# Algorithmen und Datenstrukturen
## Lineare Datenstrukturen II

## Aufgaben – Stack

**1. Aufgabe**

**Generischer Stack**
Implementieren Sie einen generischen `Stack<T>` mit den Operationen `Push(x)` und `x = Pop()`. `Pop` soll dasjenige Element liefern, das zuletzt mit `Push` gespeichert wurde. Implementieren Sie auch ein Property `Size`, das die Anzahl der Elemente im Stack liefert. Schreiben Sie ein Testprogramm, das Kommandozeilenargumente in einem `Stack<string>` und die Längen der Kommandozeilenargumente in einem `Stack<int>` ablegt.

**2. Aufgabe**

**Vererbung**
Implementieren Sie eine generische Klasse `StackExtended<T>` als Unterklasse der in Aufgabe 1 implementierten Klasse `Stack<T>`. Darin soll es eine Methode `Contains(x)` geben, die prüft, ob das Element `x` im Stack vorhanden ist oder nicht. Ferner soll es einen Indexer geben, mit dem man auf die einzelnen Stack-Elemente zugreifen kann.

**3. Aufgabe**

**Fibonacci**
Verwenden Sie einen Stack, um die Fibonacci-Folge iterativ bis zu einer gewünschten Zahl zu berechnen. Die gewünschte Zahl soll von der Console eingelesen werden. Für die Berechnung dürfen Sie ausschliesslich einen Stack<long> (gemäss Aufgabe 1) und eine for-Schlaufe verwenden. Am Schluss soll das Resultat auf der Console ausgegeben werden.

Beispiele:
Eingabe = 7 → Ausgabe = 13
Eingabe = 15 → Ausgabe = 610
Eingabe = 33 → Ausgabe = 3524578

# Aufgaben - Hashtable

**4. Aufgabe**

**Hashtable**

Implementieren Sie eine eigene Hashtable. Die Objekte, welche in der Hashtable gespeichert werden, sollen vom Typ Element sein:

```
public class Element {
    public string Id;
    public string Name;
}
```

Die Hashtable soll folgendes Interface implementieren:

```
public interface IHashtable {

    /// <summary>
    /// Element in der Hashtabelle einfügen
    /// </summary>
    /// <param name="e">einzufügendes Element</param>
    /// <returns>
    /// true: Element wurde eingefügt;
    /// false: Hashtabelle voll; Element nicht eingefügt
    /// </returns>
    bool Put(Element e);


    /// <summary>
    /// Element in der Hashtabelle suchen
    /// </summary>
    /// <param name="id">Schlüssel des zu suchenden Elementes</param>
    /// <returns>
    /// gesuchtes Element;
    /// null-> Element nicht gefunden
    /// </returns>
    Element Get(string id);

    /// <summary>
    /// Element in der Hashtabelle löschen
    /// </summary>
    /// <param name="id">Schlüssel des zu löschenden Elementes</param>
    /// <returns>
    /// true: Element wurde gelöscht;
    /// false: Element nicht gefunden
    /// </returns>
    bool Delete(string id);
}
```

Die Hashtable soll «geschlossenes Hashing» verwenden.

**5. Aufgabe**

**Crossreferenztabelle**

Schreiben Sie ein Programm, welches ein C#-Programm einliest (d.h. *.cs-Datei) und in einer Hashtable alle C#-Keywords dieser Datei speichert. Zudem sollen für jeden Namen die Zeilennummern gespeichert werden, auf denen der Name vorkommt. Diese Zeilennummern sollen in einer einfach verketteten Liste abgelegt werden.

Das Programm muss zudem folgende Funktionen zur Verfügung stellen:

- Ausgabe der gesamten Hashtabelle (alphabetisch sortiert)
- Ausgabe der Zeilennummern auf welcher ein bestimmter Name vorkommt.

Die Datenstruktur besteht also aus einer Hashtable. Als Schlüssel zur Berechnung der Primärindizes werden die C#-Keywords verwendet. Zur Speicherung der Zeilennummern soll eine einfache verkettete Liste verwendet werden, deren Anker in den Elementen der Hashtable enthalten ist.

Sie können wahlweise Ihre Hashtable oder die .NET-Klasse Hashtable verwenden.