EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Informatik 2
Sommersemester 2021
Prof. Dr. T. Grust, B. Dietrich, C. Duta, D. Hirn, T. Fischer

Übungsblatt 5 (02.06.2021)

Abgabe bis: Mittwoch, 09.06.2021, 14:00 Uhr

Relevante Videos bis einschließlich:
`Informatik 2 - Chapter 06 - Video #36`
https://tinyurl.com/Informatik2-SS2021

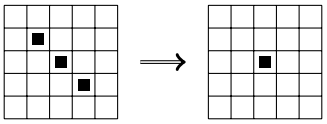
Wichtig: Ihr werdet auf diesem Blatt mit einer etwas abgeändertern `curses`-Library arbeiten. Dazu müsst ihr wieder ein Docker-Update auf euren Rechnern einspielen. Folgt dazu den Anweisungen hier: https://forum-db.informatik.uni-tuebingen.de/t/es-gibt-ein-c0-update-was-tun/9532.

Aufgabe 1: Conway's Game of Life [30 Punkte] (*Abgabe:* `gameoflife.c0`)

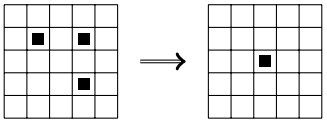
Conway's Game of Life[1] ist ein Spiel ohne Spieler, das die Evolution einer Population von Zellen auf einem zweidimensionalen Gitter simuliert. Der Ablauf der Simulation wird vollständig von der initialen Zellgeneration bestimmt. Jede Zelle des Gitters kann dabei entweder *lebendig* (■) oder *tot* (□) sein.

Zuerst wird das Gitter mit einer Belegung von Zellen (■ oder □) initialisiert. Basierend auf diesem Initialzustand (Generation 1) wird die nächste Zellgeneration (Generation 2) bestimmt. Die acht direkten Nachbarn jeder Zelle (horizontal, vertikal und diagonal) der *aktuellen* Generation werden inspiziert, um mit den folgenden Regeln zu entscheiden, ob die Zelle in der *nächsten* Generation ■ oder □ ist:

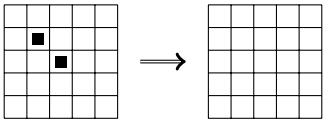
- **Survival:** Ist die Zelle ■ und hat **zwei** oder **drei** benachbarte ■ Zellen, dann ist diese Zelle in der Folgegeneration auch weiterhin ■:



- **Birth:** Ist die Zelle □ und hat **genau drei** benachbarte ■ Zellen, dann wird diese Zelle in der Folgegeneration ■:



- **Death:** Alle anderen Zellen werden bzw. bleiben □:



Dieser Prozess wird solange wiederholt, bis die Simulation von uns unterbrochen wird. Die Regeln sind einfach, können aber zu dynamischen und oszillierenden Abläufen führen, etwa für den in Abbildung 1 gezeigten *Glider*. Es gibt eine schier endlose Sammlung an interessanten Initialzuständen[2], darunter beispielsweise die *Gun*[3] , die während der Simulation endlos *Glider* produziert.

[1] https://playgameoflife.com/
[2] https://www.conwaylife.com/wiki/Category:Patterns
[3] https://www.conwaylife.com/wiki/Gun

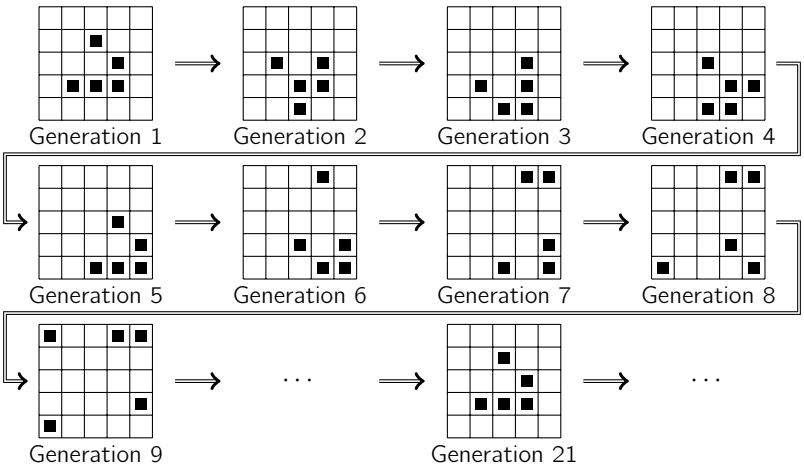


Abbildung 1: Ausschnitte eines möglichen Ablaufs des *Game of Life* in einem 5×5-Gitter mit Initialzustand, der lebendige (■) und tote (□) Zellen in Generation 1 definiert. Beachtet hier, wie Zellen auch über den Rand hinaus auf der gegenüberliegenden Gitterseite benachbart sind. Das *Game of Life* mit diesem Initialzustand kommt niemals zur Ruhe.

Eure Aufgabe ist es nun, Conway's *Game of Life* zu implementieren. Geht dazu folgendermaßen vor:

1. Schreibe eine Funktion `bool[][] state_from_cells(string[] cells, int width, int height)`, die eine zeilenweise String-Repräsentation `cells` des Initialzustands erhält und diese in ein zweidimensionales `bool`-Array umwandelt. Stelle dabei durch Pre- und Postconditions sicher, dass sowohl (1) das Argument `cells` und (2) das konstruierte zweidimensionale Array genau die übergebene Breite (`width`) und Höhe (`height`) besitzen. Es gilt die offensichtliche Zuordnung: Ein `'#'` in `cells` entspricht einer lebendigen Zelle ■, die im zweidimensionalen Array mit `true` repräsentiert wird (`' '`, □, `false` entsprechend). Position (0,0) liegt im Gitter links oben.

   Dieses Beispiel für das Argument `cells` definiert die Generation 1 aus Abbildung 1:

   ```
   cells[0] = "     ";
   cells[1] = "  #  ";
   cells[2] = "   # ";
   cells[3] = " ### ";
   cells[4] = "     ";
   ```

   Funktionen aus der C0-Bilbiothek `string` sind hier hilfreich.

2. Schreibe eine Funktion `int count_neighbors(int x, int y, bool[][] state, int width, int height)`, die für die Zelle an Position `(x,y)` die Anzahl der lebendigen ■ direkten Nachbarn (horizontal, vertikal und diagonal) zählt. Definiere eine Postcondition, die versichert, dass diese Anzahl zwischen 0 und 8 liegt.

   Wichtig: Im Gitter des *Game of Life* respektiert Nachbarschaft *wrap around* (effektiv gibt es gibt also keinen Rand). Für das 5×5-Gitter aus Abbildung 1 gilt beispielsweise: die acht Nachbarn der Zelle an Position `(0,0)` sind: `(4,4)`, `(0,4)`, `(1,4)`, `(4,0)`, `(1,0)`, `(4,1)`, `(0,1)` und `(1,1)`.

3. Schreibe eine Funktion `bool next(bool cell, int neighbors)`, die eine Zelle `cell` und die Anzahl ihrer lebendigen Nachbarn erhält und mittels der drei Regeln des *Game of Life* ableitet, ob diese Zelle in der kommenden Generation ■ oder □ sein wird.

4. Schreibe eine Funktion `bool[][] game_of_life(bool[][] state, int width, int height)`, die einen Gitterzustand `state` erhält und mittels der oben implementierten Funktionen den Zustand der nächsten Generation berechnet.

5. Schreibe eine Funktion `void run(bool[][] state, int width, int height)`, die mittels der zuvor implementierten Funktion `game_of_life` das *Game of Life* solange simuliert und Gitterzustände auf dem Terminal ausgibt, bis die `Enter`-Taste gedrückt wird. Nutzt dazu die `#use <curses>`-Library. Die Dokumentation dazu findet ihr in der aktuallisierten Referenz **C0 Libraries (Addendum)** im Forum[4]. In dieser Dokumentation findet ihr auch ein Beispielprogramm, das den Einsatz der relevanten Funktion der `curses`-Library zeigt.

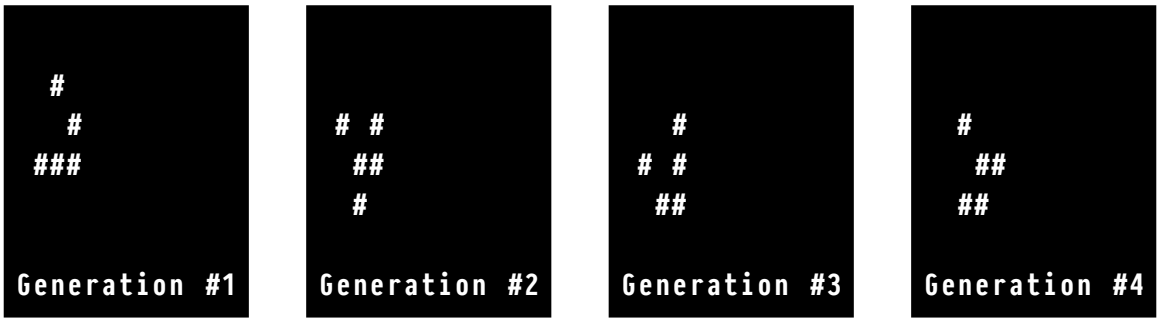


Abbildung 2: Beispielausgabe auf dem Terminal. Die Generationen schreiten mit jedem Tastendruck voran. Abbruch der Simulation, wenn die `Enter`-Taste gedrückt wurde.

   Geht dazu folgendermaßen vor:

   i. Initialisiert zuerst die `curses`-Schnittstelle `w` vom Typ `window_t` mit einem Aufruf der `curses`-Function `window_t c_initscr()`.
   ii. Deaktiviert dann den Cursor mit der `curses`-Function `c_curs_set(0)`. Wenn ihr später den Cursor wieder aktivieren möchtet, ruft ihr einfach `c_curs_set(1)` auf.
   iii. Die Ausgabe der Zellen der aktuellen Generation sowie der Generationsnummer könnt ihr mittels der folgenden `curses`-Funktionen implementieren:
   `void c_wclear(window_t w)` löscht alle Zeichen auf dem Terminal. `void c_wmove(window_t w, int y, int x)` bewegt den Cursor an die angegebene Position auf dem Terminal. Beachtet dabei die Reihenfolge, in welcher die Koordinaten (`x`,`y`) übergeben werden. `void c_waddch(window_t w, int d)` und `void c_waddstr(window_t w, string s)` geben jeweils das Zeichen mit ASCII-Code `d` bzw. den String `s` an der aktuellen Cursorposition aus. Damit diese Zeichen auch auf dem Terminal erscheinen, muss noch `void c_wrefresh(window_t w)` aufgerufen werden. Orientiert euch für das Format der Ausgabe an der Beispielausgabe in Abbildung 2.
   iv. Nach Ausgabe einer Generation wird auf einen Tastendruck gewartet. Bei Eingabe von `Enter` wird die Simulation beendet, ansonsten wird die kommende Generation berechnet (Funktion `game_of_life`) und dann ausgegeben.
   Nutzt dazu die `curses`-Funktion `int c_getch()`, die auf eine Eingabe über die Tastatur wartet und den ASCII-Code des eingegebenen Zeichens als `int` codiert zurückgibt. Um den Abbruch zu implementieren, benötigt ihr die `curses`-Funktion `bool cc_key_is_enter(int d)`, die genau dann `true` liefert, wenn der ASCII-Code `d` die `Enter`-Taste darstellt.

Nutzt keine C0-Bibliotheken ausser `string`, `curses` und `util`.

[4] https://forum-db.informatik.uni-tuebingen.de/t/c0-language-reference-c0-library-guides/9571

Aufgabe 2: Insertion Sort [10 Punkte] (*Abgabe:* `insertionsort.c0, printarray.c0`)

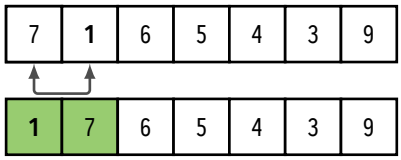
Insertion Sort ist ein Sortieralgorithmus, der mit seinem Laufzeitverhalten von $O(n^2)$ den in der Vorlesung behandelten *Mergesort*- und *Quicksort*-Verfahren unterlegen ist. *Insertion Sort* ist konzeptuell aber sehr einfach.

Wie *Mergesort* arbeitet auch *Insertion Sort in-place* (oder: *in situ*) — es ist also nicht nötig, ein neues Array für das resultierende sortierte Array zu allozieren. Vielmehr werden die Elemente innerhalb des Arrays selbst umsortiert. Und auch für *Insertion Sort* gilt: Arrays der Länge 0 oder 1 Element sind per Definition bereits sortiert.

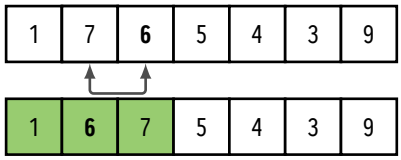
Sei `xs` ein Array vom Typ `int[]` mit mindestens zwei Elementen. Der Algorithmus betrachtet sukzessive alle Elemente von `xs` von *links nach rechts*. Jedes Element `x` wird dabei so lange nach links verschoben, bis sich direkt links von `x` ein Element befindet, dass kleiner oder gleich groß ist. Dabei kann (und soll) im Laufe der Durchführung zugesichert werden, dass alle Elemente links vom aktuell zu sortierenden Eintrag im Array bereits sortiert sind.

Wir beschreiben am folgenden Beispiel `[7,1,6,5,4,3,9]` den Algorithmus nochmal genauer:

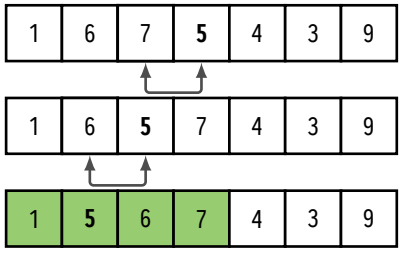
- *Insertion Sort* beginnt mit dem ersten zu betrachtenden Element an Position 1 (hier: **1**). Links davon steht nur das Element 7. Da 7 > **1**, wird **1** einmal nach links verschoben. Wir markieren hierbei die Positionen des Arrays, die bereits sortiert sind, mit ■:



- Position 2 (**6**) wird als nächstes betrachtet. Die beiden Elemente links davon sind sortiert. Verschiebe also **6** solange nach links, bis das Element links davon kleiner ist. In diesem Fall wird das Element genau einmal nach links geschoben, denn an Position 0 steht 1 und 1 ⩽ **6**:



- Der nächste Schritt startet mit dem Element an Position 3 (**5**). Dieses mal muss das Element zweimal nach links geschoben werden:



- Wir setzen den Algorithmus für Position 4 (**4**), Position 5 (**3**) und Position 6 (**9**) fort. Danach ist das gesamte Array sortiert:

Implementiert die Funktion `void insertionsort(int[] xs, int len)`, die nach dem oben erklärten Algorithmus *Insertion Sort* ein Array `xs` der Länge `len` sortiert. Achtet dabei darauf, **Loop-Invarianten und Postconditions** zu formulieren, die unsere Beobachtungen über die (teilweise) Sortierung des Arrays festhalten. Verwendet dafür die aus der Vorlesung bekannte Funktion `is_sorted`, die ihr einfach aus dem Source-File `mergesort.c0` in eure Abgabe kopieren könnt.

Ausser den Libraries `conio`, `rand` und `util` sind keine C0-Bibliotheken zu verwenden. Nutzt das bekannte `printarray.c0`, um in eurer Funktion `main` Tests durchzuführen.

Wichtig: Das File `printarray.c0` gehört mit in euer ZIP-File.