

# EXAMEN SYSTEEMPROGRAMMEREN - 3 FEBRUARI 2023

2de Bachelor Informatica - Faculteit Wetenschappen - Academiejaar 2022-2023

Naam:

Studentennummer:

Indieningstijdstip:

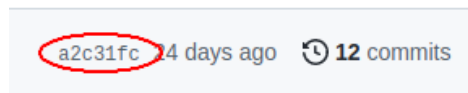
Laatste commit hash:

## 1 INDIENEN

Volg de instructies op GitHub (<https://github.ugent.be/sysprog-examen-2023-02>). Enkel onderstaande bestanden mogen aangepast worden:

- gestart.txt
- custom\_string.cpp
- quick\_map.cpp
- main.cpp (enkel voor jouw eigen code te testen)

Op het einde van het examen vul je bovenaan het examenformulier het tijdstip en de hash in van jouw laatste commit. Figuur 1 toont een voorbeeld van waar je jouw laatste commit hash kan terugvinden. Ga hiervoor naar jouw repository, de commit hash vind je terug naast het aantal commits.



Figuur 1: Commit hash

Tijdens het examen is het gebruik van ondersteunende extensies, zoals bijvoorbeeld geavanceerde *code completion*, niet toegestaan. Dit zijn onder andere ook extensies die gebruikmaken van AI. Extensies zoals Copilot, ChatGPT, ... mogen dus niet gebruikt worden. Verwijder deze extensies voor de aanvang van het examen. Uitschakelen is niet voldoende! Indien een dergelijke extensie geïnstalleerd is tijdens het examen (ook als je hier geen gebruik van maakt), staat dit gelijk aan fraude en resulteert dit in een 0 voor het examen. Daarnaast zal de examencommissie een beslissing nemen over de mogelijkheid tot slagen van andere vakken.

## 2 PROGRAMMEREN IN C++ (08U30-11U00) (OP 20 PUNTEN)

Het opslaan van *sleutel-waarde* paren kan voor allerlei doelen gebruikt worden. Een voorbeeld hiervan is het bewaren van het examencijfer van een student. Hier is de naam van de student de sleutel en de score de waarde.

Daarnaast is het in verschillende toepassingen van de computerwetenschappen vaak nodig om snel toegang te krijgen tot de x-aantal kleinste en x-aantal grootste entries. Als we bijvoorbeeld snel de 3 beste en 3 slechtste examencijfers willen opzoeken.

In deze opgave is het de bedoeling om een eigen datastructuur te bouwen die deze twee eigenschappen combineert, we noemen deze datastructuur `quick_map`, omdat het snelle toegang geeft tot de kleinste en grootste waarden. De datastructuur maakt intern gebruik van een `std::map` voor het bewaren van de *sleutel-waarde* paren. Om de waarden van de x-aantal kleinste en de x-aantal grootste waarden bij te houden, wordt gebruik gemaakt van arrays.

In de opgave staan `//TODO` comments die aanduiden waar code verwacht wordt. Hieronder wordt iedere methode toegelicht.

## Eigen string klasse [op 8 van de 20 punten]

De `quick_map` maakt gebruik van sleutels van het type `custom_string`. Dit is een eigen, vereenvoudigde implementatie van `std::string`. De bedoeling is om de basisfunctionaliteit van `std::string` aan te bieden.

Het gebruik van `std::string` of andere ingebouwde string containers is **niet** toegestaan in de gehele examenopgave. Enkel operaties op `char*`, zoals hieronder beschreven, zijn toegestaan.

Om de basisfunctionaliteit van `std::string` na te bootsen, wordt gebruik gemaakt van functies die terug te vinden zijn in de `<cstring>` header. Dit is de C++ variant van de C-header `<string.h>`. Typische C-string functies zoals `strlen` en `strcpy` worden hier gedefinieerd. Het is de bedoeling om, onder andere, met dergelijke functies de gewenste werking te verkrijgen.

Hieronder wordt een overzicht gegeven van de gevraagde functies.

```
custom_string();
```

Dit is de default constructor van `custom_string`. Deze initialiseert de datavelden, maar allocceert nog geen geheugen. [0.25 punten]

```
custom_string(const char *c_str);
```

Deze constructor heeft een C-string als argument. Deze constructor dient deze C-string op te slaan in `char *content` (diepe kopie). In deze constructor wordt de minimale hoeveelheid geheugen gealloceerd om de C-string te kunnen opslaan. Hou ook steeds bij hoe lang de `custom_string` is (in `length`) en wat de huidige capaciteit is (in `capacity`). [1 punt]

```
~custom_string();
```

De destructor van `custom_string` geeft al het geheugen dat gealloceerd werd terug vrij. [0.25 punten]

```
custom_string(const custom_string &str);
```

Copy constructor, die de inhoud van het opgegeven `str` argument kopieert (diepe kopie) en zo een nieuwe `custom_string` aanmaakt. Maak hier gebruik van een *delegating constructor* om code duplicatie te voorkomen. [0.5 punten]

```
custom_string(custom_string &&str);
```

Move constructor. Er wordt een nieuwe `custom_string` aangemaakt en de inhoud van het `str` argument wordt verplaatst. [1 punt]

```
custom_string &operator=(const custom_string &str);
```

Assignment operator. De bedoeling is om de inhoud van `custom_string` in het `str` argument te kopiëren (diepe kopie) naar de huidige `custom_string`. [1.5 punten]

**Belangrijk:** er mag enkel nieuw geheugen gealloceerd worden indien dit nodig is. Deze assignment operator moet, zoals altijd het geval is bij assignment operators, cascading (`a = b = c`) ondersteunen.

```
custom_string &operator=(custom_string &&str);
```

Move assignment operator. De inhoud van het `str` argument wordt verplaatst naar de huidige `custom_string`. [1.5 punten]

```
bool operator<(const custom_string &str) const;
```

Geef `true` terug als de huidige `custom_string` (alfabetisch) kleiner is dan `str`. Geef `false` terug in alle andere gevallen. Maak hiervoor gebruik van een functie uit de standaardbibliotheek die toelaat om `char *`'s met elkaar te vergelijken. Dit is een belangrijke operator, indien deze niet correct is zal dit mogelijks problemen geven bij de implementatie van de `quick_map`. [1.25 punten]

```
bool operator==(const custom_string &str)const;
```

Geef `true` terug wanneer de huidige `custom_string` gelijk is aan `str`. Geef `false` terug in alle andere gevallen. [0.25 punten]

```
friend std::ostream &operator<<(std::ostream &os, const custom_string &str);
```

Implementeer de output stream operator. Een voorbeeld van de werking wordt hieronder gegeven. [0.5 punten]

```
custom_string my_string("systeemprogrammeren <3");
std::cout << my_string << std::endl;
```

```
• * Executing task in folder solution-code: ./main
systeemprogrammeren <3
```

### Quick map [op 12 van de 20 punten]

De `quick_map` maakt intern gebruik van een `std::map` met naam `internal_map`. De `internal_map` heeft als key `custom_string` en als value `std::shared_ptr<const node>`. De klasse `node` is gegeven in `node.hpp`.

Een aantal vereenvoudigingen en assumpties:

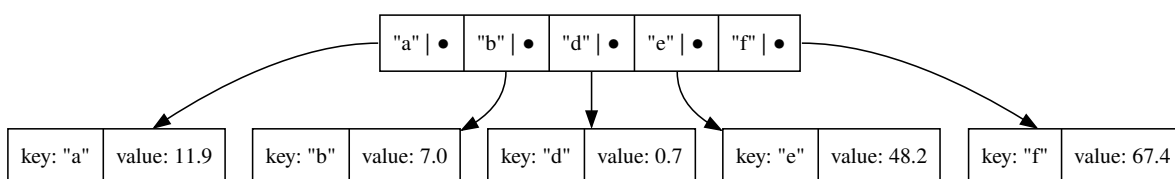
- Alle keys die worden toegevoegd zijn gegarandeerd uniek, hier dient niet op gecontroleerd te worden.
- De waarden zijn gegarandeerd  $\geq 0$ .

Zoals hierboven al kort besproken, houdt een `quick_map` ook bij welke de kleinste en grootste waarden zijn. De hoeveelheid grootste en kleinste waarden dat een `quick_map` dient bij te houden, wordt meegegeven met de constructor en wordt daarna bewaard in de `quick_access_amount` variabele.

Er zijn twee arrays gedeclareerd: `smallest_values` en `largest_values`.

Deze arrays geven toegang tot de `quick_access_amount` kleinste en grootste nodes, door elk shared pointers bij te houden die wijzen naar de kleinste en grootste nodes. Deze arrays noemen we vanaf nu de *quick-access* arrays. Een pointer naar een nieuwe `node` komt bijvoorbeeld in de `smallest_values` array als die `node` zijn `double` value klein genoeg is. Het totaal aantal bewaarde nodes in `internal_map` wordt bijgehouden in de `node_count` variabele.

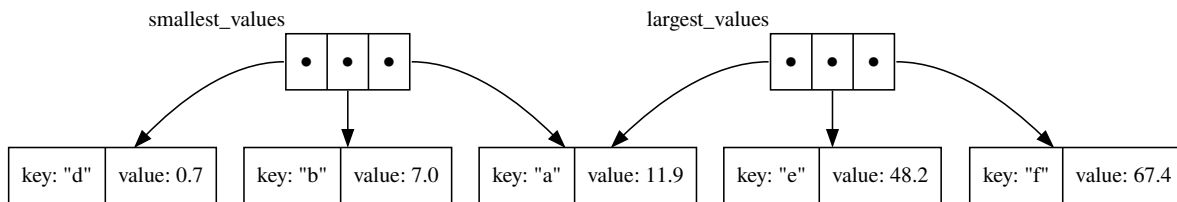
Figuur 2 geeft een voorstelling van hoe data opgeslagen wordt in de `internal_map` van `quick_map`. Hier is zichtbaar hoe een key ("a", "b", ...) wijst (door het gebruik van een `std::shared_ptr`) naar de overeenkomende `node`.



Figuur 2: Voorbeeld van `internal_map` bewaard in `quick_map`

Figuur 3 geeft weer hoe de arrays `smallest_values` en `largest_values` wijzen (`std::shared_ptr`) naar de overeenkomende kleinste en grootste nodes. De *quick-access* arrays wijzen dus naar `node`-objecten, waar de `internal_map` ook naar wijst (m.a.w. het zijn dezelfde objecten, er bestaat op eenzelfde moment maar 1 `node` met een bepaalde key-value.)

**Opmerking:** De *quick-access* arrays hoeven **niet** gesorteerd te zijn (dit is toevallig wel zo in Figuur 3, maar hoeft dus niet).



Figuur 3: Voorbeeld van `smallest_values` en `largest_values`, wanneer `quick_access_amount` gelijk is aan 3

Er is gekozen om gebruik te maken van `std::shared_ptr` om het geheugenbeheer van `node` objecten te automatiseren. Dit omdat er op verschillende plaatsen naar een `node` object gewezen kan worden en zo dus *dangling pointers* of *memory leaks* worden vermeden.

Hieronder wordt een overzicht gegeven van de gevraagde methoden.

```
quick_map(int quick_access_amount);
```

Constructor die als argument `quick_access_amount` heeft. Het `quick_access_amount` argument geeft aan hoe groot de arrays `smallest_values` en `largest_values` moeten zijn, de grootte van de *quick-access* arrays dus. Deze constructor allocceert geheugen voor deze arrays en initialiseert alle datavelden van de klasse `quick_map`. **[1 punt]**

```
quick_map(int quick_access_amount, std::pair<const custom_string, double>
*elements, int length);
```

Deze constructor heeft net als de constructor hierboven een argument dat meegeeft hoe groot de *quick-access* arrays moeten zijn. Daarnaast wordt er ook een array (`elements`) meegegeven van `std::pair` elementen. Ieder `std::pair` element stelt een key-value paar voor dat moet toegevoegd worden aan de `quick_map`. Elk element moet ook toegevoegd worden aan een *quick-access* array indien het aan de voorwaarden voldoet (`double` value is klein of groot genoeg). De lengte van de array `elements` wordt meegegeven met `length`. **[1.5 punten]**

Tip: Gebruik de `quick_map::insert()` methode om duplicate code te vermijden.

```
~quick_map();
```

De destructor van `quick_map` geeft al het gealloceerde geheugen terug vrij. **[0.5 punten]**

```
void insert(const custom_string &key, double val);
```

Deze functie voegt een nieuwe `node` toe aan de `quick_map`. Indien de nieuwe `node` aan de voorwaarden voldoet (waarde is klein of groot genoeg) moet het ook toegevoegd worden aan één of beide *quick-access* arrays. Vergeet de `node_count` niet aan te passen. **[4 punten]**

Tip: Een `shared_ptr` kan op meerdere manieren aangemaakt worden. Door middel van de `shared_ptr` constructor, of door gebruik te maken van `std::make_shared`. Jullie mogen zelf kiezen welke optie je gebruikt. Een voorbeeld wordt hieronder gegeven.

```
// Constructor versie
auto one = std::shared_ptr<const node>(new node(key, val));

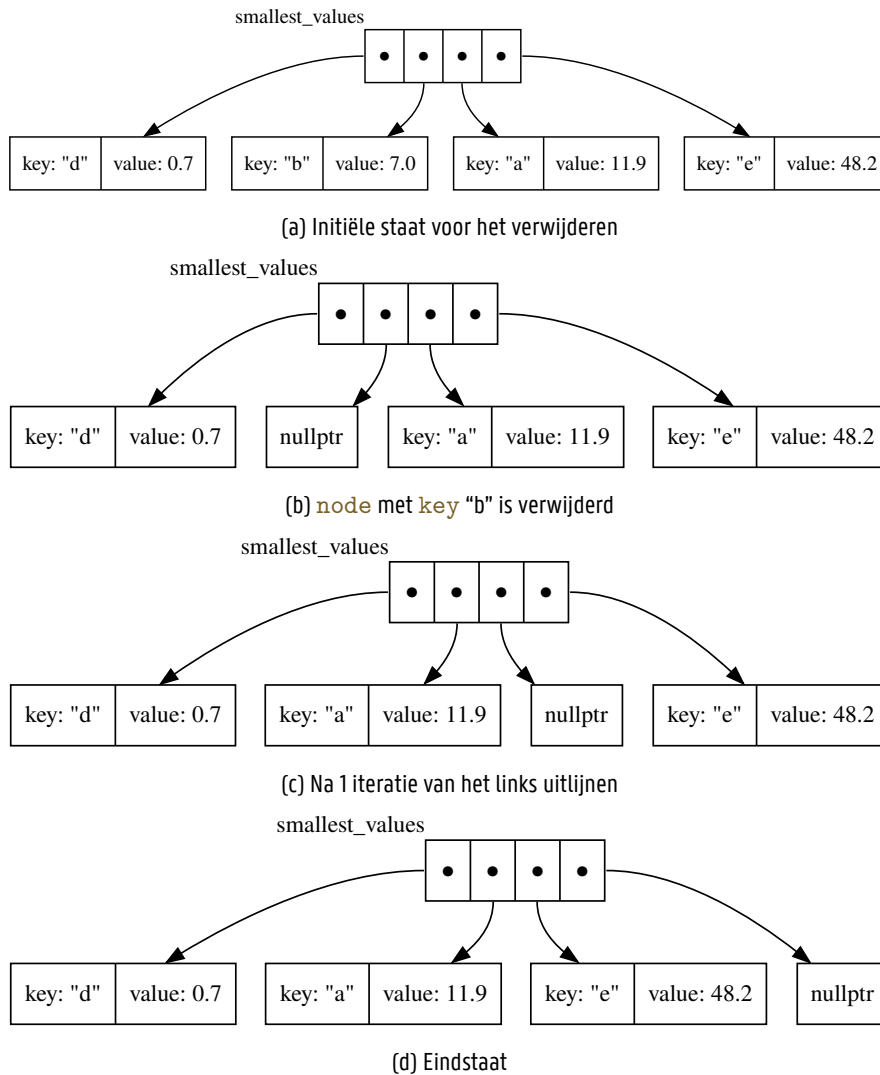
// std::make_shared versie
auto two = std::make_shared<const node>(key, val);
```

```
double remove(const custom_string &key);
```

Deze functie verwijdert de `node` met `key` uit de `quick_map` en geeft de waarde van die `node` terug. Indien de `node` niet aanwezig is in de `quick_map` wordt `-1` teruggegeven (maak hiervoor gebruik van de gegeven `quick_map::search()` methode). De `node` moet ook uit de *quick-access* arrays verwijderd

worden indien het daar aanwezig is. Het is de bedoeling dat de *quick-access* arrays steeds links uitgelijnd zijn. Hiermee wordt bedoeld dat er enkel lege plaatsen aanwezig mogen zijn op de meest rechtse plaatsen van de arrays (zie Figuur 4 voor een illustratie). Vergeet de `node_count` ook niet aan te passen. [3 punten]

**Belangrijk:** Ter vereenvoudiging van deze methode is het **niet** de bedoeling om de plaats(en) van de verwijderde `node`, in de *quick-access* arrays, terug op te vullen met een `node` uit `internal_map` die aan de voorwaarden zou voldoen. De plaats(en) van de verwijderde `node` hoort een `nullptr` te worden, waarna de arrays uitgelijnd worden.



Figuur 4: Illustratie links uitlijnen array na het verwijderen van een element

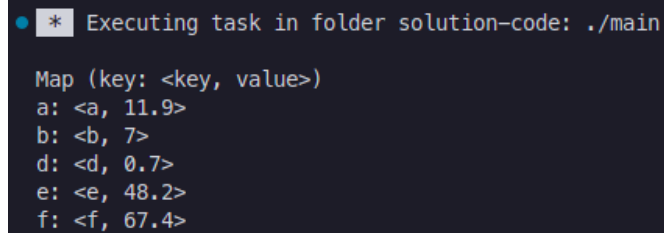
```
friend std::ostream &operator<<(std::ostream &os, const quick_map &map);
```

Deze operator schrijft de `quick_map` naar de output stream. Een voorbeeld van de gewenste output wordt hieronder weergegeven. [2 punten]

```
quick_map my_quick_map(3);

my_quick_map.insert("e", 48.2);
// ...

std::cout << my_quick_map << std::endl;
```



```
• * Executing task in folder solution-code: ./main

Map (key: <key, value>)
a: <a, 11.9>
b: <b, 7>
d: <d, 0.7>
e: <e, 48.2>
f: <f, 67.4>
```

### Gegeven methoden

```
double search(const custom_string &key) const;
```

Deze methode zoekt of de gegeven `key` aanwezig is in de `quick_map`. Indien de `key` aanwezig is, wordt de waarde van deze `node` teruggegeven. Indien de `key` niet aanwezig is, wordt `-1` teruggegeven.

```
void print_smallest_values() const;
```

Deze methode print de `smallest_values` array. Deze methode kan gebruikt worden voor debugging of controle van de correcte werking.

```
void print_largest_values() const;
```

Deze methode print de `largest_values` array.