Stefan Kurtz **Universität Hamburg**
Abteilung für Genominformatik **Zentrum für Bioinformatik**

Programmierung für Naturwissenschaften 2
Sommersemester 2020
Übungen zur Vorlesung: Ausgabe am 23.06.2020

Aufgabe 8.1 (5 Punkte)

In den Materialien zu dieser Aufgabe finden Sie im Verzeichnis `Csources` in den Dateien `multiseq.c` und `multiseq.h` die C-Implementierung einer Klasse `Multiseq` zum Parsen von Dateien im Multi-Fasta Format und zur Verwaltung der darin enthaltenen Informationen. Dieses Format wird für biologische Sequenzen, insbesondere DNA- und Proteinsequenzen, verwendet. Für jede einzelne Sequenz gibt es eine eigene Kopfzeile, die mit dem Zeichen > beginnt. Darauf folgen dann beliebig viele Zeilen mit Zeichen aus dem DNA- oder Aminosäurealphabet. In den Materialien finden Sie mehrere Beispieldateien mit dem Suffix `.fna`. Einige Dateien sind allerdings nicht korrekt formatiert, was im C-Programm auch erkannt und mit einer entsprechenden Fehlermeldung quittiert wird.

Ihre Aufgabe besteht darin, in der Datei `multiseq.cpp` die Methoden der Klasse `Multiseq` in C++ zu implementieren. Dabei sollen Sie den C-Programmcode übertragen und an die Syntax einer C++-Klasse anpassen. Dabei können Sie einen großen Anteil des C-Codes übernehmen und müssen im Wesentlichen die Zugriffe auf den Zeiger `multiseq` durch entsprechende Syntax mit dem Schlüsselwort `this` ersetzen. Die Methodennamen der Klasse finden Sie in `multiseq.hpp`. Außerdem sollen Sie in der Datei `multiseq_test.cpp` ein Testprogramm in C++ analog zu `Csources/multiseq_test.c` entwickeln. Ihre Implementierung soll die folgenden Bedingungen erfüllen:

- Die Fehlerbehandlung soll durch eine Ausnahmebehandlung erfolgen. D.h. es müssen jeweils die Aufrufe der Funktionen `error_msg_empty_sequence` und `error_msg_missing_header` einschließlich der folgenden **`return`** `NULL`-Anweisung (siehe `multiseq.c`) durch Aufrufe der entsprechenden Funktion mit dem Präfix `throw_` ersetzt werden. Diese Funktionen sind in der Datei `multiseq_template.cpp` implementiert. Diese Datei benennen Sie in `multiseq.cpp` um. Im Hauptprogramm müssen Sie eine mögliche Ausnahme, initiiert durch `throw`, mit einem `try`-Block und einem `catch (std::invalid_argument &msg)`-Block behandeln. Im letzteren Block wird auf die Fehlermeldung aus dem String-Objekt `msg` mit `msg.what()` zugegriffen. In den Vorlesungsfolien finden Sie ein Beispiel mit einer `bad_alloc`-Ausnahmebehandlung. Sie müssen in beiden genannten `.cpp`-Dateien eine Anweisung **`#include`** `<stdexcept>` einfügen.
- Die Fehlermeldungen müssen identisch sein mit den in `files2msg` aus `check_err.py` spezifizierten Fehlermeldungen. Das ist gewährleistet, wenn Sie die beiden Funktionen `throw_empty_sequence` und `throw_missing_header` verwenden.
- Die Fehlermeldungen werden über den Fehlerstream `std::cerr` ausgegeben. Die Ausgabe erfolgt ausschließlich in `multiseq_test.cpp`. Daher ist eine Anweisung **`#include`** `<iostream>` notwendig.
- Sie dürfen in Ihrer eigenen Implementierung Speicher nicht mit `malloc`, `calloc` oder `realloc` allokieren.

- Das Einlesen des Dateiinhaltes soll wie im C-Programm mit Hilfe der Klasse `PfNFileInfo` erfolgen. Dafür finden Sie die entsprechenden Dateien in den Materialien. Diese können unverändert benutzt werden, d.h. eine Konvertierung nach C++ ist nicht erforderlich.
- Während im C-Programm die Zeiger auf die Anfänge der Kopfzeilen bzw. der Sequenz jeweils in einem dynamischen Array gespeichert werden, sollen Sie in der C++-Implementierung Instanzen `header_vector` und `sequence_vector` der Klasse `std::vector` verwenden, die bereits in `multiseq.hpp` deklariert sind. Sie brauchen dazu die Methoden `push_back()`, `back()`, `size()` und einen indexbasierten Zugriff.
- Zur Ausgabe von Sequenzen in der Methode `show` können Sie wie bei der C-Implementierung `fwrite` verwenden.

In den Materialien finden Sie ein Makefile zum Kompilieren Ihrer Quelldateien. Durch `make test` verifizieren Sie, dass Ihre Implementierung korrekt funktioniert. Wenn der Test nicht erfolgreich ist, dann liefert das Shell-Skript `multiseq_test.sh` entsprechende Fehlermeldungen. Sie sollten sich den ersten fehlgeschlagenen Test ansehen, den Fehler in Ihrem Programm beseitigen, und dann weiter fortfahren. Bzgl. des Python-Skripts `check_err.py`, das die korrekte Fehlerbehandlung verifiziert, gehen Sie analog vor. Falls Sie nicht unter macOS arbeiten und das Programm `valgrind` verfügbar ist, wird dieses in einem weiteren Test verwendet.

Punkteverteilung:

- 2 Punkte für den Konstruktor `Multiseq`.
- 1 Punkt insgesamt für die anderen Funktionen der Klasse `Multiseq`.
- 1 Punkt für die Implementierung der Funktion `main()` in `multiseq_test.c`.
- 1 Punkt für bestandene Tests.

Zur Bearbeitung dieser Aufgabe ist es hilfreich, die Abschnitte 11, 21, 34, 35, 36, 37 der Vorlesung (siehe Spalte *Nummer* in der Tabelle in `pfn2_vorlesung_2020.html`), zu kennen.

Aufgabe 8.2 (5 Punkte)

In dieser Aufgabe geht es darum, verschiedene Hash-Funktionen für alle Worte $words(T)$ in einem Text T zu bewerten. Alle Studierenden, die den Begriff Hash-Funktion noch nicht kennen, sollten sich dazu die Folien in der Datei `hashfunctions.pdf` ansehen. Sei h eine Hash-Funktion, die für alle Worte w über einem Alphabet einen Hash-Wert $h(w) \in \mathbb{N}_0$ liefert. Sei $H(h,T) = \{h(w) \mid w \in words(T)\}$ die Menge aller Hash-Werte von Worten des Textes T.

Das Kollisionsmaß einer Hash-Funktion wird durch die Anzahl der Kollisionen bestimmt. Eine Kollision tritt auf, wenn verschiedene Worte den gleichen Hash-Wert haben. Sei daher $f(h,T,i)$ die Anzahl der Worte $w \in words(T)$ mit $h(w) = i$. Dann soll

$$\mathsf{hashcoll}(h,T) = \frac{1}{|H(h,T)|} \sum_{i \in H(h,T)} f(h,T,i)^2$$

das Kollisionsmaß von h bzgl. T sein. Falls es keine Kollisionen gibt, dann ist $f(h,T,i) = 1$ für alle $i \in H(h,T)$. Damit gilt $\mathsf{hashcoll}(h,T) = 1$, d.h. die Hash-Funktion h hat bzgl. T das geringste Kollisionsmaß. Je mehr Kollisionen es gibt, umso größer ist $\mathsf{hashcoll}(h,T)$.

Beispiel: Wir betrachten einen Text T mit 15 Worten und eine der implementierten Hash-Funktion $h =$ ELFHash, deren Werte in der folgenden Tabelle angegeben sind:

w	$h(w)$
BUT	18 340
Act	18 340
Add	18 340
Last	338 084
Meet	342 980
Heard	5 159 044
Guard	5 159 044
Herod	5 163 348
Inherit	5 163 348
Sibyl	5 896 700
Sicil	5 896 700
Adding	75 149 383
Acting	75 149 383
Always	75 749 635
penalties	75 749 635

Offensichtlich gibt es 3 Worte mit dem gleichen Hash-Wert 18 340 und 5 Paare von Worten jeweils mit dem gleichen Hash-Wert, wie man leicht in der folgenden Tabelle sieht:

i	$\{w \mid h(w) = i\}$	$f(h, T, i)$
18 340	BUT Act Add	3
338 084	Last	1
342 980	Meet	1
5 159 044	Guard Heard	2
5 163 348	Herod Inherit	2
5 896 700	Sibyl Sicil	2
75 149 383	Adding Acting	2
75 749 635	Always penalties	2

Damit ist $|H(h,t)| = 8$ und

$$\mathsf{hashcoll}(h, T) = \frac{3^2 + 1 + 1 + 5 \cdot 2^2}{8} = \frac{31}{8} = 3.875.$$

Schreiben Sie ein C++-Programm `hashcoll.cpp`, das für alle Hash-Funktionen, die in der Datei `hashfunctions.cpp` implementiert sind, das Kollisionsmaß bzgl. eines gegebenen Textes bestimmt.

Gehen Sie dabei wie folgt vor:

1. Öffnen Sie die Datei, deren Name durch `argv[1]` referenziert wird. Ein entsprechendes Beispiel finden Sie im Abschnitt zur Codon-Translation in der Datei `Cpp_slides.pdf`.
2. Die Adresse der resultierenden Instanz der Klasse `std::ifstream` wird an die Funktion `file2wordset` aus `tokenizer.cpp` übergeben. Diese Funktion liefert die Worte in der Datei als Menge vom Typ `std::set<std::string>` zurück.
3. Die Anzahl der Hash-Funktionen liefert die Funktion `hashfunction_number`. Für alle `i` zwischen 0 und `hashfunction_number-1` liefert `hashfunction_get` Informationen zur `i`-ten Hash-Funktion als Zeiger auf eine Struktur vom Typ `Hashfunction`. Diese enthält den Namen und einen Zeiger auf die Hash-Funktion.

4. In einer Schleife über alle Hash-Funktionen wenden Sie die aktuelle Hash-Funktion nun auf alle Worte aus der Menge an und speichern Sie die Hash-Werte in einer geeigneten Datenstruktur. Damit können Sie für alle $i \in H(h, T)$, den Wert $f(h, T, i)$ bestimmen. Hierzu gibt es zwei Möglichkeiten:
   - Man speichert für eine gegebene Hash-Funktion alle Hash-Werte in einem Array, sortiert dieses und kann dann in einem linearen Durchlauf die Häufigkeit eines jeden Hash-Wertes bestimmen.
   - Man nutzt eine `map`, die Hash-Werte (als Schlüssel) auf Ihre Häufigkeit (als Werte) abbildet.

   Aus den Häufigkeiten der Hash-Werte kann man schließlich das Kollisionsmaß der Hash-Funktion berechnen.

Als Ausgabe soll Ihr Programm für jede Hash-Funktion den Namen sowie das Kollisionsmaß der Hash-Funktion in einer Tabulator-separierten Zeile ausgeben. Diese Zeilen sollen aufsteigend nach dem Kollisionsmaß sortiert sein. Wenn die Kollisionsmaße von zwei Hashfunktionen sich um weniger als 10^{-8} unterscheiden, sollen sie als identisch betrachtet werden. In diesem Fall soll die Sortierung lexikographisch aufsteigend nach dem Namen der Hash-Funktion erfolgen.

In den Materialien finden Sie ein Makefile zum Kompilieren Ihres Programms sowie eine Testdatei mit dem erwarteten Ergebnis. Durch `make test` verifizieren Sie die Korrektheit Ihrer Implementierung für diese Testdaten.

Punkteverteilung:

- 1 Punkt für die korrekte Berechnung der Wort-Menge.
- 1 Punkt für die Iteration über alle Hash-Funktionen zur Berechnung der Hash-Werte für die Wort-Menge
- 1 Punkt für die Berechnung der Kollisionsmaße für alle Hash-Funktionen
- 1 Punkt für die Sortierung entsprechend der Aufgabenstellung
- 1 Punkt für die bestandenen Tests

Zur Bearbeitung dieser Aufgabe ist es hilfreich, die Abschnitte 11, 21, 34, 35, 37, 39 der Vorlesung (siehe Spalte *Nummer* in der Tabelle in `pfn2_vorlesung_2020.html`), zu kennen.

Bitte die Lösungen zu diesen Aufgaben bis zum 30.06.2020 um 18:00 Uhr an pfn2@zbh.uni-hamburg.de schicken.