06. April 2025

FDS2 - Übung 5

SS 2025

Tim Peko

Inhaltsverzeichnis

06. April 2025

1. Beispiel 1: Kaninchen mit Stoppuhr

1.1. Lösungsansatz

Die Fibonacci-Funktion wurde in drei Varianten implementiert:

1. **Rekursive Implementierung**: Die klassische rekursive Implementierung, bei der direkt die mathematische Definition

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = 0 \mid F(1) = 1$$

umgesetzt wird.

2. **Entrekursivierte Implementierung mit std::stack**: Hier wird die Rekursion durch die Verwendung eines Stacks simuliert, wobei der Stack den Aufrufparameter n speichert.

3. **Entrekursivierte Implementierung mit eigener intstack-Klasse**: Die entrekursivierte Implementierung mit der selbst implementierten Stack-Klasse anstelle des `std::stack`.

Bei den entrekursifizierten Lösungen werden alle rekursiven Aufrufe in einem Stack gespeichert und bei Abarbeitung die Basecases $\{0, 1\}$ so berücksichtigt, dass sie das Gesamtergebnis entsprechend beeinflussen (wie in der klassischen rekursiven Lösung).

1.2. Testfälle

Die Testfälle vergleichen alle drei Implementierungen für Fibonacci-Zahlen von 0 bis 10 und messen die Laufzeit für größere Werte.

1.2.1. Korrektheit der Implementierungen

Output:

n	Rekursiv	STD Stack	Custom Stack
0	0	0	0
1	1	1	1
2	1	1	1
3	2	2	2
4	3	3	3
5	5	5	5
6	8	8	8
7	13	13	13
8	21	21	21
9	34	34	34
10	55	55	55

Alle drei Implementierungen liefern die gleichen Ergebnisse, die den erwarteten Fibonacci-Zahlen entsprechen.

Ergebnis: success

1.2.2. Laufzeitvergleich

Output:

n	Rekursiv	STD Stack	Custom Stack
10	**0.00000530s**	0.00008450s	0.00003570s
15	**0.00001420s**	0.00135440s	0.00055960s
20	**0.00014080s**	0.01193570s	0.00356280s
25	**0.00176910s**	0.11157950s	0.03796390s
30	**0.01155780s**	1.27071180s	0.42477520s
35	**0.10993520s**	13.57575520s	5.28246510s

Der Laufzeitvergleich zeigt deutlich, dass auch die entrekursifizierte Implementierung exponentiell mit der Größe von n $O(2^n)$ wächst. Das liegt daran, dass die Rekursion nur substituiert wird, es werden keine Berechnungen eingespart.

2. Beispiel 2: Verkehrte Listen

2.1. Lösungsansatz

In diesem Beispiel wurden zwei rekursive Funktionen für einfach verkettete Listen implementiert:

1. **print_list_reverse**: Diese Funktion gibt eine verkettete Liste rückwärts aus, indem sie rekursiv bis zum Ende der Liste navigiert und dann während des Rückwegs die Elemente ausgibt.
2. **reverse_list**: Diese Funktion dreht eine verkettete Liste um, indem sie rekursiv jeden Knoten besucht und die Zeigerrichtung umkehrt.

2.2. Testfälle

2.2.1. Testfall 1: Liste mit mehreren Elementen

Input:

```
list = 1 -> 2 -> 3 -> 4 -> 5
```

Output:

```
list (reversed) = 5 <- 4 <- 3 <- 2 <- 1
reversed_list = 5 -> 4 -> 3 -> 2 -> 1
```

Ergebnis: success

2.2.2. Testfall 2: Liste mit einem Element

Input:

```
list = 42
```

Output:

```
list (reversed) = 42
reversed_list = 42
```

Ergebnis: success

2.2.3. Testfall 3: Leere Liste

Input:

```
list =
```

Output:

```
list (reversed) =
reversed_list =
```

Ergebnis: success

3. Beispiel 3: Labyrinth

3.1. Lösungsansatz

Die Klasse `Maze` implementiert zwei verschiedene Methoden, um zu prüfen, ob ein Weg durch ein Labyrinth vom Start zum Ausgang existiert:

1. **can_escape**: Eine rekursive Implementierung, die Tiefensuche verwendet, um einen Pfad vom Start zum Ausgang zu finden. Sie markiert besuchte Positionen und probiert alle möglichen Richtungen (oben, rechts, unten, links).
2. **can_escape_i**: Eine entrekursivierte Implementierung, die explizit Stacks verwendet, um den Pfad und die zu untersuchenden Richtungen zu verwalten.

Beide Algorithmen erzeugen bei erfolgreicher Suche einen Pfad durch das Labyrinth, der mit ‘.’ gekennzeichnet wird.

3.2. Testfälle

3.2.1. Testfall 1: Rekursive Methode

Input:

```
***************
*     *       *
*** * *    *  *
*   * ******* *
*   *         *
* *********   *
*   *         *
*** *  ****   *
X    *     *  *
***** *****   *
*        *    *
* ********** **
*  S      *   *
*    *        *
***************
```

Output:

```
can_escape = true

Solved Maze (.):
***************
*  ...*       *
***.*.*    *  *
*...*.******* *
*...*........ *
*.********* . *
*...*       . *
***.*  **** . *
X... *     *. *
***** ***** . *
*        *  . *
* **********.**
*  S.....*... *
*    *  ...   *
***************
```

Ergebnis: success

3.2.2. Testfall 2: Entrekursivierte Methode

Input:

```
***************
*     *       *
*** * *    *  *
*   * ******* *
*   *         *
* *********   *
*   *         *
*** *  ****   *
X    *     *  *
***** *****   *
*        *    *
* ********** **
*  S      *   *
*    *        *
***************
```

Output:

```
can_escape_i = true

Solved Maze (.):
***************
*  ...*       *
***.*.*    *  *
*...*.******* *
*...*........ *
*.********* . *
*...*       . *
***.*  **** . *
X... *     *. *
***** ***** . *
*        *  . *
* **********.**
*  S.....*... *
*    *  ...   *
***************
```

Ergebnis: success

3.2.3. Testfall 3: Labyrinth ohne Ausweg

Input:

```
*******
*     *
* *** *
* *S* *
* *** *
*     *
*******
```

Output:

```
can_escape = false
can_escape_i = false
```

Ergebnis: success

4. Beispiel 4: Rekursives Directory-Listing

4.1. Lösungsansatz

Die rekursive Funktion `list_directory_recursive`:

1. Überprüft, ob der angegebene Pfad existiert und ein Verzeichnis ist
2. Iteriert durch alle Einträge im Verzeichnis
3. Gibt für jede Datei den Namen und die Größe aus
4. Ruft sich selbst rekursiv für Unterverzeichnisse auf
5. Verwendet eine Einrückung, um die Hierarchie visuell darzustellen

Die Fehlerbehandlung erfolgt durch try-catch-Blöcke, um mit möglichen Dateisystemfehlern umzugehen.

4.2. Testfälle

4.2.1. Testfall 1: Aktuelles Verzeichnis

Input:

```
<aktuelles Verzeichnis>
```

Output:

```
current_path = <...>/Semester-2_Exercise-05/solution/example_04
|-example_04.cpp (3.75 KB)
|-example_04.vcxproj (6.61 KB)
|-example_04.vcxproj.filters (1.10 KB)
|-example_04.vcxproj.user (168 Bytes)
|-pfc-mini.hpp (20.60 KB)
|-x64
  |-Debug
    |-example_04.exe.recipe (348 Bytes)
    |-example_04.ilk (2.86 MB)
    |-example_04.log (152 Bytes)
    |-example_04.obj (961.69 KB)
    |-example_04.tlog
      |-CL.command.1.tlog (972 Bytes)
      |-Cl.items.tlog (225 Bytes)
      |-CL.read.1.tlog (29.60 KB)
      |-CL.write.1.tlog (908 Bytes)
      |-example_04.lastbuildstate (210 Bytes)
      |-link.command.1.tlog (1.62 KB)
      |-link.read.1.tlog (3.58 KB)
      |-link.secondary.1.tlog (237 Bytes)
      |-link.write.1.tlog (668 Bytes)
    |-vc143.idb (235.00 KB)
    |-vc143.pdb (988.00 KB)
```

Ergebnis: success

4.2.2. Testfall 2: Nicht existierendes Verzeichnis

Input:

```
nicht_existierender_ordner
```

Output:

```
Fehler: Der angegebene Pfad existiert nicht.
```

Ergebnis: success

4.2.3. Testfall 3: Datei statt Verzeichnis

Input:

```
<eigene Programmdatei>
```

Output:

```
self_path = <...>/Semester-2_Exercise-05/solution/example_04/example_04.cpp
Fehler: Der angegebene Pfad ist kein Verzeichnis.
```

Ergebnis: success

Aufwand in h: 9