3 Datenstrukturen 1

Lernziele

- Kenntnis einiger (weniger) grundlegender Datenstrukturen
- Umgang mit den Datenstrukturen
- Algorithmen auf den Datenstrukturen

Felder

Merkmale

- Grundlegende Datenstruktur – in praktisch allen Programmiersprachen verfügbar
- Feste, geordnete Ansammlung von einzelnen Daten
- Einzeldaten haben alle den gleichen Datentyp
- Zugriff auf ein Element erfolgt stets in konstanter Zeit
- Zugriff auf ein Element erfolgt durch einen Index
- Indizes sind ganze Zahlen (oder verlustlos in ganze Zahlen konvertierbar)
- Felder stehen in direktem Zusammenhang zum Speichersystem des Rechners
- Größe muss (oft) im Voraus bekannt sein
- Es sind ein- oder mehrdimensionale Felder möglich

Definition und Verwendung in C: 1D-Feld

```
const int N = 10;

int A[N];

for (i = 0; i < N; i++)
{
    A[i] = 2 * i;
}
```

Definition und Verwendung in C: 2D-Feld

```
const int M = 10;
const int N = 5;

int A[M] [N];

for (i = 0; i < M; i++)
{
    for (j = 0; j < N; j++)
    {
        A[i][j] = i * j;
    }
}
```

Speicherabbildungsfunktion

- Wenn in einem Programm auf Elemente eines Feldes zugegriffen werden soll, muss man deren Speicheradresse berechnen.
- Dies erledigt die Speicherabbildungsfunktion. Wichtig ist dabei bei mehrdimensionalen Arrays die Speicherreihenfolge („row-major order“,
- „column-major order“). Wir gehen im Folgenden von row-major order aus.
- Normalerweise erledigt diese Aufgabe der Compiler bzw. die Laufzeitumgebung des Compilers.
- In Assemblersprachen bleibt diese Aufgabe dem Programmierer überlassen.

Speicherabbildungsfunktion – 1D-Array

In einem Array A mit 10 Elementen A[0] … A[9] soll das Element mit Index 7 adressiert werden.

- Die Position im Array ergibt sich direkt aus dem Index, pos = 7
- Die wirkliche Position im Speicher berechnet sich aus der Anfangsadresse des Arrays, der Position des Elements im Array sowie der Länge der einzelnen Elemente l_A (abh. Vom Basisdatentyp), in unserem Fall
- Auf den folgenden Folien wird nur noch die Formel für die Position angegeben.

Speicherabbildungsfunktion – 1D-Array

In einem Array A mit Elementen A[i] … A[k] soll das Element mit Index j adressiert werden.

Die Position im Array ergibt sich zu

$$pos = (j - i)$$

Hinweis:

Im Folgenden werden mehrere Arrayelemente A[i] … A[k] als A[i : k] notiert.

Speicherabbildungsfunktion – 1D-Array

Spezialfall $i=0$:

In einem Array A mit Elementen $A[0 : k]$ soll das Element mit Index j adressiert werden.

Die Position im Array ergibt sich zu

$$pos = j$$

Speicherabbildungsfunktion - 2D-Array

In einem 2D Array A mit Elementen A[0 : k1, 0 : k2] soll das Element A[j1, j2] adressiert werden.

Die Position im Array ergibt sich zu

$$pos = j_1 * (k_2 + 1) + j_2$$

Speicherabbildungsfunktion – 3D-Array

In einem 3D Array A mit Elementen $A[0{:}k_1, 0{:}k_2, 0{:}k_3]$ soll das Element $A[j_1, j_2, j_3]$ adressiert werden.

Die Position im Array ergibt sich zu

$$pos = j_1 * (k_2+1)(k_3+1) + j_2 * (k_3+1) + j_3$$

Speicherabbildungsfunktion

Etwas allgemeiner kann man die Position in einem nD Array A[0:k1,0:k2,…0:kn] berechnen zu

$$pos = \left[\sum_{d=1}^{n-1} j_d \cdot \prod_{t=d+1}^{n} (k_t + 1) \right] + j_n$$

Anwendungsbeispiel: Sieb des Erathostenes (Primzahlen)

Eine natürliche Zahl $p > 1$ heißt Primzahl, wenn sie ausschließlich durch sich selbst und durch 1 teilbar ist.

1. Schreibe alle Zahlen von 1 bis n auf
2. Streiche die 1
3. Streiche alle Vielfachen der bislang kleinsten Primzahl $p = 2$ („Aussieben“)
4. Wiederhole dies bis $p^2 > n$
5. Die nicht durchgestrichenen Zahlen sind Primzahlen

Anwendungsbeispiel: Sieb des Erathostenes

```
for i = 1 to N do
    Prim[i] = true
end for
Prim[1] = false
for i = 2 to √N do
    if Prim[i] then
        for j= 2i to N by i do
            Prim[j] = false
        end for
    end if
end for
for i = 1 to N do
  if Prim[i] then
        Ausgabe von i
  end if
end for
```

Anwendungsbeispiel: Sieb des Erathostenes

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40

Anwendungsbeispiel: Sieb des Erathostenes

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40

Anwendungsbeispiel: Sieb des Erathostenes

1 2 3 4 5 6 7 8 9 10

11 12 13 14 15 16 17 18 19 20

21 22 23 24 25 26 27 28 29 30

31 32 33 34 35 36 37 38 39 40

Anwendungsbeispiel: Sieb des Erathostenes

									1 2 3 4 5 6 7 8 9 10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40

Anwendungsbeispiel: Sieb des Erathostenes

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40

Beobachtung

- Beim Streichen der Vielfachen einer Zahl p genügt es, beim Quadrat p^2 zu beginnen
- Alle Zahlen n mit $2p <= n < p^2$ sind bereits gestrichen

Anwendungsbeispiel: Sieb des Erathostenes

```
for i=1 to N do
    Prim[i] = true
end for
Prim[1] = false
for i = 2 to √N do
    if Prim[i] then
        for j=i^2 to N by i do
            Prim[j] = false
        end for
    end if
end for
for i=1 to N do
    if Prim[i] then
        Ausgabe von i
    end if
```

Anwendungsbeispiel: Sieb des Erathostenes

```c
void erathostenes(int n)
{
    bool isPrime[N+1];

    isPrime[0] = false;
    isPrime[1] = false;

    for (int i=2; i<=N; i++) isPrime[i] = true;

    for (int i=2; i<=sqrt(N); i++)
    {
        if (isPrime[i])
        {
            for (int j=i*i; j<N; j=j+i) isPrime[j] = false;
        }
    }

    for (int i=1; i<=n; i++)
    {
        if (isPrime[i]) printf("%d  ", i);
    }
}
```

Anwendungsbeispiel: Sieb des Erathostenes

2	3	5	7	11	13	17	19	23	29	31	37	41	43
47	53	59	61	67	71	73	79	83	89	97			

Minimum und Maximum

Minimum

- Wie viele Vergleiche benötigt man, um in einer Menge A mit n Elementen das kleinste Element zu finden?
- Antwort: Man kommt mit n – 1 Vergleichen aus.
- Geht es besser? Gedankenexperiment:
  - Problem als Turnier mit n Teilnehmern
  - Jeder Vergleich ist ein Kampf zwischen 2 Teilnehmern
  - Das kleinere von beiden Elementen gewinnt
  - Jedes Element außer dem Gesamtgewinner muss genau einmal verlieren
  - Daraus ergeben sich n – 1 Kämpfe oder Vergleiche
- Insofern ist der im Folgenden vorgestellte Algorithmus optimal im Sinne der geringst möglichen Anzahl von Vergleichen

Minimum

```
function MINIMUM (A)
    min = A[1]
    for i = 2 to LENGTH(A) do
        if A[i] < min then
            min = A[i]
        end if
    end for
    return min
end function
```

Maximum

```
function MAXIMUM (A)
    max = A[1]
    for i = 2 to LENGTH(A) do
        if A[i] > max then
            max = A[i]
        end if
    end for
    return max
end function
```

Simultanes Finden von Minimum und Maximum

- Eigentlich leicht, indem man innerhalb der Schleife Minimum und Maximum mit je n-1 Vergleichen findet.
- Aber: Es geht schneller!

Simultanes Finden von Minimum und Maximum

- Vergleiche je zwei Elemente der Menge zunächst untereinander
- Das größere von beiden wird gegen das Maximum verglichen, das kleinere gegen das Minimum
- Daraus ergeben sich maximal $3\lfloor n/2 \rfloor$ Vergleiche.

Simultanes Finden von Minimum und Maximum 1

```
function SIMULTANEOUSMINMAX(A)
    if LENGTH(A) is odd then
        min = A[1]
        max = A[1]
        iStart = 2
    else
        iStart = 3
        if A[1] < A[2] then
            minIndx = 1, maxIndx = 2
        else
             minIndx = 2, maxIndx = 1
        end if
        min = A[minIndx]
        max = A[maxIndx]
    end if
```

Simultanes Finden von Minimum und Maximum 1

```
for i = iStart to LENGTH(A) by 2 do
   if A[i] < A [i+1] then
      minIndx = i , maxIndx = i+1
   else
      minIndx = i + 1, maxIndx = i
   end if
   if (A[minIndx] < min) then
      min = A[minIndx]
   end if
   if (A[maxIndx] > max) then
      max = A[maxIndx]
   end if
end for
return min, max
end function
```

Simultanes Finden von Minimum und Maximum

- Abzählen der Vergleiche ergibt
  - n ist ungerade: $1 + 3(n-1)/2 = 3n/2 – 1/2$ Vergleiche
  - n ist gerade: $2 + 3(n-2)/2 = 3n/2 – 1$ Vergleiche
- In Summe also höchstens $3\lfloor n/2 \rfloor$ Vergleiche
- Laufzeit ist also $\Theta(n)$.

Bubble-Sort – Ein einfaches Sortierverfahren

- Wähle von oben an jede Position im Feld (und bringe dort große Elemente hin)
- Beginne unten im Feld und vergleiche paarweise benachbarte Elemente
- Bringe das größere zweier Nachbarelemente durch Vertauschen nach oben
- Auf diese Weise wandern größere Zahlen wie Luftblasen im Wasser nach oben

Bubble-Sort

```
function BUBBLESORT(A,n)
    for i = n downto 2 do
        for j = 2 to i do
            if A[j-1] > A[j] then
                A[j-1] ↔ A[j]
            end if
        end for
    end for
end function
```

Bubble-Sort

Beispiel

Feld mit A = 2, 8, 7, 1, 3, 5, 6, 4

Bemerkungen

- Bubble-Sort gilt nicht als effizientes Sortierverfahren
- Laufzeit?

Bubble-Sort

Laufzeiten

$$T_{WC} \in O(n^2)$$
$$T_{BC} \in O(n)$$
$$T_{AC} \in O(n^2)$$

Dynamische Felder

- Felder haben normalerweise eine feste Größe
- Manche Programmiersprachen unterstützen Größenänderung bei Feldern
  - Z.B. in C: malloc, free, realloc
- Unterschied zwischen fester und variabler Größe ist die Zuordnung zu Programmspeicher oder Programm-Heap