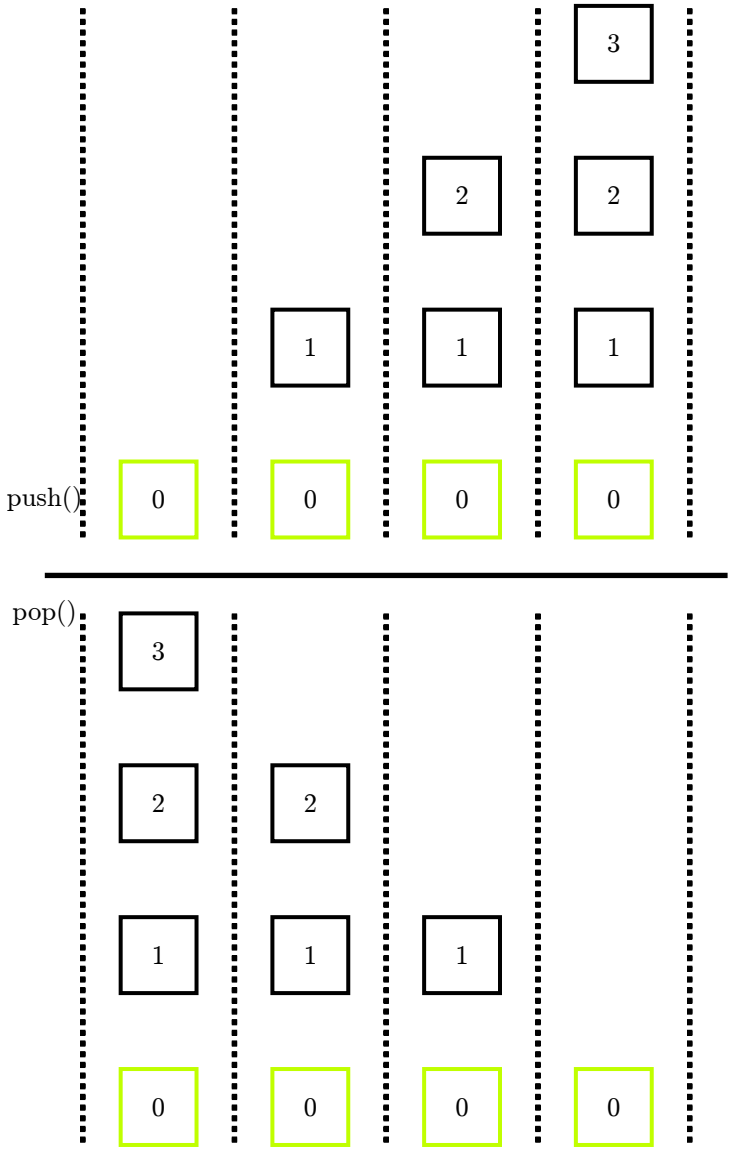
Dynamische Datenstrukturen

Marvin Baeumer **2023-12-06 10:20**

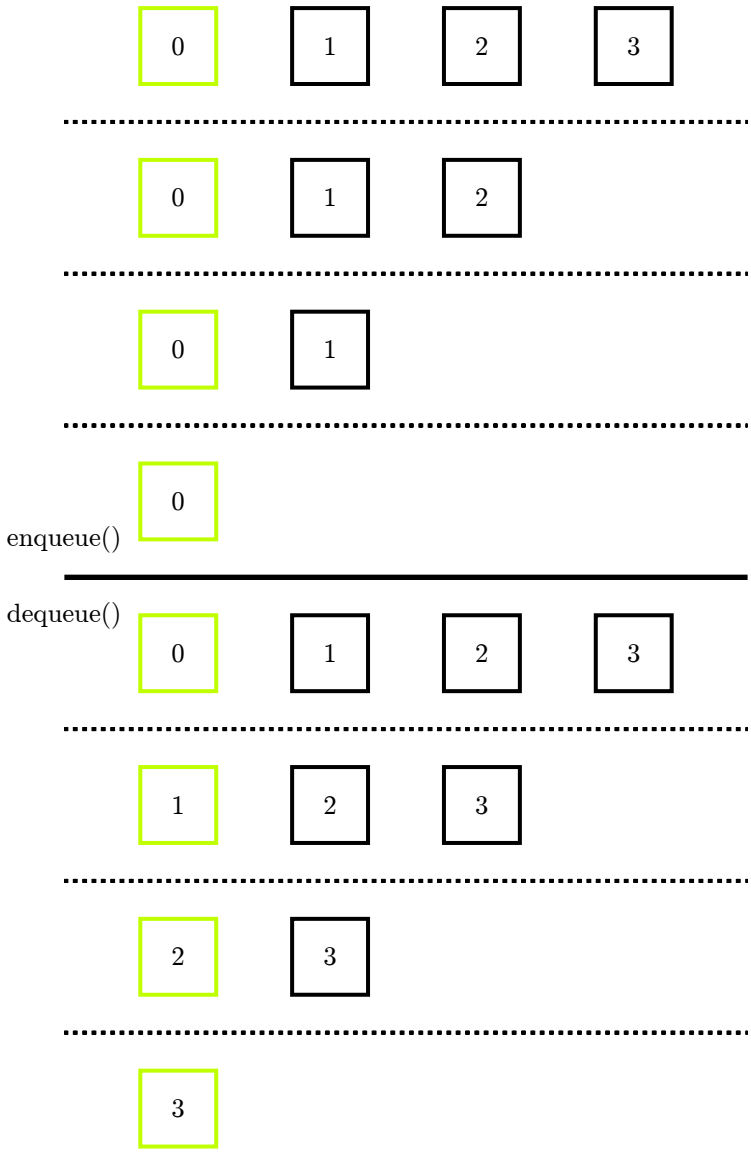
Stack

> Ein Stack ist eine Datenstruktur, die nach dem Prinzip "Last-In-First-Out" (LIFO) funktioniert. Das zuletzt hinzugefügte Element wird als erstes entfernt.



Queue

> Eine Queue ist eine Datenstruktur, die Elemente nach dem Prinzip "First-In-First-Out" (FIFO) verwaltet. Das bedeutet, dass das zuerst eingefügte Element auch als erstes wieder entfernt wird.



List

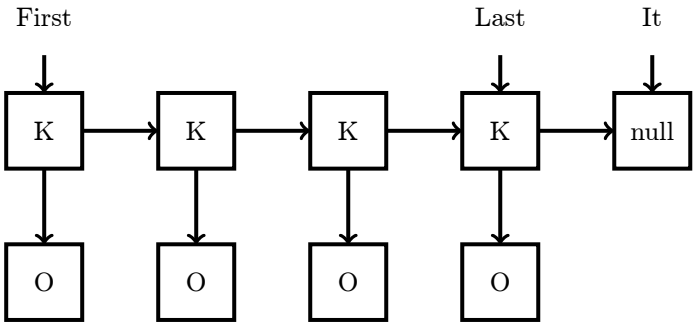
Beschreibung einer List

> Eine Liste ermöglicht den direkten Zugriff auf jedes Element und unterscheidet sich daher von einer Queue, bei der der Zugriff in der Regel auf das vorderste Element beschränkt ist. In einer Liste können Elemente an beliebigen Positionen eingefügt oder entfernt werden, was durch die Verwendung von Pointern wie "First", "Last" und "It" ermöglicht wird. Eine Liste kann entweder einfach verkettet sein, wobei jeder Knoten den nächsten Knoten speichert, oder doppelt verkettet, wobei jeder Knoten sowohl den nächsten als auch den vorherigen Knoten speichert.

Knoten Klasse

> Die Knotenklasse repräsentiert einen einzelnen Knoten in einer verketteten Datenstruktur. Jeder Knoten enthält eine Datenkomponente und einen Verweis auf den nächsten Knoten in der Kette. Diese Klasse wird häufig verwendet, um verkettete Listen zu implementieren.

Visualisierung von Knoten als List



einfach verkettet

> In einer Liste zeigt der Zeiger "First" immer auf das erste Element, während der Zeiger "Last" auf das letzte Element verweist. Der Zeiger "It" kann den Wert "null" haben und muss manuell einem Element zugewiesen werden. Mit Hilfe dieses Zeigers greift man auf einzelne Knoten und ihre Objekte zu, was Operationen wie das Einfügen, Löschen usw. ermöglicht.

Implementierung

```
public class Container {
        //Attribute
        Object item;
        Container next;
        // Konstuktor
        public Container(Object object) {
                this.object = object;
                this.next = null;
        }
        // get- und set Methoden
        public Object getObject() {
                return this.object;
        }
        public Container getNext() {
                return this.next;
        }
        public void setObject(Object object) {
                this.object = object;
        }
        public void setNext(Container next) {
                this.next = next;
        }
}
```

Documentation Queue

Art	Beschreibung
Konstruktor	Queue() - eine leere Queue wird erzeugt
Anfrage	boolean isEmpty() - Die Anfrage liefert den Wert true, wenn die Schlange keine Objekte enthält, sonst liefert sie den Wert false.
Auftrag	void enqueue(Object item) - Das Objekt item wird an die Schlange angehängt. Falls item gleich null ist, bleibt die Schlange unverändert.
Auftrag	void dequeue() - Das erste Objekt wird aus der Schlange entfernt. Falls die Schlange leer ist, wird sie nicht verändert.
Anfrage	Object front() Die Anfrage liefert das erste Objekt der Schlange. Die Schlange bleibt unverändert. Falls die Schlange leer ist, wird null zurückgegeben.

Documentation List

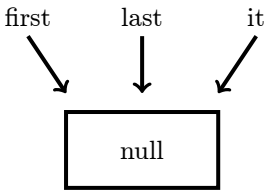
Art	Beschreibung
Konstruktor	List() Eine leere Liste wird erzeugt.
Anfrage	boolean isEmpty() Die Anfrage liefert den Wert true, wenn die Liste keine Objekte enthält, sonst liefert sie den Wert false.
Anfrage	boolean hasAccess() Die Anfrage liefert den Wert true, wenn es ein aktuelles Objekt gibt, sonst liefert sie den Wert false.
Auftrag	void next() Falls die Liste nicht leer ist, es ein aktuelles Objekt gibt und dieses nicht das letzte Objekt der Liste ist, wird das dem aktuellen Objekt in der Liste folgende Objekt zum aktuellen Objekt, andernfalls gibt es nach Ausführung des Auftrags kein aktuelles Objekt.
Auftrag	void toFirst() Falls die Liste nicht leer ist, wird das erste Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.
Auftrag	void toLast() Falls die Liste nicht leer ist, wird das letzte Objekt der Liste aktuelles Objekt. Ist die Liste leer, geschieht nichts.
Anfrage	Object getObject() Falls es ein aktuelles Objekt gibt, wird das aktuelle Objekt zurückgegeben, andernfalls gibt die Anfrage den Wert null zurück.
Auftrag	void setObject(Object pObject) Falls es ein aktuelles Objekt gibt und pObject ungleich null ist, wird das aktuelle Objekt durch pObject ersetzt.
Auftrag	void append(Object pObject) Ein neues Objekt pObject wird am Ende der Liste angefügt. Das aktuelle Objekt bleibt unverändert. Wenn die Liste leer ist, wird das Objekt pObject in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt. Falls pObject gleich null ist, bleibt die Liste unverändert.
Auftrag	void insert(Object pObject) Falls es ein aktuelles Objekt gibt, wird ein neues Objekt vor dem aktuellen Objekt in die Liste eingefügt. Das aktuelle Objekt bleibt unverändert. Falls die Liste leer ist und es somit kein aktuelles Objekt gibt, wird pObject in die Liste eingefügt und es gibt weiterhin kein aktuelles Objekt. Falls es kein aktuelles Objekt gibt und die Liste nicht leer ist oder pObject gleich null ist, bleibt die Liste unverändert.
Auftrag	void concat(List pList) Die Liste pList wird an die Liste angehängt. Das aktuelle Objekt bleibt unverändert. Falls pList null oder eine leere Liste ist, bleibt die Liste unverändert.
Auftrag	void remove() Falls es ein aktuelles Objekt gibt, wird das aktuelle Objekt gelöscht und das Objekt hinter dem gelöschten Objekt wird zum aktuellen Objekt. Wird das Objekt, das am Ende der Liste steht, gelöscht, gibt es kein aktuelles Objekt mehr. Wenn die Liste leer ist oder es kein aktuelles Objekt gibt, bleibt die Liste unverändert.

Relevante Faelle der Methode Insert - Graphische Darstellung

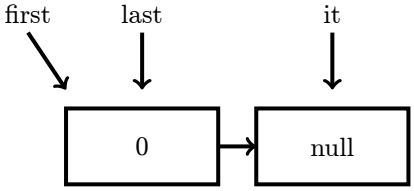
Fall 1 $\rightarrow$ leere Liste

> Bei einer leeren Liste zeigen die Zeiger `first`, `last` und `it` auf `null`. Beim Aufrufen der Methode `insert` wird nun ein Element eingefügt, wobei `it` weiterhin `null` bleibt und `first` und `last` auf das einzige Element (0) in der Liste zeigen.

Zustand vor dem Aufruf der Methode Insert



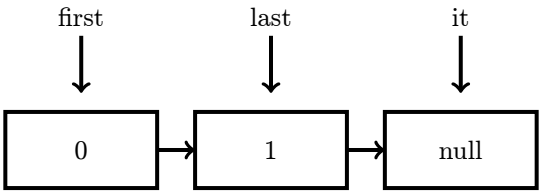
Zustand nach dem Aufruf der Methode Insert



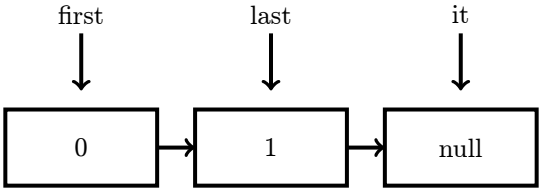
Fall 2 $\rightarrow$ nicht leere Liste ohne aktuelles Object

> Wenn die `Insert`-Methode aufgerufen wird und es kein aktuelles Element in der Liste gibt, die Liste nicht leer ist, bleibt die Liste unverändert.

Zustand vor dem Aufruf der Methode Insert



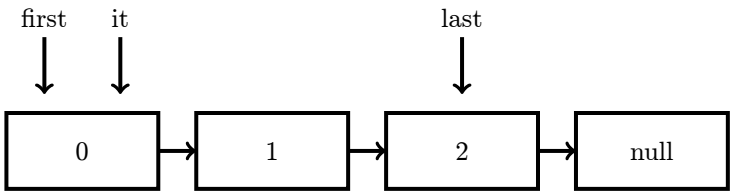
Zustand nach dem Aufruf der Methode Insert



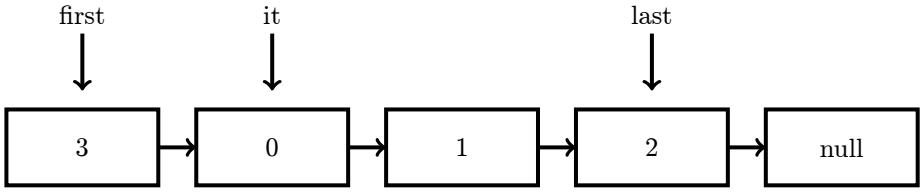
Fall 3 → Einfügen vor dem ersten Element

> Das aktuelle Element `it` muss dasselbe Element wie `first` sein. Danach wird die Methode `insert` aufgerufen. Das neue Element (3) wird dann eingefügt, wobei `first` auf das neue Element (3) zeigt und `it` auf dasselbe Element (0) zeigt, nur eine Position weiter hinten.

Zustand vor dem Aufruf der Methode Insert



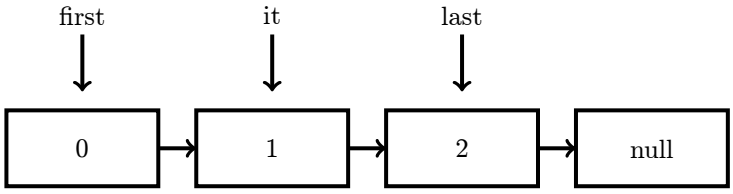
Zustand nach dem Aufruf der Methode Insert



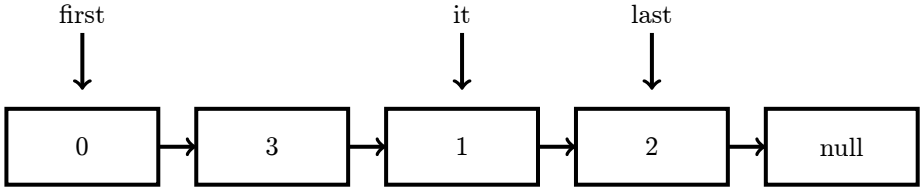
Fall 4 → Allgemeiner Fall – es gibt ein aktuelles Element, welches nicht das erste Element ist.

> Bei diesem Fall gibt es ein aktuelles Element, und die Liste ist nicht leer. Wenn nun die Methode `Insert` aufgerufen wird, wird das neue Element (3) vor dem aktuellen Element eingefügt. Das aktuelle Element bleibt dabei unverändert, also weiterhin die

Zustand vor dem Aufruf der Methode Insert



Zustand nach dem Aufruf der Methode Insert



Implementierung der Methode `remove` und `insert`

```
public void remove() {
        if(hasAccess()) {
//Schauen ob Elemente in der Liste sind
                if(first == last) {
//Schauen ob nur ein Element in der Liste ist
                        first = null;
                        last = null;
                        it = null;
//Wenn nur ein Element vorhanden dann wird alles null da wir es entfernen wollen
                } else {
                        if(first = it) { //Loeschen des ersten Elements
                                it = first.getNext();
                                first = it;
//Wenn nun das erste Element auch unser aktuelles Element ist so kann it das
nachfolgende Element sein und first ebenso da wir so kein Pointer mehr auf dem
vorherigen Element haben und es so geloscht wird
                        }
                        else {
                                Container current = first;
                                while(current != it) {
                                        current = current.getNext();
                                }
                                current.setNext(it.getNext)
                                it = current.getNext();
//Solnage durchgehen bis wir it erreichthaben
                                if(current.getNext() == null) {
                                        last = current;
                                }
                        }
                }
        }
}
```

void insert(Object item)

```
public void insert(Object item) {
        if(this.isEmpty()) {
            append(item);
        }
        else if(this.hasAccess() && item != null) {
            Knoten neu = new Knoten(item, aktuell);
            if(aktuell == kopf) {
                kopf = neu;
            }
            else {
                Knoten front = kopf;
                while(front.getNachfolger() != aktuell) {
                    front = front.getNachfolger();
                }
                front.setNachfolger(neu);
            }
        }
    }
```