Dynamische Datenstrukturen

von Christian Bartl

Inhaltsverzeichnis

1 Dynamische Datenstrukturen

Dynamische Datenstrukturen bezeichnen in der Informatik und Programmierung Datenstrukturen die eine flexible Menge an Arbeitsspeicher reservieren. Dynamische Datenstrukturen ändern ihre Struktur und den von ihnen belegten Speicherplatz während der Programmausführung und je nach Anzahl der hinzugefügten Elemente. Sie sind aus einzelnen Elementen, den einzelnen Knoten, aufgebaut, zwischen denen üblicherweise eine gewisse Nachbarschaftsbeziehung (Verweise) besteht. Die Dynamik liegt im Einfügen neuer Knoten, Entfernen vorhandener Knoten und in der Änderung der Nachbarschaftsbeziehungen (Verschieben eines Knoten). Die Beziehung zwischen den einzelnen Knoten wird über Zeiger hergestellt, die jeweils auf den Speicherort des logischen Nachbarn zeigen. Jeder Knoten enthält daher neben den zu speichernden Nutzdaten mindestens einen Zeiger auf die jeweiligen Nachbarknoten. Man nennt solche Strukturen daher auch verkettete oder rekursive Datenstrukturen.

Die wichtigsten dieser Datenstrukturen sind:

- **Listen**
  - Einfachverkettete Liste
  - Doppeltverkettete Liste
  - Queue (First in First out)
  - Stack (Last in First out)
- **Bäume**
  - Binärbaum (zwei Nachfolger)
  - Vielwegbaum (viele Nachfolger)

Die wichtigsten Operationen mit dynamischen Datenstrukturen sind:

- Erzeugen eines neuen Elementes
- Einfügen eines Elementes
- Entfernen eines Elementes

2 Die wichtigsten Datenstrukturen

2.1 Einfachverkettete Liste

Bei der dynamischen Datenstruktur einfachverkettete Liste enthält jedes Element einen Datensatz und einen Zeiger auf das nächste Element in der Liste. Der Zeiger des ersten Elementes ist der Root-Zeiger (Ausgangszeiger) von dem aus jede Operation in der Liste begonnen wird. Der Zeiger des letzten Elements zeigt immer auf NULL, so kann das Ende der Liste durch Abfrage auf NULL in einer Schleife gefunden werden. Es ist möglich ein neues Element an jeder beliebigen Stelle einzufügen.

2.1.1 Pseudocode für die Implementierung

2.1.1.1 Liste

```
object einfacheListe
{
	Daten;

	Zeiger auf nächstes Listenelement/bei letztem Element auf Null;
}
```

2.1.1.2 Einfügen

```
// Einfügen hinter dem Element Nummer index
```

```
void insert(object, index)
{
	einfacheListe neuesElement;

	einfacheListe.Daten=object;
	//Queue-Element mit Daten (object) befüllen;

	repeat until (Element mit Nummer index erreicht)
	{
		Element-Zeiger = Element-Zeiger (von Element index);

		// Zeiger des gesuchten Elementes ermitteln
	}

	neuesElement-Zeiger.„nächstes Element“-Zeiger = Element-Zeiger.“nächstes
	Element“-Zeiger;
	Element-Zeiger.„nächstes Element“-Zeiger = neuesElement-Zeiger;

	/* Bei erstem Element ist der Zeiger auf das vorherige Element NULL,
	beim letzten Element ist der Zeiger auf das nächste Element NULL */
}
```

2.1.1.3 Auslesen

```
object get(index)
{
	repeat until (Element mit Nummer index erreicht)
	{
		return object;
	}
}
```

2.1.1.4 Löschen

```
void delete(index)
{
	repeat until (Element mit Nummer index erreicht)
	{
		vorherigesElement-Zeiger = Element-Zeiger (von Element index-1);
		deleteElement-Zeiger = Element-Zeiger (von Element index);
		// Zeiger des vorherigen und des gesuchten Elementes ermitteln.
	}

	vorherigesElement.„nächstes Element“-Zeiger = deleteElement-
	Zeiger.“nächstes Element“-Zeiger;
	FreeMem(deleteElement-Zeiger);
	/* „nächstes Element“-Zeiger des vorherigen Elements auf das nächste
	Element legen. Speicher des zu löschenden Elementes freigeben. */
}
```

2.2 Doppeltverkettete Liste

Bei der dynamischen Datenstruktur der doppeltverketteten Liste enthält jedes Element einen Datensatz und einen Zeiger auf das nächste Element, sowie einen Zeiger auf das vorherige Element in der Liste. Der „nächstes Element“-Zeiger des ersten Elementes ist der Root-Zeiger (Ausgangszeiger), von dem aus jede Operation in der Liste begonnen wird. Der „vorheriges Element“-Zeiger des ersten Elements zeigt auf NULL; Der „nächstes Element“-Zeiger des letzten Elements zeigt ebenfalls auf NULL, so kann das Ende der Liste durch Abfrage auf NULL in einer Schleife gefunden werden. Es ist möglich ein neues Element an jeder beliebigen Stelle einzufügen. Der Vorteil der doppeltverketteten Liste liegt darin, dass man von jedem Element aus in jede Richtung gehen kann.

2.2.1 Pseudocode für die Implementierung

2.2.1.1 Liste

```
object doppelteListe
{
	Daten;

	Zeiger auf vorheriges Listenelement/bei ersten Element auf Null;
```

```
        Zeiger auf nächstes Listenelement/bei letzten Element auf Null;
}
```

2.2.1.2 Einfügen

```
// Einfügen hinter dem Element Nummer index
void insert(object, index)
{
        doppelteListe neuesElement;

        doppelteListe.Daten=object;
        //Queue-Element mit Daten (object) befüllen;

        repeat until (Element mit Nummer index erreicht)
        {
                Element-Zeige r= Element-Zeiger (von Element index);
        }
        neuesElement.“vorheriges Element“-Zeiger = Element-Zeiger;
        neuesElement.“nächstes Element“-Zeiger = Element-Zeiger.“nächstes
        Element“-Zeiger;

        /* „nächstes Element“-Zeiger des neuen Elements auf das nachfolgende
        Element legen. „vorheriges Element“-Zeiger des neuen Elements auf das
        vorherige Element legen. */

        Element-Zeiger.“nächstes Element“-Zeiger.“vorheriges Element“-Zeiger =
        neuesElement-Zeiger;
        Element-Zeiger.“nächstes Element“-Zeiger = neuesElement-Zeiger;

        /* „nächstes Element“-Zeiger des vorherigen Elements auf das neue
        Element legen. „vorheriges Element“-Zeiger des nächsten Elements auf das
        neue Element legen. */
}
```

2.2.1.3 Auslesen

```
object get(index)
{
        repeat until (Element mit Nummer index erreicht)
        {
                return object;
        }
}
```

2.2.1.4 Löschen

```
void delete(index)
{
        repeat until (Element mit Nummer index erreicht)
        {
                deleteElement-Zeiger = Zeiger (von Element index);
        }

        deleteElement.„nächstes Element“-Zeiger.„vorheriges Element“-Zeiger =
        deleteElement.„vorheriges Element“-Zeiger;
        deleteElement.„vorheriges Element“-Zeiger.„nächstes Element“-Zeiger =
        deleteElement.„nächstes Element“-Zeiger;
        FreeMem(deleteElement-Zeiger);

        /* „nächstes Element“-Zeiger des vorherigen Elements auf das nächste
        Element legen. „vorheriges Element“-Zeiger des nächsten Elements auf das
        vorherige Element legen. Speicher des zu löschenden Elementes freigeben.
        */
}
```

2.3 Binärer Baum

Jedes Element eines binären Baumes enthält einen Datensatz sowie einen Zeiger auf das links darunterliegende Element und einen auf das rechts darunterliegende Element. Der Root-Zeiger zeigt auf das oberste Element, von diesem ausgehend werden alle Operationen innerhalb des Baumes gestartet. Jeder Zeiger, der auf kein Element zeigt, wird auf NULL gesetzt.

2.3.1 Pseudocode für die Implementierung

```
Object Baum
{
	Daten des Blattes;

	Zeiger auf linkes Blatt;
	Zeiger auf rechtes Blatt;
}
```

2.4 Queue

Die Queue (deut. Warteschlange) ist eine lineare dynamische Datenmenge. Elemente werden am Ende hinzugefügt und am Anfang entfernt. Die Queue implementiert die so genannte **FIFO**-Strategie (**F**irst **I**n, **F**irst **O**ut). Das heißt, dass das erste Element, das eingefügt wurde, auch das erste ist, welches wieder entfernt wird.

Für die Einfüge- und Löschoperation einer Queue benutzt man die Bezeichnungen enqueue (einfügen) und dequeue (löschen).

2.4.1 Praxisbeispiel

Die FIFO-Strategie ist vergleichbar mit dem Ablauf an einer roten Ampel:

Das erste Auto bleibt ganz vorne stehen. Alle nachfolgenden bleiben hinter dem ersten, dem zweiten, usw. stehen. Schaltet die Ampel auf grün, fährt zuerst das vorderste, also jenes Auto das zuerst an der Ampel war, und dann nach einander alle anderen weiter.

2.4.2 Pseudocode für die Implementierung

2.4.2.1 Queue

```
Object Queue
{
	Daten;

	Zeiger auf nächste Element/bei letztem Element auf NULL;
}
```

2.4.2.2 Enqueue

```
void enqueue(object)
{
	Queue neuesElement;

	neuesElement.Daten = object;
	//Queue-Element mit Daten (object) befüllen

	repeat until (letztes Element erreicht)
	{
		lastElement-Zeiger = Element-Zeiger;
	}

	lastElement-Zeiger.„nächstes Element“-Zeiger = neuesElement-Zeiger;
	neuesElement-Zeiger.„nächstes Element“-Zeiger = NULL;

	/* Letztes Element der Queue ermitteln. „nächstes-Element“-zeiger auf
	das neue Element legen. Zeiger des neuen Elements auf NULL setzen. */
}
```

2.4.2.3 Dequeue

```
object enqueue()
{
	repeat until (letztes Element erreicht)
	{
		vorletztesElement-Zeiger = Element-Zeiger (von Element index-1);
```

```
                lastElement-Zeiger = Element-Zeiger (von Element index);
                // Letztes und vorletztes Element der Queue ermitteln
        }

        object temp=lastElement;

        vorletztesElement-Zeiger.„nächstes Element“-Zeiger=NULL;
        FreeMem(lastElement-Zeiger);

        /* Letztes Element auslesen und Speicher freigeben. Zeiger des
        vorletzten Elements auf NULL setzen. */

        return temp;
}
```

2.5 Stack

Ein Stack (deut. Stapel) ist eine dynamische Datenmenge, in der Daten linear „übereinander“ gespeichert werden. Es kann nur immer das oberste (erste) Element des Stacks entfernt werden und genauso können neue Elemente nur an oberster Stelle des Stacks eingefügt werden. Der Stack realisiert die so genannte **LIFO**-Strategie (**L**ast **I**n, **F**irst **O**ut). Das bedeutet, dass jenes Element, welches zuletzt eingefügt wurde auch wieder zuerst entfernt wird.

Für die Operationen an einem Stack gibt es 2 spezielle Begriffe. Push bezeichnet den Vorgang des Einfügens eines Elementes an der Spitze des Stacks. Pop ist die Bezeichnung für das Entfernen des obersten Elementes vom Stack.

2.5.1 Praxisbeispiel

Die LIFO-Strategie ist vergleichbar mit einem Stapel Holzbretter:

Beim übereinanderschlichten von Holzbrettern wird das nächste immer ganz oben aufgelegt. Wenn ich jetzt ein Holzbrett vom Stapel nehme, nehme ich das oberste. Das heißt das Brett das ich zuletzt auf den Stapel gelegt habe ist auch das erste das ich wieder herunternehme.

2.5.2 Pseudocode für die Implementierung

2.5.2.1 Stack

```
Object Stack
{
        Daten;

        Zeiger auf unteres Element/ bei letztem Element auf NULL;
}
```

2.5.2.2 Push

```
void push(object)
{
        Stack neuesElement;

        neuesElement.Daten = object;
        //Stack-Element mit Daten (object) befüllen

        neuesElement-Zeiger.„nächstes Element“-Zeiger = StackRoot-Zeiger;
        StackRoot-Zeiger = neuesElement-Zeiger;

        /* Oberstes Element des Stacks ermitteln. Zeiger des neuen Stacks auf
        das oberste Element setzten.*/
}
```

2.5.2.3 Pop

```
object pop()
{
```

```
            object temp = StackRoot.object;

            temp-Zeiger = StackRoot-Zeiger;
            StackRoot-Zeiger=StackRoot-Zeiger.„nächstesElement“-Zeiger;
            FreeMemory(temp-Zeiger);

            return object;
    }
```

3 Zusammenhang - dynamischen Datenstrukturen und Collections

Dynamische Datenstrukturen, wie wir sie oben kennen gelernt haben, sind in Java durch das Collections-Framework realisiert. Dies sind eigene Klassen die es erlauben Sammlungen von Objekten zu erstellen und diese nennt man eben Collections. Eine Collection verfügt über die passenden Methoden um Objekte einzufügen und wieder zu entfernen und über Methoden die Abfragen zulassen. Das Collections-Framework befindet sich in dem Paket „java.util“. Es besteht aus einer Reihe von Interfaces und Klassen die diese Interfaces implementieren.