Übung 5

Abgabe: 22. Mai 2023, 18:00 Uhr

Aufgabe 5.1: Make-a-Meal [10 Punkte, Link zur Abgabe] In dieser Aufgabe geht es darum, ein 3-Gänge-Menü mit Hilfe eines Makefiles zu organiseren.

Für das Menü müssen drei Dinge getan werden: Es sollen eine Vorspeise, ein Hauptgericht und ein Dessert serviert werden.

- Als Vorspeise gibt es einen Salat mit Kräuterdressing. Der Salat muss geputzt, die Kräuter gehackt und das Dressing hergestellt werden.
- Der Hauptgang besteht aus gebratenem Fleisch und Kartoffeln. Das Fleisch muss geschnitten und angebraten werden; die Kartoffeln müssen geschält, geschnitten und anschließend gekocht werden. Dazu ist es nötig, zunächst Wasser aufzusetzen.
- Als Dessert gibt es Eis mit roter Grütze und Schokoladensplittern. Das Eis muss portioniert, mit Grütze übergossen und mit der Schokolade garniert werden.

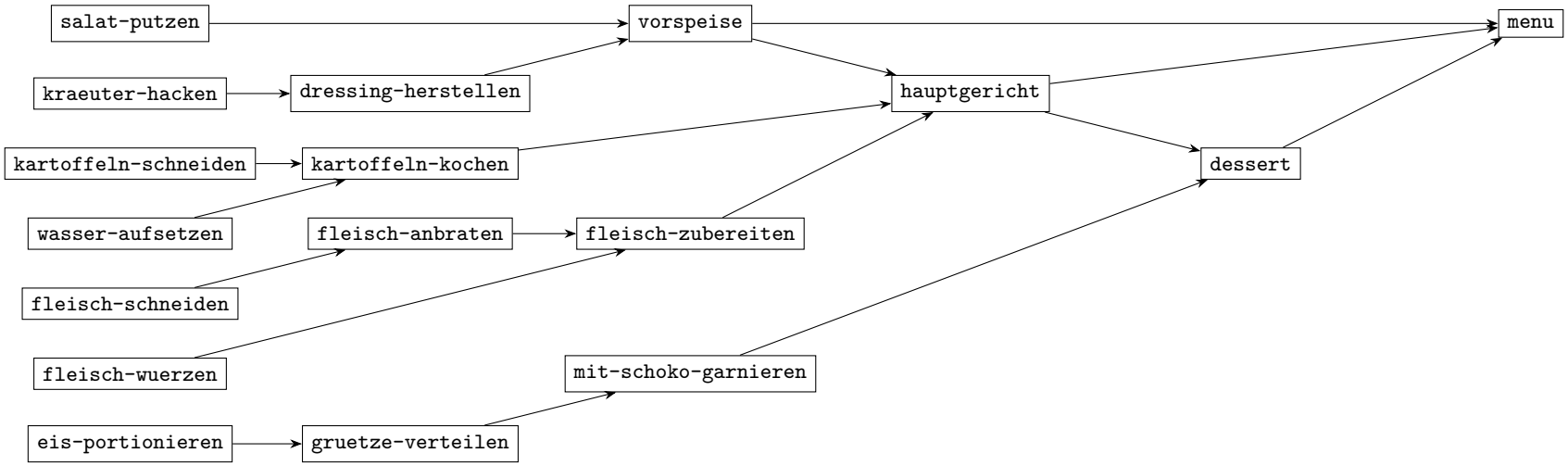
Der Teufel steckt aber im Detail: Zum Beispiel kann das Dressing erst hergestellt werden, wenn die Kräuter gehackt wurden... Auf der nächsten Seite sehen Sie daher eine genaue Darstellung der Abhängigkeiten in einem Abhängigkeitsgraphen; hier zeigt eine Kante von Schritt A nach Schritt B, wenn A abgeschlossen werden muss, bevor B durchgeführt werden kann.

Im Github-Repository finden Sie ein Makefile, in dem bereits alle Schritte definiert sind (für jeden Schritt gibt das Makefile auf der Konsole aus, was getan werden muss). Leider fehlen im gesamten Makefile die Abhängigkeiten.

Ihre Aufgabe: Übertragen Sie die Abhängigkeiten aus dem Abhängigkeitsgraphen in das Makefile im Repository.

Ändern Sie nur das Makefile und keine anderen Dateien! Für diese Aufgabe soll kein C-Code geschrieben werden.

Abhängigkeitsgraph zu Aufgabe 1



Aufgabe 5.2: Dateioperationen [10 Punkte, Link zur Abgabe] In dieser Aufgabe lernen Sie, Dateien zu lesen und zu schreiben. Außerdem lernen Sie, wie Sie Ihrem C-Programm Kommandozeilenparameter übergeben können.

Im Gegensatz zu den bisherigen Aufgaben ist die Eingabe hier nicht in `stdin` gegeben, sondern in einer tatsächlichen Textdatei. Der Name der Textdatei wird Ihrem Programm beim Aufruf als Kommandozeilenparameter übergeben:

Listing 1: Aufruf mit Kommandozeilenparameter

```
1 > ./sum testcases/numbers1.in
```

Um den Namen der Eingabedatei im Programm abfragen zu können, müssen Sie die Signatur der `main`-Methode zunächst so ändern, dass Sie zwei Variablen entgegennimmt:

Listing 1: Main-Methode mit Kommandozeilenparametern

```
1 int main(int argn, const char* args[]) {...}
```

Hier ist `args` ein String-Array. In `args[0]` steht der Name Ihres Programms, in `args[1]` der erste Kommandozeilenparameter, in `args[2]` der zweite usw. Die Variable `argn` sagt Ihnen, wie lang das Array `args` ist (da `args[0]` durch den Programmnamen belegt ist, ist `argn` gerade $i + 1$, wenn i Kommandozeilenparameter übergeben werden).

In Ihrem Programm steht also der Name der Eingabedatei als null-terminierter String in `args[1]`.

Eingabe für Ihr Programm In der ersten Zeile der Eingabedatei stehen zwei ganze Zahlen n und m. Es folgen n Zeilen mit jeweils m ganzen Zahlen (durch Leerzeichen getrennt), so dass dieser Teil der Eingabe aus n Zeilen und m Spalten besteht.

Listing 2: Beispieleingabe

```
1 > cat example.in
2 2 4
3 1 2 3 4
4 5 6 7 8
```

Um die Eingabedatei zu lesen, müssen Sie sie zunächst mit der Funktion `fopen(filename, mode)` aus `stdio.h` öffnen. Hier ist `filename` der Name der Datei, die Sie öffnen möchten und `mode` der Bearbeitungsmodus: `mode="r"` öffnet die Datei zum Lesen, `mode="w"` öffnet sie zum Schreiben. Die Funktion gibt einen Pointer vom Typ `FILE*` zurück. Nun können Sie mit `fscanf` aus der Datei lesen. Die Funktion `fscanf(FILE* f, const char* format, ...)` funktioniert genauso wie `scanf(const char* format, ...)`, bekommt aber zusätzlich als ersten Parameter einen Pointer auf eine geöffnete Datei. Wenn Sie fertig mit lesen sind, schließen Sie die Datei mit `fclose(FILE* f)` wieder.

Ausgabe Ihres Programmes Ihre Aufgabe ist es, die Spaltensummen zu berechnen:

- Geben Sie nach jeder gelesenen Zeile die aktuelle Spaltensumme auf die Konsole aus.
- Nachdem Sie alle Zeilen gelesen haben, schreiben Sie die abschließende Spaltensumme *zusätzlich* in eine Datei `numbers.out`.

Geben Sie die Spaltensummen in beiden Fällen durch Leerzeichen getrennt aus.

Listing 3: Beispielausführung und Ausgabe

```
1 > ./sum example.in
2 1 2 3 4
3 6 8 10 12
4 > cat numbers.out
5 6 8 10 12
6 >
```

Hinweis: Das Schreiben in eine Datei funktioniert analog zum Lesen aus einer Datei; verwenden Sie hier `fprintf(FILE* f, const char* format, ...)` anstelle von `printf` und öffnen Sie die Ausgabedatei `numbers.out` im Schreibmodus. Denken Sie auch hier daran, die Datei wieder zu schließen!

Hinweis II: Die Headerdatei `stdio.h` definiert drei Dateipointer `FILE* stdin`, `FILE* stdout` und `FILE* stderr`, die als Parameter für `fscanf` und `fprintf` verwendet werden können.

Fehlerbehandlung Sollte ein Tabelleneintrag nicht als Zahl interpretierbar sein, geben Sie mit Hilfe der vorgegebenen Funktion `print_error` auf der Standardfehlerausgabe aus, in welcher Zeile und Spalte sich der fehlerhafte Eintrag befindet (Zeile 1 ist diejenige, die n und m enthält; wir indizieren die Spalten ab 1).

Listing 4: Fehlerhafte Eingabedatei

```
1 > cat example.in
2 2 4
3 1 2 3 4
4 5 X 7 8
5 > ./sum example.in
6 1 2 3 4
7 Fehler: Konnte Zeile 3 nicht lesen: Fehler in Spalte 2.
8 >
```

Sie müssen keine Fehlerbehandlung für den Fall schreiben, dass die Eingabedatei nicht die korrekte Anzahl an Zeilen und Spalten enthält und dürfen davon ausgehen, dass die erste Zeile `n m` immer zwei natürliche Zahlen enthält.

Aufgabe 5.3: Union-Find [10 Punkte, Link zur Abgabe] Die Datenstruktur *Union-Find* verwaltet eine Partitionierung einer Menge $M = \{0, \ldots, n-1\}$ von n Schlüsseln in disjunkte Teilmengen $M_0, \ldots, M_k$ mit der Eigenschaft, dass $M_0 \cup \cdots \cup M_k = M$.

Initial ist die Menge M so partitioniert, dass jeder Schlüssel $i \in M$ in einer eigenen Teilmenge liegt, d.h. es ist $M = M_0 \cup \cdots \cup M_{n-1}$ mit $M_i = \{i\}$. Jede Teilmenge hat einen „Namen“, den sog. *Repräsentanten* der Menge. Hier ist das einfach ein beliebiges, festes Element der Teilmenge. Die Datenstruktur unterstützt zwei Operationen:

- `union(x,y)` vereinigt die Teilmenge, die x enthält mit der Teilmenge, die y enthält.
- `find(x)` Sagt uns, in welcher Teilmenge x liegt; d.h., liefert den Repräsentanten der Teilmenge zurück, die x enthält.

Array-basiertes Union-Find implementiert `find(x)` in Worst-Case-Zeit $O(1)$, aber `union(x)` in Worst-Case-Zeit $\Theta(n)$. Allerdings können wir garantieren, dass $n-1$ `union` und f `find`-Operationen in Zeit $\Theta(n \log n + f)$ ausgeführt werden können. *Baum*-basiertes Union-Find implementiert sowohl `find(x)` als auch `union(x,y)` in Worst-Case-Laufzeit $\Theta(\log n)$, kann aber mittels *Pfadverkürzung* fast konstante Laufzeit für beide Operationen erreichen.

Wir implementieren hier Baum-basiertes Union-Find (s. AlDat-Skript, Kapitel 7.4).

Sie finden im Repository ein Grundgerüst für die Implementierung. Ihre Aufgabe ist es, in der Datei `unionfind.c` die Funktionen `uf_find` und `uf_union` zu implementieren. Verwenden Sie dabei die Strategie *Vereinigung nach Größe*. Die Implementierung unterscheidet sich leicht vom Aldat-Skript:

- die Methoden `union` und `find` erwarten als Parameter einen `UFNode*` statt eines Schlüssels.
- Die `UFNode`s werden in einem Array in der `main`-Methode gespeichert.
- Ein Knoten x ist eine Wurzel, wenn sein Elter-Zeiger auf sich selbst, d.h. auf x, zeigt.

Um die Testfälle zu bestehen und Punkte zu erhalten ist es wichtig, dass Sie die folgende Regel für die Vereinigung `union(x,y)` der Partition M_x von x und M_y von y einhalten:

1. Falls $|M_x| > |M_y|$, wird M_y in M_x eingehangen und x wird Repräsentant von $M_x \cup M_y$.
2. Falls $|M_x| < |M_y|$, wird M_x in M_y eingehangen und y wird Repräsentant von $M_x \cup M_y$.
3. Falls $|M_x| = |M_y|$, entscheidet der Schlüssel der Partitionen: Es wird die Partition mit dem *kleineren* Schlüssel in die Partition mit dem größeren Schlüssel eingehangen. Der Schlüssel der Partition mit dem größeren Schlüssel wird Repräsentant von $M_x \cup M_y$.

Wenn Sie diese Regeln nicht beachten, erkennt das Testskript Ihre Lösung nicht!

Implementieren Sie bitte *keine Pfadverkürzung* – auch das erkennt das Testskript nicht – und verändern Sie nur die Datei `unionfind.c`!

Zum Lösen der Aufgabe genügt es, die beiden geforderten Operationen zu implementieren, da der restliche Code bereits vorgegeben ist. Ihr Code wird mit Testfällen im folgenden Format getestet:

Listing 5: Eingabeformat

```
> cat example.in
5 4
u 1 2
f 2
u 2 3
f 3
```

Die erste Zeile enthält zwei natürliche Zahlen n und m, wobei n die Anzahl der Schlüssel und m die Anzahl der durchzuführenden Operationen ist. Es folgen m Zeilen, die jeweils eine Operation enthalten: Für Zeilen im Format `u <x> <y>` wird `union(x,y)` ausgeführt und der Repräsentant von $x \cup y$ ausgegeben. Für Zeilen im Format `f <x>` wird `find(x)` ausgeführt und der Repräsentant von `x` ausgegeben. Nach jeder Operation gibt der vorgebene Code einmal die gesamte Datenstruktur aus.

Listing 6: Ausgabeformat

```
> ./unionfind < example.in
 / 0: 0 / 1: 1 / 2: 2 / 3: 3 / 4: 4 /
union 1, 2: 2
 / 0: 0 / 1: 2 / 2: 2 / 3: 3 / 4: 4 /
find 2: 2
 / 0: 0 / 1: 2 / 2: 2 / 3: 3 / 4: 4 /
union 2, 3: 2
 / 0: 0 / 1: 2 / 2: 2 / 3: 2 / 4: 4 /
find 3: 2
 / 0: 0 / 1: 2 / 2: 2 / 3: 2 / 4: 4 /
```