Lernzettel

Queues und Deques: Prinzipien und Implementierungen

Universität: Technische Universität Berlin
Kurs/Modul: Einführung in die Informatik - Vertiefung
Erstellungsdatum: September 20, 2025

Lernzettel: Queues und Deques: Prinzipien und Implementierungen

(1) Definition. Eine Queue (Warteschlange) ist ein Abstract Data Type (ADT), bei dem Elemente in der Reihenfolge ihres Einfügens wieder entnommen werden: First-In, First-Out (FIFO). Typische Operationen sind Enqueue (Hinzufügen am Ende) und Dequeue (Entnehmen vom Anfang). Zusätzlich sinnvoll: Peek/Front (aktuelles vorderes Element), isEmpty, size.

Eine Deque (Double-Ended Queue) erweitert dieses Konzept: Elemente können sowohl am Anfang als auch am Ende eingefügt und entfernt werden. Typische Operationen sind addFirst, addLast, removeFirst, removeLast, getFirst, getLast, isEmpty, size.

(2) Grundoperationen. Queue

- Enqueue x: Element x wird ans Ende der Warteschlange angehängt.
- Dequeue: Element am Anfang wird entfernt und zurückgegeben.
- Peek/Front: Das vorderste Element wird ohne Entfernen zurückgegeben.
- isEmpty: Gibt an, ob die Struktur leer ist.
- size: Anzahl der Elemente.

Deque

- addFirst x / addLast x: Element x wird am vorderen bzw. hinteren Ende eingefügt.
- removeFirst / removeLast: Element wird vom vorderen bzw. hinteren Ende entfernt.
- getFirst / getLast: Vorderes bzw. hinteres Element ohne Entfernen.
- isEmpty, size: wie oben.

(3) Abstrakte Typen in Java (Schnittstellen).

```
public interface Queue<T> {
  void enqueue(T item);
  T dequeue();
  T peek();
  boolean isEmpty();
  int size();
}

public interface Deque<T> {
  void addFirst(T item);
  void addLast(T item);
  T removeFirst();
  T removeLast();
  T getFirst();
  T getLast();
  boolean isEmpty();
  int size();
}
```

(4) Implementierungen – Queue.
(4a) Array-basierte Queue (ring buffer). Kernidee: Ein ringförmiges Array mit Kopfzeiger (head) und Schl coming tail, der nächste freie Index ist tail. Der Speicher nutzt ein zusätzliches freies Slot-Konzept, um Leer- von Voll-Zuständen zu unterscheiden.

```
public class ArrayQueue<T> implements Queue<T> {
  private T[] data;
  private int head, tail; // head: erstes Element, tail: nächster freier Slot
  public ArrayQueue(int capacity) {
    data = (T[]) new Object[capacity + 1];
    head = tail = 0;
  }
  public void enqueue(T x) {
    if ((tail + 1) % data.length == head) resize();
    data[tail] = x;
    tail = (tail + 1) % data.length;
  }
  public T dequeue() {
    if (isEmpty()) throw new java.util.NoSuchElementException();
    T v = data[head];
    data[head] = null;
    head = (head + 1) % data.length;
    return v;
  }
  public T peek() {
    if (isEmpty()) throw new java.util.NoSuchElementException();
    return data[head];
  }
  public boolean isEmpty() { return head == tail; }
  public int size() { return (tail - head + data.length) % data.length; }

  private void resize() {
    int n = data.length - 1;
    T[] nd = (T[]) new Object[2 * n + 1];
    for (int i = 0; i < size(); i++) {
      nd[i] = data[(head + i) % data.length];
    }
    head = 0;
    tail = size();
    data = nd;
  }
}
```

(4b) Linked-Queue (verkettete Liste).

```
public class LinkedQueue<T> implements Queue<T> {
  private static class Node<T> { T value; Node<T> next; }
  private Node<T> head, tail;
  private int n = 0;

  public void enqueue(T x) {
```

```
    Node<T> node = new Node<>();
    node.value = x;
    node.next = null;
    if (tail != null) tail.next = node;
    tail = node;
    if (head == null) head = node;
    n++;
  }

  public T dequeue() {
    if (isEmpty()) throw new java.util.NoSuchElementException();
    T v = head.value;
    head = head.next;
    if (head == null) tail = null;
    n--;
    return v;
  }

  public T peek() {
    if (isEmpty()) throw new java.util.NoSuchElementException();
    return head.value;
  }

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

(5) Implementierungen – Deque.
(5a) ArrayDeque (ring buffer).

```
public class ArrayDeque<T> implements Deque<T> {
  private T[] data;
  private int head, tail; // leer: head == tail
  public ArrayDeque(int capacity) {
    data = (T[]) new Object[capacity + 1];
    head = tail = 0;
  }
  public void addFirst(T x) {
    head = (head - 1 + data.length) % data.length;
    if (head == tail) resize();
    data[head] = x;
  }
  public void addLast(T x) {
    data[tail] = x;
    tail = (tail + 1) % data.length;
    if (head == tail) resize();
  }
  public T removeFirst() {
    if (isEmpty()) throw new java.util.NoSuchElementException();
    T v = data[head];
```

```
    data[head] = null;
    head = (head + 1) % data.length;
    return v;
  }
  public T removeLast() {
    tail = (tail - 1 + data.length) % data.length;
    if (isEmpty()) throw new java.util.NoSuchElementException();
    T v = data[tail];
    data[tail] = null;
    return v;
  }
  public T getFirst() {
    if (isEmpty()) throw new java.util.NoSuchElementException();
    return data[head];
  }
  public T getLast() {
    if (isEmpty()) throw new java.util.NoSuchElementException();
    int idx = (tail - 1 + data.length) % data.length;
    return data[idx];
  }
  public boolean isEmpty() { return head == tail; }
  public int size() { return (tail - head + data.length) % data.length; }

  private void resize() {
    int n = data.length - 1;
    T[] nd = (T[]) new Object[2 * n + 1];
    for (int i = 0; i < size(); i++) {
      nd[i] = data[(head + i) % data.length];
    }
    head = 0;
    tail = size();
    data = nd;
  }
}
```

(5b) LinkedDeque (mit Doppelt-Kette und Sentinel).

```
public class LinkedDeque<T> implements Deque<T> {
  private static class Node<T> { T value; Node<T> prev, next; }
  private final Node<T> sentinel = new Node<>();
  private int n = 0;

  public LinkedDeque() {
    sentinel.next = sentinel.prev = sentinel;
  }

  public void addFirst(T x) {
    Node<T> node = new Node<>();
    node.value = x;
    node.next = sentinel.next;
    node.prev = sentinel;
```

```
    sentinel.next.prev = node;
    sentinel.next = node;
    n++;
  }

  public void addLast(T x) {
    Node<T> node = new Node<>();
    node.value = x;
    node.prev = sentinel.prev;
    node.next = sentinel;
    sentinel.prev.next = node;
    sentinel.prev = node;
    n++;
  }

  public T removeFirst() {
    if (isEmpty()) throw new java.util.NoSuchElementException();
    Node<T> node = sentinel.next;
    sentinel.next = node.next;
    node.next.prev = sentinel;
    n--;
    return node.value;
  }

  public T removeLast() {
    if (isEmpty()) throw new java.util.NoSuchElementException();
    Node<T> node = sentinel.prev;
    sentinel.prev = node.prev;
    node.prev.next = sentinel;
    n--;
    return node.value;
  }

  public T getFirst() {
    if (isEmpty()) throw new java.util.NoSuchElementException();
    return sentinel.next.value;
  }

  public T getLast() {
    if (isEmpty()) throw new java.util.NoSuchElementException();
    return sentinel.prev.value;
  }

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

(6) Komplexität und Eigenschaften. - Speicherbedarf: Bezeichnet durch die Anzahl der Elemente n, O(n) worst-case bei dynamischen Strukturen; In festen Arrays O(n) für Reallocation, ansonsten O(1) pro Element. - Zeitkomplexität (typisch): - Enqueue / Dequeue / addFirst /

addLast / removeFirst / removeLast: in der Regel O(1) amortisiert; Resize eines Arrays kostet O(n) und tritt selten auf. - Peek / getFirst / getLast: O(1). - size / isEmpty: O(1).

(7) Übungen (Beispiele). - Implementiere eine generische Klasse ArrayQueue<T> und teste mit Integer- und String-Werten. Prüfe en passant die Resize-Logik. - Implementiere eine einfache LinkedQueue<T> und vergleiche die Speicherbelegung mit der ArrayQueue für ähnliche Datenmengen. - Implementiere ein Deque-Interface mit einer ArrayDeque-Implementierung und teste alle sechs Operationen (addFirst, addLast, removeFirst, removeLast, getFirst, getLast). - Diskutiere Vor- und Nachteile von Array-basierten gegenüber Linked-Listen-Implementierungen für Queues und Deques.