Arrays, Stacks, Strings*

Hans Jürgen Ohlbach

23. März 2018

Keywords: Arrays, Stacks, Strings, binäre Suche in Arrays

Empfohlene Vorkenntnisse: Primitive Datentypen, C oder Java ist hilfreich

Inhaltsverzeichnis

*Dieser Text ist Teil einer Sammlung von Miniskripten zur Einführung in die Informatik. Er ist in erster Linien für Nichtinformatiker gedacht, kann aber natürlich auch als erste Einführung für Informatiker nützlich sein.

1 Einleitung

In vielen Programmen benötigt man größere Mengen von Daten des gleichen Typs. Ein Buch, z.B. besteht aus einer Menge von Kapiteln, ein Kapitel aus einer Menge von Paragraphen, ein Paragraph aus einer Menge von Zeilen, eine Zeile aus einer Menge von Wörtern und schließlich ein Wort aus einer Menge von Buchstaben. All diese Mengen sind zudem geordnet: das 1. Kapitel, das 2. Kapitel usw.

Eine sehr gute Datenstruktur, um auf diese geordnete Menge von Objekten zuzugreifen, ist das *Array*. Ein (eindimensionales) Array ist einfach eine im Speicher hintereinanderliegende Sequenz von Daten *des gleichen Typs*. Dass die Daten hintereinander liegen, und von gleicher Länge sind, bedeutet, dass man auf jedes Element zugreifen kann, indem man seine Adresse aus der Anfangsadresse, der Länge der Elemente im Speicher, und der Nummer des Elements berechnet.

In diesem Miniskript werden wir zunächst eindimensionale Arrays und dann auch mehrdimensionale Arrays einführen. In der Programmiersprache C kann man verdeutlichen, wie genau die Arrays im Speicher abgelegt werden. Zusätzlich werden auch die Arrays in Java behandelt.

Die ursprüngliche Version des Array-Typs ist *statisch*, d.h. man muss die Länge eines Arrays beim Erzeugen des Arrays bestimmen, und kann sie später nicht mehr verändern[1] . Inzwischen gibt es aber auch *dynamische Arrays*, deren Länge veränderbar ist.

Strings (Zeichenketten) sind im Prinzip Arrays von Buchstaben. Wenn es so einfach wäre, gäbe es nicht viel mehr dazu zu sagen. Mit der Einführung von Unicode, und den Kodierungen mit UTF-8 und UTF-16 ist die Länge der Buchstaben leider nicht mehr gleich. Daher sind Strings komplexer als normale Arrays. Dies werden wir im Kapitel 8 kennenlernen.

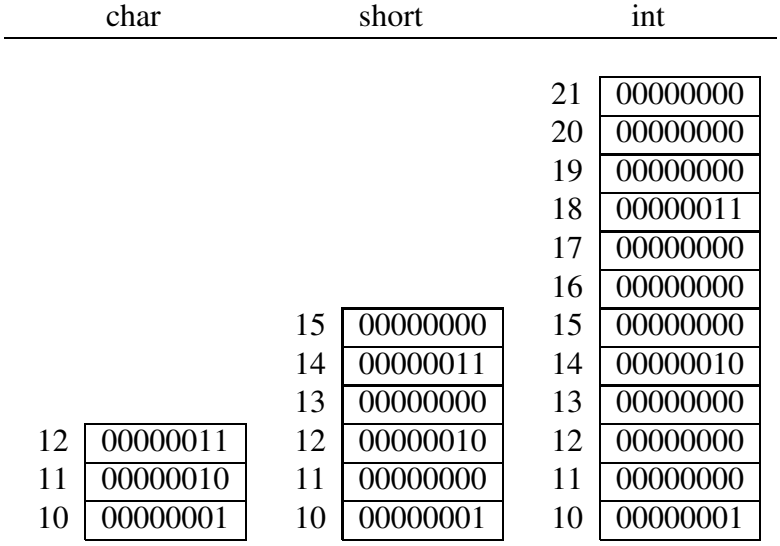
Eine sehr nützliche Verwendung von Arrays sind Stacks, die in Kapitel 7 vorgestellt werden.

2 Statische eindimensionale Arrays

Wie schon in der Einleitung gesagt, ist ein Array eine im Speicher hintereinanderliegende Sequenz von Daten *des gleichen Typs*. „Des gleichen Typs“ bedeutet insbesondere *von gleicher Länge*. Dies ist wichtig, um auf effiziente Weise die Adresse jedes Arrayelements berechnen zu können. Wir betrachten zunächst eindimensionale Arrays, und verallgemeinern dann auf mehrdimensionale Arrays.

Zunächst illustrieren wir die Speicherung der Zahlen 1,2,3 (binär 01,10,11) als char-Array (1 Byte), short-Array (2 Bytes) und int-Array (4 Bytes) ab der Speicherzelle 10:

[1] In der Anfangszeit der Programmiersprache Fortran musste sogar der Programmierer die Länge des Arrays als konkrete Zahl festlegen.



Die short- und int-Werte sind hier im little-Endian Format abgelegt. D.h. das niedrigstwertige Byte kommt an die kleinste Adresse. Bei char-Arrays spielt das keine Rolle.

Arrays kann man entweder ohne Initialisierung oder mit Initialisierung vereinbaren. In C und Java sieht die Vereinbarung eines int-Arrays mit 3 Zellen so aus:

	C	Java
ohne Initialisierung:	`int a[3];`	`int[] a = new int[3];`
mit Initialisierung:	`int a[3] = {1,2,3};`	`int[] a = {1,2,3};`

In beiden Fällen werden drei hintereinanderliegende int-Speicherzellen reserviert. „Ohne Initialisierung“ bedeutet in C, dass die Werte in den Zellen davon abhängen, was bisher in den Zellen, die dem Array als Speicherplatz zugewiesen wurden, gespeichert ist. Das sieht dann ziemlich zufällig aus. Bei Java werden die Zellen automatisch mit 0 initialisiert.

Man kann jedoch schon beim Anlegen des Arrays in die Zellen Werte speichern. Im Beispiel oben werden die Werte 1,2,3 in die Zellen gespeichert.

Mit `a[0], a[1], a[2]` kann man dann in C und in Java auf die einzelnen Werte zugreifen. Es ist ganz wichtig zu wissen, dass in C und in Java, und in vielen anderen Programmiersprachen, das erste Arrayelement *den Index 0* hat.

Der Index des letzten Arrayelements ist daher *Länge des Arrays - 1*.
Der Index des letzten Elements eines Arrays der Länge 3 ist 2 (und *nicht* 3)!

Adressberechnungen
In C kann man sogar auf die Adressen der Speicherzellen zugreifen. Mit folgendem C-Programm kann man sie ausdrucken:

```
#include <stdio.h>

int main(){
  int a[3] = {1,2,3};
  printf("%p␣%p\n",&a[0], &a[1]);}
```

`&a[0]` steht für die Adresse der ersten Speicherzelle. Mit der Direktive `%p` der printf-Funktion kann man sie (hexadezimal) ausdrucken. Der Ausdruck könnte z.B. so aussehen:
`0x7ffd0ea80420 0x7ffd0ea80424`

Wichtig ist dabei die Differenz zwischen den beiden Adressen: 4. Das ist genau die Speichergröße eines int-Wertes: 4 Bytes. Bei einem char-Array wäre die Differenz 1: ein Byte.

In Java kann man nicht auf die Adressen der Speicherzellen zugreifen. Das liegt daran, dass Java einen *Garbage Collector* benutzt. Der Garbage Collector bereinigt ab und zu nicht mehr benötigten Speicherplatz. Dabei kann er insbesondere auch Objekte im Speicher verschieben, wenn zwischen diesen Lücken sind. Wenige große Lücken lassen sich leichter verwalten als viele kleine. Wenn aber Objekte verschoben werden, dann stimmen ihre Adressen nicht mehr. Daher bietet Java nicht die Möglichkeit, mit Adressen zu rechnen.

Arrayzugriffe
Wie schon gesagt, bieten Arrays die Möglichkeit, direkt auf einzelnen Element zuzugreifen, indem man deren Adresse aus der Anfangsadresse des Arrays, der Länge der Zellen, und dem Index des Elements berechnet: **Anfangsadresse + index*Zellenlänge**.

In C kann man das sogar programmieren:

```
int main(){
  int a[3] = {1,2,3};
  int* anfang = &a[0];
  int index = 2;
  printf("%i\n",*(anfang+index));
}
```

(Das `#include <stdio.h>` lassen wir ab jetzt weg.)

`int*` in `int* anfang = &a[0];` ist der „Addresstyp“ für int-Werte. `anfang` ist die Adresse des ersten Array-Elements, und damit auch gleichzeitig die Anfangsadresse des Arrays selbst. `anfang+index` ist in diesem Fall die Adresse des Array-Elements an der Position 2. Das funktioniert deshalb, weil der Compiler wegen des Typs `int*` weiß, dass die Länge der Zellen 4 ist. Über den Adresstyp kann der Compiler immer die Zellenlänge ermitteln, und damit die Adresse des jeweiligen Elements berechnen. `*(anfang+index)` schließlich greift auf den Inhalt der Zelle zu. In diesem Fall würde die 3 ausgedruckt.

Allerdings würde man nicht auf diese umständliche Weise auf eine Arrayzelle zugreifen, sondern einfach mit `a[index]`. Der Compiler würde aber genau die oben angedeutet Berechnung programmieren.

An dieser Stelle besteht aber ein großer Unterschied zwischen C und Java. In C könnte man nach der Deklaration `int a[3] = {1,2,3};` *ohne Fehlermeldung* auf z.B. `a[-1]` oder `a[4]` zugreifen. Diese Zellen gehören überhaupt nicht zu dem Array, sondern liegen davor bzw. dahinter. Sie enthalten natürlich auch irgendwelche Werte, die das Programm dann weiterverarbeiten würde. (Das ist eine üble Fehlerquelle in C).

In Java gäbe es dabei aber eine Fehlermeldung `ArrayIndexOutOfBoundsException`.

Komplexität der Zugriffe
Die Adressberechnung für die Arrayzugriffe hängt nicht von der Länge des Arrays ab. Sie hängt nur von der Geschwindigkeit der arithmetischen Operationen ab. Da diese in einem Prozessor konstant ist, ist der Aufwand für die Adressberechnung ebenfalls konstant. Die dafür verwendete Formel ist $\mathcal{O}(1)$. Die 1 symbolisiert, dass der Aufwand nicht von der Größe der Eingabe abhängt, sondern konstant ist.

Schleifen über Arrays
Ein Array wird oft in einer Schleife weiterverarbeitet, z.B. in C:

```
int main(){
  int a[3] = {1,2,3};
  for(int i=0; i<3; ++i) {
    printf("%i\n",a[i]);}
}
```

Dieses Programm würde
1
2
3
ausdrucken.

In C muss man die Länge des Arrays zur Programmierzeit kennen, um die Schleife korrekt zu beenden. In Java ist das etwas komfortabler.

Den gleichen Ausdruck bekommt man mit folgendem Java Programm:

```
public static void main(String[] args) {
   int[] a = {1,2,3};
   for(int i = 0; i < a.length; ++i) {
     System.out.println(a[i]);}
   }
```

wobei `a.length` die Länge des Arrays angibt.

Noch komfortabler kann man die Schleife so programmieren:

```
public static void main(String[] args) {
   int[] a = {1,2,3};
   for(int e:a) {
      System.out.println(e);}
   }
```

Die Schleife `for(int e:a)` bindet nacheinander die Variable `e` an die Arrayelemente.

3 Arrays von komplexeren Objekten

Man kann natürlich nicht nur Arrays von primitiven Datentypen bilden, sondern Arrays von *beliebigen* Datentypen.

Komplexere Datentypen kann man in C durch die `struct`-Anweisung bilden.

Beispiel:

```
struct Koordinate {
  int x;
  int y;};
```

Dies definiert einen Datentyp `Koordinate` mit zwei int-Werten, x und y. Die Länge eines `Koordinate`-Objekts ist damit 2*4=8 Bytes.

Auf folgende Weise könnte man jetzt ein Array von Koordinaten definieren.

```
int main(){
  struct Koordinate a[3];
  a[0].x=5; a[0].y=10;
  a[1].x=6; a[1].y=12;
  a[2].x=7; a[2].y=14;
  printf("%i␣%i\n",a[1].x,a[1].y);
}
```

Die Anweisungen `a[0].x=5; a[0].y=10;` usw. füllen das Array mit Koordinatenwerten.
`a[1].x,a[1].y` greifen auf die x- und y-Koordinaten des Arrayelements mit dem Index 1 zu. Ausgedruckt würde daher 6 12.

Arrays von komplexeren Objekten in Java
In Java ist die Bildung von Arrays von komplexeren Objekten zunächst etwas umständlicher. Komplexere Objekte definiert man als Klassen, z.B.:

```
public class Koordinate {
    int x;
    int y;}
```

Üblich wäre aber zusätzlich noch die Definition eines *Konstruktors*, z.B.

```
public Koordinate(int x, int y) {
        this.x = x; this.y = y;}
```

Jetzt kann man mit `new Koordinate(3,5)` ein *Koordinate*-Objekt erzeugen.

Ein Array von Koordinate-Objekten erhält man so:

```
public static void main(String[] args) {
  Koordinate[] a = new Koordinate[3];
  a[0] = new Koordinate(3,5);
  a[1] = new Koordinate(4,6);
  a[2] = new Koordinate(5,7);
  System.out.println(a[1].x);}
```

Man kann das auch abkürzen:

```
public static void main(String[] args) {
  Koordinate[] a = {new Koordinate(3,5),
                    new Koordinate(4,6),
                    new Koordinate(5,7)};
  System.out.println(a[1].x);}
```

In beiden Fällen würde 4 ausgedruckt.

Unterschied C vs. Java: In C bewirkt die Anweisung `struct Koordinate a[3];` die Bereitstellung von 3 Speicherzellen für je ein `Koordinate`-Objekt, also insgesamt 6 int-Zellen (6*4 = 24 Bytes).

Eine entsprechende Anweisung `Koordinate[] a = new Koordinate[3];` bewirkt zunächst lediglich die Bereitstellung von 3 Speicherzellen für *Adressen von Koordinate-Objekten*, nicht der Objekte selbst. Diese müssen zunächst mit dem Konstruktor gebildet werden, und dann deren Adressen in das Array eingefügt werden.

Eine Speicherbelegung für ein dreielementiges Koordinate-Array mit den Werten (5,10), (6,11) und (7,12) könnte in C und Java so aussehen:

C			Java		
			42	0000 1100	12 (a[2].y)
			38	0000 0111	7 (a[2].x)
			34	0000 1011	11 (a[1].y)
30	0000 1100	12 (a[2].y)	30	0000 0110	6 (a[1].x)
26	0000 0111	7 (a[2].x)	26	0000 0110	10 (a[0].y)
22	0000 1011	11 (a[1].y)	22	0000 0101	5 (a[0].x)
18	0000 0110	6 (a[1].x)	18	0010 0110	38: a[2]
14	0000 0110	10 (a[0].y)	14	0001 1110	30: a[1]
10	0000 0101	5 (a[0].x)	10	0001 0110	22: a[0]

Die Byte-Zellen, die nur Nullen enthalten, sind weggelassen.

Für Java stimmt das nicht ganz, denn ein Koordinate-Objekt in Java hätte neben den x- und y-Werten noch andere Daten, z.B. einen Zeiger auf die Klasse selbst. Daher wären die Einträge länger als je zwei int-Zellen. In Java müssten die Koordinate-Objekte auch nicht direkt am Anschluss des Arrays abgelegt werden. Sie könnten an beliebiger Stelle im Speicher stehen. Ihre Adressen im Array müssten natürlich auf die Stelle zeigen, wo sie tatsächlich abgelegt sind.

Der Programmierer sieht allerdings nicht unmittelbar etwas davon, sondern erst dann wenn er Fehler macht. Die Sequenz

```
Koordinate[] a = new Koordinate[3];
System.out.println(a[1].x);}
```

würde eine Fehlermeldung produzieren, weil in `a[1]` noch keine Adresse eines Koordinate-Objekts gespeichert ist, sondern `null` (32 Nullen).

4 Statische mehrdimensionale Arrays

Die Verallgemeinerung von eindimensionalen Arrays sind natürlich zwei- und mehrdimensionale Arrays. Zweidimensionale Arrays können z.B. bei mathematischen Anwendungen zur Speicherung von Matrizen benutzt werden.

In C und Java können diese wiederum mit und ohne Initialisierung erzeugt werden. Um z.B. eine Matrix aus zwei Zeilen mit drei Spalten zu erzeugen, genügen folgende Anweisungen.

	C	Java
ohne Initialisierung:	`int a[2][3];`	`int[][] a = new int[2][3];`
mit Initialisierung:	`int a[2][3] = {{1,2,3},{4,5,6}}`	`int[][] a = {{1,2,3},{4,5,6}};`

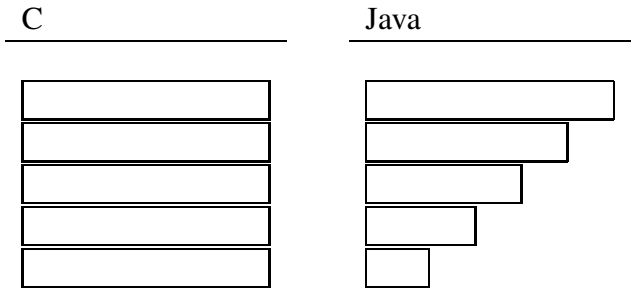
Sowohl in C als auch in Java lässt sich das auf beliebige Dimensionen verallgemeinern. Mit `a[0][1]` würde man dann auf das Element mit dem Wert 2 zugreifen.

Ein Unterschied zwischen C und Java ist wiederum, dass in Java die Arrayelement immer initialisiert werden, entweder mit 0 oder mit den gegebenen Werten. Bei C enthält ein uninitialisiertes Array die Werte, die zufällig vorher in den Speicherzellen waren.

Es gibt noch einen weiteren Unterschied zwischen C und Java. In C sind die Arrays immer rechteckig. Jede Zeile hat gleich viele Spalten. In Java muss das nicht so sein. In folgendem Programm wird ein zweidimensionales int-Array angelegt, dessen erste Zeile 4 Elemente hat, und dessen zweite Zeile 5 Elemente hat.

```
public static void main(String[] args) {
        int[][] a = new int[2][];
        a[0] = new int[4];
        a[1] = new int[5];
        a[1][4] = 3;
        System.out.println(a[1][4]);}
        }
```

Damit kann man z.B. Diagonalmatrizen, wie sie in machen mathematischen Verfahren gebraucht werden, speichereffizient realsisieren. Die Speicherplatzbelegung für eine Diagonalmatrix in C und in Java könnte dann in etwa so aussehen:



wobei in C die Hälfte des Platzes ungenutzt bliebe.

Adressberechnung
Für ein zweidimensionales Array in C ergibt sich in die Adresse einer Zelle aus
Anfangsadresse + (Zeilenindex * Zeilenlänge + Spaltenindex) * Blockgröße.
In Java muss zunächst die Adresse der jeweiligen Zeile berechnet werden. Dort findet man die Adresse der Zeile, und daraus ergibt sich die Adresse des Arrayelements. Die Berechnung ist in beiden Fällen ebenfalls nicht abhängig von der Größe des Arrays, sondern ergibt sich aus einer festen Zahl von Multiplikationen und Additionen, d.h. die Komplexität ist wiederum $\mathcal{O}(1)$.

Bei mehr als zwei Dimensionen ist die Berechnung analog.

Möglichkeiten zur Optimierung
Für Programmierer, die extrem effizient programmieren wollen, ist die Reihenfolge der Speicherung der Arrayzeilen interessant.

Das kann man z.B. mit folgendem Programmstück herausbekommen:

```
int a[2][3] = {{1,2,3},{4,5,6}};
printf("%p␣%p␣\n",&a[0][0],&a[0][1]);
```

Eine Ausgabe könnte sein:
0x7ffded16b590 0x7ffded16b594
Das bedeutet, dass die Werte zeilenweise abgespeichert werden.

Warum ist das interessant? Man betrachte folgende Code-Schnipsel:

```
int a[1000][1000] = ...;
for(int i = 0; i < 1000; ++i) {
  for(int j = 0; j < 1000; ++j) {
      ... a[i][j] ...}}
```

und

```
int a[1000][1000] = ...;
for(int i = 0; i < 1000; ++i) {
  for(int j = 0; j < 1000; ++j) {
      ... a[j][i] ...}}
```

Im ersten Fall wird das Array zeilenweise abgelaufen, und im zweiten Fall spaltenweise.

Moderne Computer haben zwischen dem Prozessor und dem Arbeitsspeicher einen oder mehrere schnelle Zwischenspeicher, die sog. *Caches*. In diesen werden ganze Blöcke aus dem Arbeitsspeicher geladen, um den Zugriff schneller zu machen. Ein Zugriff auf eine bestimmte Zelle im Arbeitsspeicher bewirkt daher oft, dass auch ein größerer Block aus der Umgebung der Zelle in den Cache geladen wird, in der Erwartung, dass diese Elemente auch benötigt werden. Da die Arrays zeilenweise gespeichert werden, bewirkt ein Zugriff auf ein Element einer Zeile, dass auch andere Elemente dieser Zeile in den Cache geladen werden. Läuft die Schleife die Elemente zeilenweise ab, dann beschleunigt dieser Mechanismus die Abarbeitung enorm. Läuft die Schleife die Elemente aber spaltenweise ab, dann ist der Cachemechanismus völlig nutzlos, und eher kontraproduktiv.

Man kann sich ein 2D Array vorstellen als einen Schrank mit Schubladen mit Kästchen. Durchläuft man das Array zeilenweise, öffnet man je in einer Schublade alle Kästchen. Durchläuft man es spaltenweise öffnet man in allen Schubladen je nur ein Kästchen

5 Dynamische Arrays

Statische Arrays haben den Nachteil, dass ihre Größe bei der Erzeugung festgelegt werden muss, und später nicht mehr geändert werden kann. Oft weiß man die Größe dann aber noch nicht. Sie ergibt sich vielleicht erst im Lauf der Berechnung, oder wenn man Daten von irgendwo her einliest und in einem Array abspeichern will. C und Java bieten dazu in der Programmiersprache selbst keine Lösung. Allerdings gibt es Bibliotheken, die solche *dynamischen Arrays* zur Verfügung stellen. Die Idee dabei ist, zunächst ein Array fester Größe zu erzeugen. Solange diese Größe ausreicht, wird damit so gearbeitet, wie in einem statischen Array. Sobald aber mehr Elemente gebraucht werden als vorgesehen, wird ein komplett neues, größeres Array angelegt, und das alte Array dort hinein kopiert. Nach außen sieht das Array dann so aus wie vorher, hat aber mehr Speicherzellen.

Leider gibt es das nicht in C. Nur in C++ gibt es die Klasse `vector`, die das ermöglicht. Ein Beispielprogramm wäre:

```
#include <vector>

int main(){
    std::vector<int> v;
    for (int i=0; i<100; ++i) {
        v.push_back(i); // Fuegt i ans Ende von v an.
        ++v[i];         // v[i] muss bereits existieren.
    }}
```

In Java gibt es die Klasse `ArrayList`, die eine ähnliche Funktion hat. Ein Programm damit wäre:

```
public static void main(String[] args) {
    ArrayList<Integer> a= new ArrayList();
    for(int i = 0; i < 100; ++i) {
        a.add(i);}      // Fuegt die Zahl i ans Ende des Arrays an.
    System.out.println(a.get(99));}
```

Es gibt jedoch auch nichttriviale Unterschiede zur vector-Klasse in C++:

- Die ArrayList-Klasse erzeugt Arrays für Objekttypen, aber *nicht für primitive Datentypen*. Man muss also dann die Wrapper-Klassen benutzen, z.B. Integer statt int.
- Der Zugriff auf das Array geschieht in Java über die Methodenaufrufe get und add. In C++ wird weiterhin die Arraysyntax verwendet, z.B. `++v[i]` erhöht den Wert an der Stelle `v[i]` um 1. Das ist möglich, da man in C++ jeden Operator für jeden Datentyp speziell definieren kann. Das geht in Java nicht.

6 Binäre Suche in Arrays

In vielen Anwendungen speichern Arrays Objekte, für die eine Totalordnung definiert ist. Das einfachste Beispiel sind natürliche Zahlen. Ein anderes Beispiel wären Strings mit der lexikographischen Ordnung. Ist das Array sortiert, dann kann man mit einem sehr schnellen Algorithmus ein bestimmtes Element im Array auffinden: nämlich durch fortgesetzte Halbierung.

Das Verfahren macht (fast) jeder, wenn er z.B. ein Wort in einem dicken Lexikon sucht. Man schlägt das Lexikon in der Mitte auf. Entweder das Wort ist schon auf der Seite, dann hat man es gefunden. Falls nicht, muss es in der ersten Hälfte oder in der zweiten Hälfte sein. Falls es in der ersten Hälfte ist, schlägt man diese in der Mitte auf, und so weiter.

Wir illustrieren das Verfahren an folgendem Array:

13	15	19	25	30	47	48	49	60	61	63	80	83	85	90	95

Angenommen, wir suchen die 30.

Das Array wird halbiert.

13	15	19	25	30	47	48	49	60	61	63	80	83	85	90	95

Die 30 liegt in der linken Hälfte. Diese wird wiederum halbiert.

13	15	19	25	30	47	48	49

Jetzt liegt die 30 in der rechten Hälfte. Diese muss halbiert werden:

30	47	48	49

Jetzt liegt die 30 wiederum in der linken Hälfte. Diese muss halbiert werden:

30	47

Danach wird die 30 in der linken Hälfte gefunden.

In diesem Beispiel hatten die Hälften immer eine gerade Anzahl Zellen. Falls das nicht so ist, bekommt man halt eine „Hälfte“ mit einer Zelle mehr.

Die Effizienz des Halbierungsverfahrens wird bestimmt durch die mögliche Anzahl von Teilungen einer Zahl n, nämlich $\lceil \log_2(n) \rceil$ mögliche Teilungen. Das Beispiel illustriert das.

$$\underbrace{1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1}_{10 \text{ mal halbiert}}$$

Als Zeitkomplexität der binären Suche ergibt sich damit $\mathcal{O}(log(n))$.

Binäre Suche funktioniert nur wenn das Array sortiert ist, und die Einträge sich nicht allzu oft ändern. Bei Anwendungen, wo man öfter Elemente hinzu nehmen oder wegnehmen muss, sollte man statt Arrays eher *Suchbäume* verwenden, z.B. ALC-Bäume oder B-Bäume.

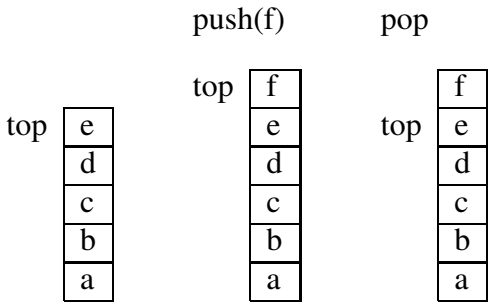
7 Stacks (Kellerspeicher)

Eine sehr häufige Verwendung von Arrays sind sog. *Stacks*. Ein Stack ist eine Art dynamisches Array, bei dem man nur Zugriffe vom Ende her hat.

Angenommen, das Array ist bis zur Zelle 5 gefüllt, mit Buchstaben a,b,c,d,e. Jetzt hat man folgende Möglichkeiten:

- Man kann auf die Zelle 5 zugreifen (das *Top of Stack*) (Funktion top).
  Auf andere Elemente gibt es keinen Zugriff.
- Man kann die Zelle 5 löschen (Funktion pop), und damit das Top of Stack auf die Zelle 4 zurücksetzen,
- Man kann ein neues Element anhängen (Funktion push). Das Top of Stack ist dann die Zelle 6.

Eine Sequenz `top(), push(f), pop()` könnte folgende Veränderung bewirken:



Meist macht man das Stack nicht wirklich dynamisch, sondern initialisiert es ausreichend groß. Falls es doch überläuft, gibt es eine Fehlermeldung.

Man kann natürlich auch das Stack nach unten wachsen lassen. Dann beginnt man mit einer hohen Adresse und fügt ein neues Element unten an.

Eine einfache Java-Implementierung für ein int-Stack wäre:

```
public class Stack {
    int[] stack;
    int topOfStack = -1;
    int size;

    public Stack(int size) {
        this.size = size;
        stack = new int[size];}

    public int top() {
        if(topOfStack < 0) {
                System.err.println("Stack␣Empty"); return 0;}
        return stack[topOfStack];}

    public void push(int item) {
        if(topOfStack == size-1) {
                System.err.println("Stack␣Overflow"); return;}
        stack[++topOfStack] = item;}

    public void pop() {
        --topOfStack;}
}
```

Das könnte man z.B. so testen:

```
 public static void main(String[] args) {
   Stack st = new Stack(2);
   st.push(4); st.push(5);
   System.out.println(st.top());
   st.pop();
   System.out.println(st.top());}
```

Die Ausgabe wäre
5
4

In Java gibt es schon die generische Klasse `Stack<E>`, die für beliebige Objekttypen verwendet werden kann, aber wiederum nicht für primitive Datentypen wie int usw. Für C müsste man ein Stack selbst implementieren. In C++ gibt es dagegen das Template[2] `stack`.

[2]Ein Template in C++ ist so ähnlich wie eine generische Klasse in C. Man kann es mit konkreten Klassen instanziieren.

8 Strings (Zeichenketten)

Zeichenketten sind einfach Folgen von Zeichen. Man könnte sie daher einfach als Character-Arrays realisieren. Leider ist es heutzutage nicht so einfach. Zu Beginn der Computerei gab es im wesentlichen als Zeichentyp den ASCII-Zeichensatz[3]. Da ein ASCII-Buchstabe in 8 Bits passt, kann man eine Zeichenkette als char-Array programmieren.

Mit der Entwicklung von Unicode reichen aber 8 Bits nicht mehr. Für die erste Ebene von Unicode braucht man 16 Bits pro Zeichen, und für die weiteren Ebenen 21 Bits pro Zeichen. Allerdings stimmt die ASCII-Kodierung mit dem ASCII-Fragment von Unicode überein. 21 Bits passen in einen 32-Bit int-Block. Man könnte also eine Unicode Zeichenkette als int-Array programmieren (das ist die UTF-32 Kodierung). Da in den meisten westlichen Texte die allermeisten Buchstaben 7-Bit ASCII-Buchstaben sind, wären von den 32 Bits eines int-Blocks, immer 25 Bits gleich 0. Um diese Verschwendung einzudämmen, hat man kompaktere Darstellungen von Unicode-Zeichenketten entwickelt: UTF-8 und UTF-16.

UTF-8
Die Idee für UTF-8 ist folgende: Jeder 7-Bit ASCII-Buchstabe passt in ein Byte, mit einer führenden 0. Solange die Buchstaben nur aus ASCII bestehen, bleibt es so. Kommt jedoch ein Unicode-Buchstabe, der nicht im ASCII-Fragment liegt, dann muss das führende Bit, was sonst 0 ist, 1 sein. Daran kann man erkennen, dass etwas Besonderes vorliegt. Danach kommt die Information, aus wie vielen Bytes der Unicode-Buchstabe besteht. Die 21 Unicode Bits werden dann auf bis zu 4 Bytes verteilt.

UTF-8 wird meist zur kompakten Darstellung von Texten auf Files verwendet, weniger zur Darstellung von Zeichenketten im Arbeitsspeicher.

UTF-16
Die UTF-16 Kodierung besteht aus Blöcken von 16 Bits. Damit lässt sich die erste Unicode-Ebene, die *Basic Multilingual Plane* (BMP) direkt kodieren. Kommt ein Zeichen, welches nicht in der BMP liegt, dann werden 20 Bits der Ebenen 0 bis 15[4] auf zwei 16-Bit Blöcke verteilt, wobei bestimmte festgelegte Bitmuster der Erkennung dieses Falls dienen.

Beispiele:

Buchstabe	Unicode	binär	UTF-16
Buchstabe ä	U+00E4	00000000 11100100	00000000 11100100
Eurozeichen €	U+20AC	00100000 10101100	00100000 10101100
Violinschlüssel	U+1D11E	00000001 11010001 00011110	**11011**00 00110100 **11011**01 00011110

Die fettgedruckten Bits sind die Markierungen, an denen man erkennt, dass zwei 16-Bit Blöcke gebraucht werden.

[3] Es gab auch noch andere Kodierungen, z.B. EBCDIC für IBM-Großrechner.
[4] Ebene 16, die Supplementary Private Use Area-B (PUA-B), wird nicht kodiert.

Strings in Java
Die Zeichendarstellung in Java basiert schon seit Beginn auf Unicode. Der char-Typ in Java hat 16 Bits. Damit lassen sich alle Zeichen der BMP direkt kodieren. Für alle anderen Zeichen muss man eine int-Variable benutzen.

Die String-Klasse allerdings kodiert Unicode-Zeichen in UTF-16. Bleibt man in der BMP, dann kann man aus einem String Zeichen für Zeichen als char-Wert extrahieren. Das geht mit der Methode `char charAt(int index)`. Liegt der Buchstabe jedoch außerhalb der BMP, dann braucht man die Methode `int codePointAt(int index)`. Sie liefert den Buchstaben als int-Wert zurück.

ICU
Unicode ist nicht direkt in C und C++ eingebaut. Es gibt aber eine große Bibliothek: **International Components for Unicode** (ICU), welche umfangreiche Funktionen zum Umgang mit Unicode für C, C++ und Java zur Verfügung stellt.

Stichwortverzeichnis