Deadline: 25.01.2026 23:59:59
https://moodle.hpi.de/course/view.php?id=1046

Abgabe

Die Übungen werden in GitHub Classroom bearbeitet und via Moodle abgegeben. Insgesamt sind **maximal 25 Punkte** erreichbar. Bei der Abgabe via Moodle ist es wichtig, dass ihr **den Link zu eurem GitHub Repository** (aus dem URL/Suchfeld) in das Textfeld (Onlinetext) bei der Abgabe kopiert, damit wir eure Abgabe bewerten können. Der Link hat folgendes Format:

```
https://github.com/hpi-aiis-pt1-fall25/<übungsname>-<username>
```

Wichtig: die Bewertung erfolgt in einem *Testatgespräch*, d.h. ihr müsst eure Lösungen in einem Termin mit einem Tutor oder einer Tutorin besprechen. Auschlaggebend für die Bewertung ist, dass ihr eure Lösungen erklären und Fragen dazu beantworten könnt! Wir werden den Stand des letzten Commit bewerten, der vor der jeweiligen Deadline abgegeben wurde.

ACHTUNG: Für diese Übung habt ihr bis zum 25.01.2026 23:59:59 Zeit. Ihr habt für diese Übung also **eine Woche** Zeit.

2er-Gruppen: In dieser Übung könnt ihr in 2er-Gruppen (den Gruppen eurer Testatgespräche) arbeiten. In diesem Fall kann die Abgabe in einem eurer Repositories erfolgen. Wie immer muss jeder von euch in der Lage sein, die Lösung zu erklären und Fragen dazu zu beantworten.

ACHTUNG: Diese Übung ist die letzte verpflichtende Übung. Es wird allerdings noch eine weitere freiwillige Übung geben, die euch bei der Klausurvorbereitung helfen soll. Die Bearbeitung der freiwilligen Übung ist nicht verpflichtend, aber sehr empfohlen.

ACHTUNG: Am 04.02.2026 findet um 13:30 in HS1 statt des "zentralen Hilfeslots" ein *Workshop zur Klausurvorbereitung* statt. Dort werden wir die wichtigsten Themen der Vorlesung und Übungen wiederholen und auf die Klausur vorbereiten. Die Teilnahme ist freiwillig, aber sehr empfohlen!

Is that a unicorn?!

Kunibert ist der Meinung, dass seine Geschäftsidee aus der letzten Übung auch ein Startup sein könnte und einen Namen hat er auch schon: `arraylist.io`. Natürlich sieht Kunibert sich selbst als CEO an, aber ihr wurdet zum CCO (Chief C Officer) ernannt. In dieser Übung werden wir den Code aus der letzten Übung weiterentwickeln, effizienter machen und einige gute Softwareentwicklungspraktiken implementieren.

Aufgabe 1: MVP - 6 Punkte

In `previous_code.c` findet ihr eine potentielle Lösung für die Letzte Übung. Da wir in dieser Übung aber einige Erweiterungen hinzufügen werden, wollt ihr zuerst einige gute Softwareentwicklungspraktiken implementieren.
In `mvp.c` soll nur die `main` Funktion implementiert sein, alle anderen Funktionen sollen *inkludiert* werden. Hier kann der Foliensatz *Softwareentwicklung 2* hilfreich sein. Konkret sollen:

1. Die Funktionen `make_list` und `list_get` in die `arraylistio_core.c` Datei ausgelagert werden. In der zugehörigen `arraylistio_core.h` soll außerdem das `struct list_t` ausgelagert sein. **Dieser Punkt ist beispielhaft bereits implementiert!** Ihr müsst den Code der beiden Dateien nicht ändern.
2. Die Funktionen `list_resize`, `list_append`, `list_remove` und `list_free` sollen nach `arraylistio_basic.c` ausgelagert werden. Die zugehörige `arraylistio_basic.h` müsst ihr ebenfalls implementieren. Achtet hier darauf auch einen sogenannten *Header-Guard* zu implementieren (wie im Beispiel in `arraylistio_core.h`).
3. Inkludiert `arraylistio_core.h` und `arraylistio_basic.h` korrekt in der `mvp.c`.

Zum Kompilieren müsst ihr nun alle genutzten `.c` Dateien auflisten. Als Beispiel:

```
clang -o mvp.out mvp.c arraylistio_core.c arraylistio_basic.c
```

Stattdessen verwendet man in der Regel eine *Makefile*. Im Template ist bereits eine Makefile mit dem Befehl `make mvp` verwendbar (für gcc-User: `make CC=gcc mvp`).

Abgabe: `arraylistio_basic.c`, `arraylistio_basic.h`, `mvp.c`

Aufgabe 2: Release - 14 Punkte

Euer Startup `arraylist.io` skaliert und der Release rückt näher! Die Download-Zahlen steigen und die bisherige Implementierung der ArrayList ist zu langsam. Kunibert kommt in blauer Jeanshose und einem schwarzen Rollkragenpullover in euer Büro gestürmt und verlangt eine effizientere Implementierung. Nachdem ihr einige Stunden in eurem CCO-Büro im 16. Stock gegrübelt habt, kommt ihr auf eine Idee. Bisher wird der zugrunde liegende Speicher der ArrayList bei jeder Operation (append / remove) angepasst. Diese `realloc`-Operation könnte viel Zeit kosten.

Stattdessen soll nun folgende Strategie implementiert werden: jedes Mal wenn die Kapazität erreicht wird, wird die Kapazität verdoppelt. Beim Entfernen von Elementen soll auch nur der Speicher mit `realloc` verändert werden, falls nur noch die Hälfte der Kapazität mit Elementen gefüllt ist. Somit wird die Anzahl der `realloc`-Operationen deutlich reduziert. Allerdings wird auch mehr Speicher reserviert als nötig. Ob sich dieser Tradeoff lohnt, sollt ihr nun testen.

Implementiert diese Strategie in `arraylistio_fast.c` und erstellt auch eine Header-Datei. Dabei sollt ihr die notwendigen Funktionen aus `arraylistio_basic.c` durch die neue Variante ersetzen. Die neuen Funktionsnamen sollen durch das Prefix `fast_...` ergänzt werden (also z.B. `fast_list_append`).

Ihr wollt nun testen, ob eure neue Implementierung auch wirklich schneller ist. Dafür ist die `release.c` Datei vorgesehen, die schon vorbereiteten Code zum Messen der Ausführungsgeschwindigkeit beider Varianten enthält. Fügt die notwendigen `include`-Statements hinzu. Außerdem soll in der `Makefile` der Befehl `make release` analog zu `make mvp` hinzugefügt werden. Zusätzlich liegt eine `Makefile.advanced` bei, die ein etwas komplexeres, aber in der Praxis übliches Setup mit Objekt-Dateien, optionalen Compiler-Flags und Run-Targets zeigt. Diese Datei müsst ihr nicht verwenden, könnt es aber gerne tun, wenn ihr möchtet. Ihr könnt sie mit `make -f Makefile.advanced` verwenden (z.B. `make -f Makefile.advanced mvp`).

Testet nun die beiden Implementierungen mit dem Code in `release.c` und beschreibt eure Ergebnisse (inklusive der Messergebnisse) kurz in `speedtest.txt`.

Abgabe: `arraylistio_fast.c`, `arraylistio_fast.h`, `speedtest.txt`, `release.c` (modifizieren), `Makefile` (modifizieren)

Aufgabe 3: Due Diligence - 5 Punkte

Kunibert ist begeistert von eurer neuen Implementierung und möchte sie direkt in die Produktion übernehmen. Bevor ihr das tut, wollt ihr aber einige offene Fragen klären.

1. Wenn wir in `release.c` die `BIGNUMBER` deutlich verringern, z.B. auf 1000, ist der Unterschied zwischen den beiden Implementationen deutlich kleiner. Wieso?
2. Wenn wir in `release.c` in der Schleife zum Entfernen der Elemente

   ```
   for (int i = BIG_NUMBER - 1; i >= BIG_NUMBER / 2; i--)
   ...
   ```

   mit

   ```
   for (int i = BIG_NUMBER / 2; i >= 0; i--)
   ...
   ```

   ersetzen, dauert das Entfernen viel länger. Wieso?
3. Wie viele Megabyte werden in `release.c` maximal reserviert, wenn wir die schnelle Implementierung wählen und wie viele Megabyte, wenn wir die langsame wählen? Es geht nur um den Inhalt der ArrayList in `int *contents`, anders allozierter Speicher kann vernachlässigt werden. Die Lösung soll berechnet werden und der Rechenweg kurz beschrieben werden.

Beantwortet die Fragen in einem Textdokument `due_diligence.txt`.

Abgabe: `due_diligence.txt`

BONUS: Kollision mit der Realität - 2 Punkte

Recherchiert, ob in der Praxis ein dynamisches Array (z.B. in C++) mit eurer schnellen Strategie aus Aufgabe 2 implementiert wird, oder auf eine andere Art und Weise. Beantwortet dies kurz in einer `bonus.txt`. Fügt mindestens einen Link zu einer Quelle, auf die ihr eure Antwort stützt, hinzu.

Deadline: 25.01.2026 23:59:59, **Bearbeitung** in GitHub, **Abgabe** via Moodle.