Lehrstuhl für Systemsicherheit
Prof. Dr. Thorsten Holz
M.Sc. Thomas Hupperich
M.Sc. Katharina Kohls
M.Sc. Jannik Pewny

Vorlesung Informatik II
Sommersemester 2016
Ruhr-Universität Bochum

4. Übung

06. Juni 2016
Abgabe der Hausaufgaben per `git` bis zum 20. Juni 2016, 14:00 Uhr
Bei Fragen und Problemen können Sie sich per Moodle an Kommilitonen und Betreuer wenden.

1 Leben in der WG (Insgesamt 40 Punkte)

In Ihrer WG haben Sie eine Schale mit Orangen. Da Ihre Mitbewohner jedoch nicht vertrauenswürdig sind und Ihnen nur die schlechtesten Orangen übrig lassen würden, müssen Sie selbst zusehen, dass Sie auch welche von den guten Orangen abkriegen, indem Sie die Orangen nach Güte sortieren und die besten zuerst essen.

1.1 Die Güte der Orangen (10 Punkte)

Eine Orange hat für Sie hauptsächlich zwei Eigenschaften: Süße (sweetness) und Saftigkeit (juicyness), wobei süße Orangen besser sind. Natürlich. Wenn zwei Orangen gleich süß sind, dann ist die saftigere davon besser.

Implementieren Sie eine Klasse `Orange` mit dem Konstruktor `Orange(int sweetness, int juicyness)` und mit der Methode `bool operator<(Orange other)`, welche die oben genannte Güte-Einstufung abbildet.

Beispiel:
Tolle Orange: Süße = 9000, Saftigkeit = 5000
Doofe Orange: Süße = 5, Saftigkeit = 6000

Tolle Orange < Doofe Orange: false
Doofe Orange < Tolle Orange: true
Doofe Orange < Doofe Orange: false

1.2 Orangen-Sortierung: Quicksort (30 Punkte)

Sie wissen, dass `Quicksort` zum Sortieren von Orangen hervorragend geeignet sein soll. Allerdings wollen Sie natürlich die tollen Orangen zuerst essen, weshalb Ihr Algorithmus nicht von klein nach groß (doofe bis tolle Orangen), sondern von groß nach klein (tolle bis doofe Orangen) sortieren soll. Heißt: Nach dem Sortieren soll sich die süßeste und saftigste Orange am Anfang des Liste befinden.

Implementieren Sie dazu eine Funktion `void reverse_quick_sort(std::vector<Orange> *oranges, int left, right)`, welche eine Liste von Orangen (`oranges`) zwischen den Indizes `left` und `right` in umgekehrter Reihenfolge sortiert.

Beispiel:
Tolle Orange: Süße = 9000, Saftigkeit = 5000
So-lala Orange: Süße = 9000, Saftigkeit = 4
Doofe Orange: Süße = 5, Saftigkeit = 6000

`reverse_quick_sort(*[So-lala Orange, Doofe Orange, Tolle Orange], 0, 2)`
⇒ Tolle Orange, So-lala Orange, Doofe Orange

Hinweis:

- Nicht schummeln: Normaler Quicksort + Ergebnisliste umdrehen zählt nicht. die Ergebnisliste um.
- Nicht < ist nicht das gleiche wie >.

Abgabe

Pushen Sie die Implementierung Ihrer Orangen-Klasse in die Dateien `/u4/a1/Orange.cpp` bzw. `/u4/a1/Orange.hpp`, sowie die Implementierung Ihrer Sortierung in die Dateien `/u4/a1/reserve_quick_sort.cpp` bzw. `/u4/a1/reserve_quick_sort.hpp`.

2 Apples $\neq$ Oranges (Insgesamt 60 Punkte)

Sie haben ein ähnliches Problem mit den Äpfeln. Und Sie wollen auf jeden Fall die besten Äpfel zuerst Essen. Dann bleiben für die Anderen zwar nur die doofen Äpfel übrig, aber das haben Ihr Mitbewohner Timo und Felicitas sich selbst zuzuschreiben. Sie könnten ja auch einfach mal den Abwasch machen oder den Müll runterbringen, aber neeeeeeiiiiin...

2.1 Die Güte der Äpfel (10 Punkte)

Ähnlich wie eine Orange hat ein Apfel für Sie hauptsächlich zwei Eigenschaften: Süße (sweetness) und Saftigkeit (juicyness). Selbstverständlich sind süße Äpfel besser als weniger süße, aber sie müssen auch schon ziemlich saftig sein. Süßer ist Ihnen aber wichtiger: Exakt drei mal so wichtig.

Implementieren Sie eine Klasse `Apple` mit dem Konstruktor `Apple(int sweetness, int juicyness)` und mit der Methode `bool operator<(Apple other)`, welche die oben genannte Güte-Einstufung abbildet.

Beispiel:
Toller Apfel: Süße = 2000, Saftigkeit = 1000
Doofer Apfel: Süße = 1000, Saftigkeit = 3000

Toller Apfel < Doofer Apfel: false
Doofer Apfel < Toller Apfel: true
Doofer Apfel < Doofer Apfel: false

Hinweis:

- Stellen Sie die Wichtigkeit der Eigenschaften in einer Summe dar.

2.2 Nur die besten Äpfel: Binary Heap (40 Punkte)

Das Sortieren von Orangen hat ja ziemlich gut funktioniert, allerdings ist ihnen aufgefallen, dass Sie nur an einigen wenigen (nämlich den besten) Äpfeln interessiert sind und es daher nicht sonderlich effizient ist, immer die ganze Liste zu sortieren. Ein Binary Heap, bei welchem Sie nach und nach die besten Äpfel entnehmen können, wäre wohl die bessere Lösung.

Implementieren Sie dazu die Methoden der Klasse `Binary_heap`, welche Sie in den beigelegten Dateien finden. Anders als in der in der Vorlesung vorgestellten Lösung soll Ihre Implementierung jedoch kein Array statischer Größe verwenden, sondern einen dynamisch wachsenden `std::vector`.

Hierdurch entfällt die Notwendigkeit des Parameters `capacity` beim Konstruktor, sowie die der Variable `N`; letztere dürfen Sie aber trotzdem verwenden, wenn Sie dies möchten.

Hinweis:

- Die in der Vorlesung gezeigt Implementierung verwendet ein dummy-Element an 0-ter Stelle. Sie müssen also entweder im Konstruktor auch ein solches Hinzufügen, oder aber die Indizes im Algorithmus korrigieren.
- `.size()` liefert die Größe eines `std::vector`s
- `.pop_back()` entfernt das letzte Element eines `std::vector`s

Abgabe

Pushen Sie die Implementierung Ihrer Apfel-Klasse in die Dateien `/u4/a2/Apple.cpp` bzw. `/u4/a2/Apple.hpp`, sowie die Implementierung Ihres Binary Heap in die Dateien `/u4/a2/Binary_heap.cpp` bzw. `/u4/a2/Binary_heap.hpp`.