, finds minimum and maximum values, retrieves elements by position, prints the queue in forwards and backwards, and deletes an element at a specified position.

This implementation covers various functionalities of a double-ended queue using a linked list.