Suchen von Daten

Eine sicherlich grundlegende Aufgabe in der Informatik ist das Suchen von Daten oder Objekten in Listen[1].

Die in diesem Zusammenhang wahrscheinlich sehr häufig anzutreffenden Verfahren sind die:

- Lineare Suche; hierbei muss die zu durchsuchende Menge nicht geordnet sein.
- Binäre Suche; in diesem Fall müssen die Elemente in Bezug auf das Suchkriterium bereits sortiert sein.

Lineare Suche

Bei dem Algorithmus „Lineare Suche“ oder auch „Sequentielle Suche“, werden in der zu durchsuchenden Liste oder dem Array alle Elemente nacheinander überprüft. Wird das gesuchte Element gefunden, endet die Suche. Ist das Ende der Liste/des Arrays erreicht und das gesuchte Element wurde nicht gefunden, war die Suche erfolglos.

Dieser Algorithmus ist sicherlich der Einfachste, der vor allem einen Vorteil gegenüber der binären Suche hat: Die Reihenfolge der Elemente kann beliebig sein.

Allerdings ist dieser Algorithmus lediglich für kleine Datenmengen sinnvoll. Die Zeitkomplexität der linearen Suche ist im besten Fall *O(1)*, nämlich wenn das gesuchte Element das erste Element in der Liste ist. Im schlechtesten Fall ist die Zeitkomplexität *O(n)*, wenn das gesuchte Element das letzte Element bzw. nicht vorhanden ist. Im Mittel benötigt die Suche *(n+1) / 2* Vergleichsoperationen und gehört damit ebenfalls zur Menge *O(n)*.

Die lineare Suche kann zwar für einen speziellen Datentyp wie z.B. **`int`** implementiert werden. Allerdings lässt sich die Suche auch generisch mittels einer (statischen) Klasse implementieren. Eine solche ist im nachfolgenden Kasten dargestellt.

```
// Der Datentyp T muss das das Interface
// IComparable implementieren
public static class LinearSeek<T> where T:IComparable
{
    public static int FoundIndex { get; private set; }

    public static bool IsFound { get; private set; }

    public static bool Seek(T[] array, T seekelement)
    {
        // Guard Clauses
        if (array == null) throw new ArgumentNullException("array");
        if (seekelement == null) throw new ArgumentNullException("seekelement");

        FoundIndex = 0;
        IsFound = false;

        foreach (T item in array)
        {
            if (item.CompareTo(seekelement) == 0)
            {
                IsFound = true;
                break;
            }
            FoundIndex++;
        }
        return IsFound;
    }
}
```

1 Es gibt auch andere Datenstrukturen (z.B. Bäume), bei denen andere Suchverfahren angewendet werden können

Die Klasse **`LinearSeek`** ist als statische Klasse implementiert, so dass keine Objektinstanz erstellt werden muss bzw. kann.

Diese Klasse kann auf jede Art von Objekt-Array angewendet werden, sofern die Objekt-Klasse das Interface **`IComparable`** implementiert hat. Dieses Interface verpflichtet die betreffende Klasse zur Implementation einer Methode **`CompareTo(object o):int`**.

In dieser Methode muss dann für Objekte dieser Klasse definiert sein, wie der Vergleich auszuführen ist. Hierbei wird die Methode für ein bestimmtes Objekt aufgerufen und ein zweites Objekt für den Vergleich als Parameter übergeben.

Die Methode **`CompareTo()`** liefert dann einen **`int`**-Wert zurück und zwar:

- **`-1`**, wenn das Instanzobjekt in der Sortierreihenfolge vor dem Parameterobjekt liegt;
- **`0`**, wenn Instanzobjekt und Parameterobjekt an der gleichen Position liegen;
- **`1`**, wenn das Instanzobjekt in der Sortierreihenfolge nach dem Parameterobjekt liegt.

Der Vergleich kann dann in beliebiger Form anhand der Objekteigenschaften in **`CompareTo()`** implementiert werden. Bei einer Verwendung von Eigenschaften des Typs **`string`** kann die in **`string`** bereits enthaltene Methode **`CompareTo()`** verwendet werden.

Beispiele

```
public class Circle:IComparable
{
    public double Radius { get; private set; }
    public double Area { get; }

    public Circle(double radius)
    {
        Radius = radius;
        Area = radius * radius * Math.PI;
       }

    public int CompareTo(object obj)
    {
        Circle c = (Circle)obj;
        if (this.Radius < c.Radius)
            return -1;
        else if (this.Radius > c.Radius)
            return 1;
        else
            return 0;
    }
}
```

```
public class Person:IComparable
{
    public string Vorname { get; private set; }
    public string Nachname { get; private set; }

    public string VollerName
    {
        get { return Vorname + " " + Nachname; }
    }
    public Person(string vorname, string nachname)
    {
        Vorname = vorname;
        Nachname = nachname;
       }

    // Für IComparable muss die Methode
    // CompareTo() implementiert werden
    public int CompareTo(object obj)
    {
        Person p = (Person)obj;
        // Besonderheit beim Vergleich von Strings
        // Der Datentyp string hat bereits CompareTo()
        // eingebaut

        // Berücksichtigt Groß- und Kleinschreibung
        return this.VollerName.CompareTo(p.VollerName);
    }
}
```

Im Beispiel mit der Klasse **`Circle`** wird der Vergleich anhand eines numerischen Property, dem **`Radius`**, durchgeführt. Bei der Klasse **`Person`** hingegen wird der Vergleich anhand des Property **`VollerName`** durchgeführt. Da dieses Property vom Typ **`string`** ist, kann hier die in **`string`** bereits vorhandene Methode **`CompareTo()`** eingesetzt werden.

In allen Fällen muss jedoch ein cast für das Parameterobjekt durchgeführt werden, da die Schnittstelle von **`CompareTo()`** den Datentyp **`object`** als Parametertyp hat!

Aufgabe

Implementieren Sie die Klasse **`LinearSeek`** und ein Testprogramm für Suchvorgänge mit Wert- bzw. eigens erstellten Referenztypen.

Binäre Suche

Die binäre Suche ist ein Suchalgorithmus, der insbesondere bei größeren Datenmengen deutlich schneller läuft als die lineare Suche. Notwendige Vorbedingung ist jedoch, dass die Datenmenge bereits nach dem Suchschlüssel sortiert ist. Die binäre Suche verwendet das Grundprinzip „Teile und herrsche“ („Divide and Conquer“). Darunter versteht man, ein „schwer“ lösbares Problem solange in kleinere Teilprobleme zu zerlegen, bis diese dann „beherrschbar“ sind.

Bei der binären Suche wird zumeist in der Mitte der zu durchsuchenden Liste bzw. Array begonnen. Hierbei wird nun überprüft, ob dieses Element größer oder kleiner als das Gesuchte ist. Im günstigsten Fall ist es bereits das Richtige . Die Zeitkomplexität für den günstigsten Fall wäre dann *O(1)*. Jetzt wird in der entsprechenden Hälfte wiederum das jeweils mittige Element heraus genommen und überprüft. Dieser Vorgang wird fortgesetzt, bis entweder die gesamte Liste/Array ohne Erfolg durchsucht wurde oder bis das betreffende Element gefunden ist. Hierbei muss jedes Mal entweder die Ober- oder Untergrenze der verbleibenden Restmenge neu gesetzt werden.

Beispiel

Ein Array mit 10 Elementen vom Typ **`string`** ist zu durchsuchen. Der zu suchende Wert („*Paul*“) soll im ungünstigsten Fall, also ganz am Ende des Suchvorgangs, gefunden werden. Nur zur besseren Nachvollziehbarkeit sind die Indexe oberhalb der Daten angegeben.

0	1	2	3	4	5	6	7	8	9
Alfons	Carla	Emil	Jonas	Luna	Nadine	Paul	Ralf	Sabine	Theo

Zuerst das mittlere Element bestimmen. Hier wird die Mitte folgendermaßen bestimmt: „*(höchster Indexwert minus niedrigster Indexwert) / 2 + niedrigster Indexwert*“, wobei die Divison eine Integerdivision ist. In diesem Beispiel also: (9 - 0) / 2 + 0 → 4.

0	1	2	3	**4**	5	6	7	8	9
Alfons	Carla	Emil	Jonas	**Luna**	Nadine	Paul	Ralf	Sabine	Theo

Der zu suchende Name „Paul“ befindet sich in der zweiten Hälfte!
Nun wieder den Index für das mittlere Element der zweiten Hälfte bestimmen: (9 - 5) / 2 + 5 → 7.

0	*1*	*2*	*3*	*4*	5	6	**7**	8	9
Alfons	*Carla*	*Emil*	*Jonas*	*Luna*	Nadine	Paul	**Ralf**	Sabine	Theo

Der zu suchende Name „Paul“ befindet sich in der jetzigen ersten Hälfte!
Nun wieder den Index für das mittlere Element der zweiten Hälfte bestimmen: (6 - 5) / 2 + 5 → 5.

0	*1*	*2*	*3*	*4*	**5**	6	*7*	*8*	*9*
Alfons	*Carla*	*Emil*	*Jonas*	*Luna*	**Nadine**	Paul	*Ralf*	*Sabine*	*Theo*

Der zu suchende Name „Paul“ befindet sich in der jetzigen zweiten Hälfte!
Nun wieder den Index für das mittlere Element der zweiten Hälfte bestimmen: (6 - 6) / 2 + 6 → 6.

0	*1*	*2*	*3*	*4*	*5*	**6**	*7*	*8*	*9*
Alfons	*Carla*	*Emil*	*Jonas*	*Luna*	*Nadine*	**Paul**	*Ralf*	*Sabine*	*Theo*

Mit dem vierten Suchschritt ist das gesuchte Element gefunden. Wäre es nicht in der Suchmenge enthalten, würde die Suche nun ebenfalls enden, da durch die Neuberechnung die Obergrenze nun kleiner als die Untergrenze wäre.

Im ungünstigsten Fall (wie dargestellt) beträgt die Anzahl der notwendigen Vergleiche $\log_2(n)$, wobei n die Anzahl der Elemente darstellt.

In unserem Beispiel wäre dies $\log_2(10) = 3{,}3219$. Dieser Wert muss aufgerundet werden; es ergeben sich also vier Vergleiche bis im ungünstigsten Fall (gefunden oder nicht vorhanden) das Ergebnis feststeht.

Verschiedene Suchmengen

n = 1.000	→	$\log_2(1000) = 9{,}9657$	→	max. 10 Vergleiche
n = 10.000	→	$\log_2(10000) = 13{,}2877$	→	max. 14 Vergleiche
n = 100.000	→	$\log_2(100000) = 16{,}6096$	→	max. 17 Vergleiche
n = 1.000.000	→	$\log_2(1000000) = 19{,}9315$	→	max. 20 Vergleiche

Die Zeitfunktion der binäre Suche gehört damit zur Komplexitätsklasse *O(log n)*.

Implementation der binären Suche

Die binäre Suche lässt sich iterativ aber auch rekursiv implementieren. Eine iterative Variante ist -analog zur linearen Suche- als statische generische Klasse **`BinarySeek`** angegeben.

```
public static class BinarySeek<T> where T : IComparable
{
    public static int FoundIndex { get; private set; }

    public static bool IsFound { get; private set; }

    public static int SeekSteps { get; private set; }

    private static int mitte;

    public static bool SeekIterative(T[] array, T seekelement)
    {
        // Guard Clauses
        if (array == null) throw new ArgumentNullException("array");
        if (seekelement == null) throw new ArgumentNullException("seekelement");

        FoundIndex = 0; IsFound = false; SeekSteps = 0;

        int unteregrenze = 0;
        int oberegrenze = array.Length - 1;

        while (unteregrenze <= oberegrenze)
        {
            mitte = unteregrenze + (oberegrenze - unteregrenze) / 2;

            // array[mitte] < seekelement --> -1
            // array[mitte] > seekelement --> 1
            // array[mitte] = seekelement --> 0

            int rc = array[mitte].CompareTo(seekelement);
            if (rc == 1)
                oberegrenze = mitte - 1;
            else if (rc == -1)
                unteregrenze = mitte + 1;
            else
            {
                FoundIndex = mitte;
                IsFound = true;
                break;
            }
            SeekSteps++;
        }
        return IsFound;
    }
}
```

Aufgabe

Wie bei der linearen Suche! Informieren Sie sich über eine **rekursive** Variante!