Algorithmen und Datenstrukturen

Master

Basics

Inhalt

- Einstufung von Algorithmen
  - O-Notation
- Datenstrukturen
  - Array
  - List
  - Stack
  - Queue
  - Set
  - Map

Motivation

- Wenige grundlegende Algorithmen
  - z.B. Suchen, Sortieren
- Welcher wird wann eingesetzt?
- Was kostet mich ein Algorithmus?
- Bevor „neuer“ Algorithmus erfunden wird, sollte nach einem ähnlichen gesucht werden.

Analyse von Algorithmen

- Laufzeit eines Algorithmus proportional zur Anzahl der Eingangselemente
- Abstraktion unabhängig von CPU-Leistung und Optimierungsvermögen verschiedener Compiler
- Worst Case-Annahmen

Big-O Notation

- Von engl. *Order*
  - Komplexität des Algorithmus wird in Zusammenhang mit der Anzahl der Elemente gebracht

O(1)	konstant	Arrayzugriff
O(log n)	logarithmisch_2	binäre Suche
O(n)	linear	Stringvergleich
O(n log n)		Quicksort
$O(n^2)$	quadratisch	einfache Sortierverfahren

Worst Case Analyse

- Big-O Notation beschreibt „schlechtestes" Verhalten des Algorithmus
- Wichtiges Auswahlkriterium ist zu erwartende Anzahl der Elemente
  - Für wenige Elemente kann ein $O(n^2)$ durchaus besser sein als ein O(n log n)
  - In so einem Fall kann auch ein „einfacher" Algorithmus eingesetzt werden

Beispiel

$T(n) = 3n^2 + 10n + 10$

$O(T(n)) = O(3n^2 + 10n + 10) = O(3n^2) = O(n^2)$

- n^2 Anteil wird dominant, wenn n größer wird
  - Bestimme alle Teile, die von Datengröße abhängig sind
  - Reduziere auf den größten Term

Big-O Werte für N

N	lg N	N lg N	N²	N³
10	3	30	100	1000
100	6	600	10000	1000000
1000	9	9000	1000000	1,00E+09
10000	13	130000	1,00E+08	1,00E+12
100000	16	1600000	1,00E+10	1,00E+15
1000000	19	19000000	1,00E+12	1,00E+18
1 min = 60 s		1 m = 2592000 s		
1 h = 3600 s		1 y = 31557600 s		
1 d = 86400 s		10 y = 3,15E+08 s		
1 w = 604800 s				

Performance Messung in C#

- Stopwatch Klasse
  - Misst Echtzeit
  - Garbage Collector, andere Prozesse können Echtzeitverhalten beeinflussen
- Eigene Klasse Timing
  - Misst CPU-Cycles
  - Unabhängig von GC und anderen Prozessen

Array (Vector)

- Felder von Variablen gleichen Typs
- werden durch Name des Felds/Vektors und einen Index angesprochen
- Index läuft von 0 ... n-1
- Array-Operator [ ]

```
int[] iArray;
iArray = new int[10];
iArray[2] = 15;
```

ArrayList

- Array in C# immer fixe Größe
- ArrayList kann dynamisch wachsen

  - Add()
  - Insert()
  - Remove()
  - Clear()
  - IndexOf()
  - Contains()
  - ToArray()
  - Sort()

  - Count, Capacity

Strukturen (Records)

- Ansammlung von Variablen möglicherweise verschiedenen Typs, die unter einem gemeinsamen Namen angesprochen werden.
- einzelne Elemente werden mit ihrem Namen angesprochen.
- Zugriffsoperator: .

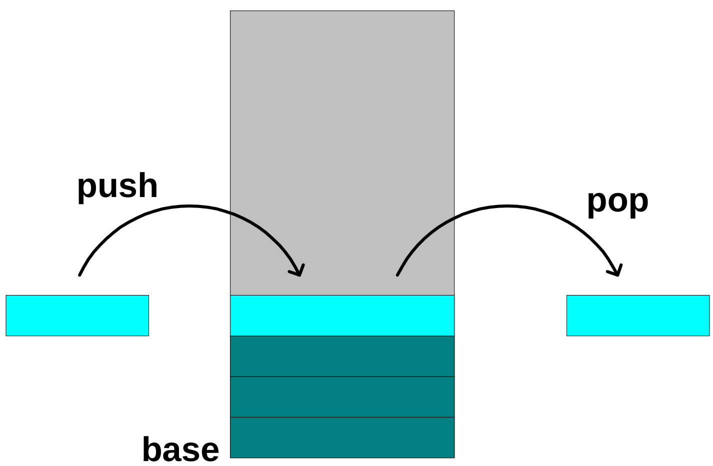
- Benutzerdefiniterte Datetypen

Stack

- Daten werden nach dem Last In First Out Prinzip behandelt (LIFO)
- Reihenfolge wird umgedreht

- Zugriffsfunktionen
  - Push (Daten ablegen) O(1)
  - Pop (Daten holen) O(1)
  - Peek (Schaue oberstes Element an, lasse es aber liegen)

Stack

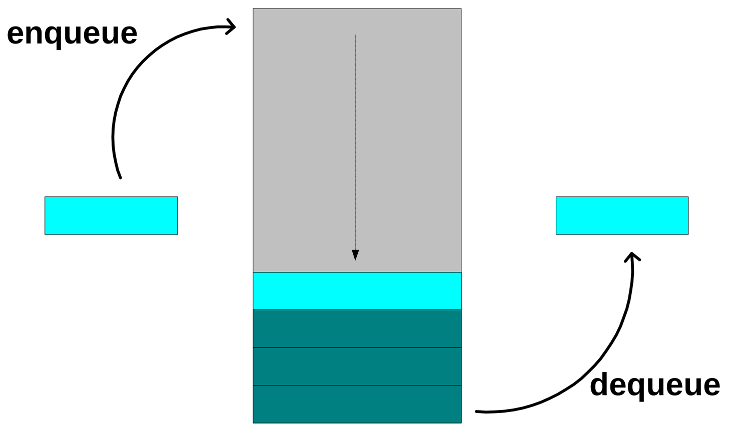


Queue

- Daten werden nach dem First In First Out Prinzip behandelt (FIFO)
- Reihenfolge bleibt erhalten

- Zugriffsfunktionen
  - enqueue (Daten ablegen) O(1)
  - dequeue (Daten holen) O(1)
  - Peek (Schaue erstes Element an, lasse es aber liegen)

Queue



Deque (Double Ended Queue)

- Kombination aus Stack und Queue
- Neue Elemente können an beiden Enden hinzugefügt und herausgenommen werden

- In der Praxis wird zwar an beiden Enden angefügt, aber nur von einem Ende gelesen (output restricted deque)

Set

- Ansammlung von Elementen, welche auf eine bestimmte Art miteinander korrellieren und pro Set nur einmal vorkommen.

- Bei Multiset können gleiche Elemente mehrfach vorkommen

Set

- Operationen im Set
  - Einfügen und Löschen von Elementen O(log(n))
  - Suchen nach einem Element O(log(n))
  - Vergleich von 2 Sets O(m log(n))
- Implementierung z.B. über Baum

Map

- Ansammlung von Key/Value Paaren, welche über den Key miteinander korrellieren und pro Map nur einmal vorkommen.
- Im Gegensatz zum Set ist hier der Schlüssel NICHT Teil des Objekts
- Bei Multimap können Schlüssel mehrfach vorkommen

Collections

- Set
- Map
- Tree

- ... kommen später!

Beispiel 1a, 1b

- Implementiere folgende Klassen:
  - Stack
  - Queue
- Und demostriere ihr Verhalten mit jeweils 10 Elementen.

Rekursion

- Beispiel: Faktorielle von n

$$F(n) = \begin{cases} 1 & n = 0, n = 1 \\ nF(n-1) & n > 1 \end{cases}$$

Rekursion

- Wird durch *terminating condition* beendet
- Zwei Phasen:
  - Winding
    - Aufrufe der gleichen Funktion
    - Stack wächst
    - Übergang zur Phase 2 beim Erreichen der termination condition
  - Unwinding
    - Stack wird wieder kleiner

Rekursion

```
F(4)=4F(3)                           winding phase
      F(3)=3F(2)
            F(2)=2F(1)
                  F(1)=1             term.cond.

            F(2)=2x1                 unwinding phase
      F(3)=3x2
F(4)=4x6
24                                   Recursion complete
```

Rekursion

- Parameter werden bei jedem Aufruf am Stack übergeben
  (= kopiert, call by value)
- Rückgabewerte werden am Stack abgelegt
  (= kopiert)
- Funktionsaufruf kostet Zeit

Tail Rekursion

- Spezielle Form der Rekursion
- Berechnung erfolgt ausschließlich in der *winding phase*
- Zusätzlicher Parameter beim Aufruf für Zwischenergebnis

Tail Rekursion

- Neue Formel für n!
- Starte mit a=1

$$F(n,a) = \begin{cases} a & n = 0, n = 1 \\ F(n-1, na) & n > 1 \end{cases}$$

Tail Rekursion

```
F(4,1)=F(3,4)                              winding phase
       F(3,4)=F(2,12)
              F(2,12)=F(1,24)
                      F(1,24)=24 term.cond.
                                           unwinding phase
                       24                  Recursion complete
```

Rekursion Probleme

- Hoher Ressourcenverbrauch (Stack)
- Fehlende Abbruchbedingung (Endlosschleife)

Rekursion auflösen

- Über Schleifen

- Beispiel: n!
- Starte mit a=1

$$F(n,a) = a * \prod_{i=1}^{n} i$$

Rekursion auflösen

```
int F(int n, int a)
{
  int tmp = a;

  for (int i=1; i<=n; ++i)
    tmp *= i;

  return tmp;
}
```

Beispiel 1c

- Implementiere die rekursive Berechnung der Faktoriellen von n
  - Als Standardrekursion
  - Als Tail Rekursion
- Implementiere die aufgelöste Version ohne Rekursion

- 3 Funktionen
- Untersuche die Grenzen der Funktionen
  - Wie weit kann jede der Funktionen rechnen?