UNIVERSITEIT GENT

FACULTEIT WETENSCHAPPEN

Systeemprogrammeren

Academiejaar 2018–2019

sysprog@lists.UGent.be

Oefeningenset 3
Omgaan met meerdimensionale arrays

Hashtabel

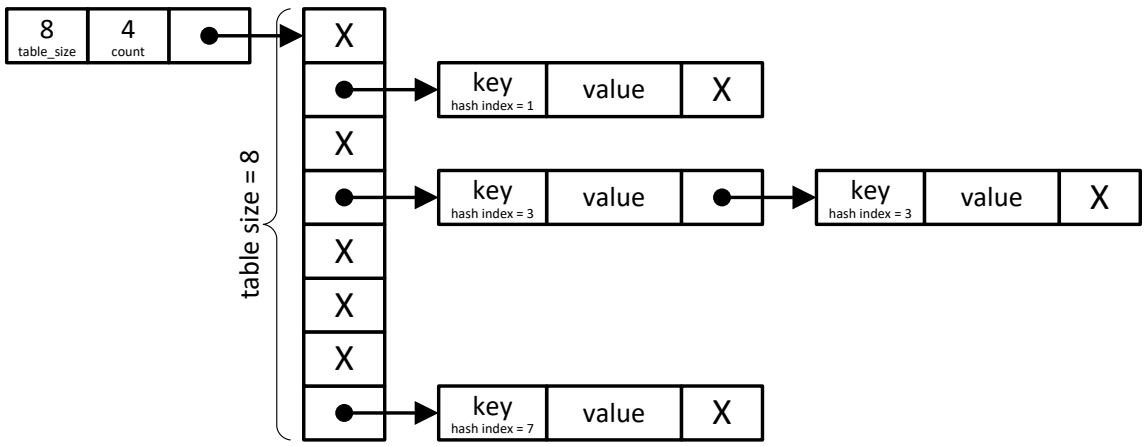
Situering

Een **map** is een datastructuur waarbij sleutels worden geassocieerd met waarden. Een map is dus een collectie van (sleutel, waarde)-combinaties. Daarbij hebben de sleutels van alle elementen hetzelfde type, en hebben de waarden van alle elementen hetzelfde type; het type van de sleutels kan echter verschillen van het type van de waarden. In een map kan nooit meer dan één element dezelfde sleutel hebben, maar meerdere elementen met dezelfde waarde kunnen wel in dezelfde map voorkomen. Dergelijke associatieve structuren worden voornamelijk gebruikt voor zoekoperaties waar men, voor een gegeven sleutel, bijvoorbeeld een naam, bijbehorende informatie wil opzoeken, bijvoorbeeld de woonplaats.

Een **hashtabel** of **hashmap** is een map waarbij een **hashfunctie** gebruikt wordt om efficiënt de opgezochte sleutel te vinden. Zo'n hashfunctie is een sleuteltransformatiefunctie die de sleutels transformeert naar een hashwaarde, die zo willekeurig mogelijk gekozen is (en uniform verdeeld) binnen het bereik van die hashfunctie. Wanneer een sleutel in de hashtabel wordt opgezocht (of toegevoegd), wordt eerst de hashwaarde van de sleutel berekend. Deze hashwaarde, modulo de grootte van de hashtabel, dient als index in de hashtabel.

Met de **tabelgrootte** *M* van de hashtabel bedoelen we de grootte van de tabel waarin de sleutel-waarde combinaties worden opgeslagen. De hashfunctie zal een sleutel afbeelden op een numerieke waarde in zijn bereik; dit bereik is over het algemeen groter dan de tabelgrootte. Daarom wordt de index van een element in de tabel bepaald als de rest na deling door de tabelgrootte van de teruggegeven hashwaarde van de hashfunctie toegepast op de sleutel van het element.

Wanneer je meerdere elementen aan een hashtabel toevoegt, kan het gebeuren dat twee sleutels dezelfde hashindices hebben. Dit heet een **botsing** (*collision*) of indexconflict. Wanneer de hashtabel meer elementen bevat dan de tabelgrootte, is het zelfs zeker dat zo'n botsing optreedt. Er zijn verschillende manieren om met botsingen om te gaan. In deze opgave implementeren we een hashtabel met **afzonderlijk geschakelde lijsten** (*separate chaining*). Hierbij bevat de hashtabel een tabelrij waarin ieder element een geschakelde lijst (*linked list*) is van elementen waarvan de sleutels dezelfde hashwaarde hebben.



De **bezettingsgraad** of **beladingsfactor** (*load factor*) van een hashtabel is het aantal elementen *N* gedeeld door de tabelgrootte *M*. Wanneer de bezettingsgraad voldoende kleiner is dan 1, en de kans op botsingen

dus nog klein is, gebeurt in een hashtabel het opzoeken van de waarde die bij de sleutel past in constante tijd. Het voordeel van een hashtabel met afzonderlijk gelinkte lijsten is dat de hashtabel blijft werken als de bezettingsgraad groter wordt dan 1; dit in tegenstelling tot hashtabellen met open-adres-schema's.

Naarmate de bezettingsgraad stijgt, daalt de prestatie slechts geleidelijk en benadert ze steeds meer een lineaire evenredigheid met de beladingsfactor (of dus met het aantal elementen). Wanneer de bezettingsgraad te groot (of te klein) wordt, kiezen bepaalde implementaties van hashtabellen ervoor om de hashtabel opnieuw te hashen: voor iedere sleutel wordt de hashwaarde herberekend en wordt een nieuwe index bepaald, zijnde de rest bij deling van de hashwaarde door een grotere (of kleinere) tabelgrootte. Daarom wordt de tabelgrootte en het aantal toegevoegde elementen per hashtabel bijgehouden. Indien de bezettingsgraad te groot wordt, wordt de tabel vergroot en wordt elke sleutel opnieuw gehasht. Analoog wordt, indien de bezettingsgraad te klein wordt, de tabel verkleind en elke sleutel opnieuw gehasht. In beide gevallen wordt dit een **rehash**operatie genoemd.

Een nadeel van hashtabellen is dat de sleutels in een willekeurige volgorde in het geheugen staan. Als toegang tot de sleutels in een bepaalde volgorde nodig is, of als men alle opeenvolgende sleutels in een bepaald interval nodig heeft, is een hashtabel niet de meest efficiënte oplossing.

Indexer voor tekst-gebaseerde boodschappen

Bij het beheren van tekst-gebaseerde boodschappen (bijv. emails, twitter-berichten, sms-berichten, etc) is het vaak nodig te zoeken naar specifieke woorden, en op basis van het resultaat van de zoekopdracht, de gewenste tekstboodschap te filteren uit de (vaak zeer grote) hoeveelheid berichten (bijv. emails in je inbox). Omdat een exhaustieve zoekopdracht in dit geval veel te lang zou duren, bouwt men in dergelijke zoekproblemen vaak een index op van de woorden waarop gezocht kan worden. Men slaat de woorden (en de plaatsen waarop ze voorkomen in de tekstboodschappen) dan op in een datastructuur waarin de opzoekoperatie zo efficiënt mogelijk kan uitgevoerd worden. We zullen een hashtabel gebruiken om een dergelijke index te implementeren.

Opgave A: implementatie van een hashtabel

De sleutels in de hashtabel zijn dus de woorden in de tekstboodschappen (`char*`). Een woord kan uiteraard meermaals voorkomen; daarom wordt voor elk woord een array bijgehouden die alle 'indices' (i.e. posities) bijhoudt waarop dit woord voorkomt.

Gecombineerd met de structuur van de hashtabel zoals beschreven in deel 1, levert dit de volgende structuur op: de hashtabel is een array van enkelvoudig geschakelde lijsten, en elk element in zo'n geschakelde lijst houdt een sleutel (=het te indexeren woord) en een array van index-waarden bij. Omdat in C de lengte van een array niet op te vragen is, moet ook een `int` bijgehouden worden die op elk moment het aantal elementen van deze value-array bevat. De 'value' van de hashtabel-elementen uit de figuur in deel 1 bestaat dus eigenlijk uit een array van index-waarden en een integer die de lengte van deze array bijhoudt.

De waarden in de hashtabel bevatten dus arrays met elementen van het volgende type:

```
typedef struct{
   int messagenr; /* nummer van het bericht waarin het woord voorkomt */
   int wordnr; /* volgnummer van het woord in het bericht */
```

```
} index;
```

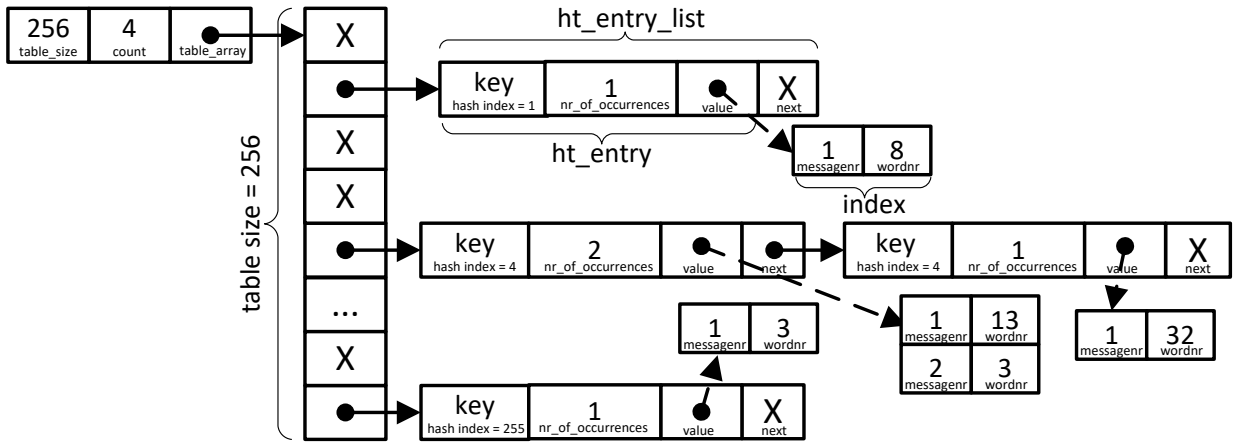
De geschakelde lijsten bestaan uit volgende structuren:

```
typedef struct ht_entry {
   char* key; /* sleutel == te indexeren woord */
   int nr_of_occurrences; /* aantal indices == lengte van value array */
   index* value; /* array met de indices waarop key voorkomt*/
} ht_entry;
```

De lijsten zelf worden gevormd en gelinkt door:

```
typedef struct ht_entry_list {
   ht_entry current;
   struct ht_entry_list* next;
} ht_entry_list;
```

Een overzicht van de gehele datastructuur is hieronder gegeven:



Als hashfunctie zal je een functie nodig hebben die een string (`char*`) transformeert naar een getal en voldoet aan de vereisten van een goede hashfunctie, zoals een zo uniform mogelijke verdeling van de hashwaarden over het bereik van de hashfunctie. Hiervoor mag je gebruik maken van Bob Jenkins' JOAAT-functie ("*Jenkins one at a time*"). Deze functie

```
unsigned int ht_joaat_hash (const char* key)
```

is gegeven. Deze functie zet een string (`const char*`) om in een `unsigned int`, tussen 0 en 2^{32} –1. Het bereik van deze hashfunctie is dus 2^{32}.

De functie `rehash` hasht alle elementen in de tabel opnieuw indien nodig. De hulpfunctie

```
int rehash_table_size(const hash_table* ht)
```

is eveneens gegeven. Deze laatste functie geeft 0 terug als er niet opnieuw gehasht moet worden. Maar als de tabelgrootte en het aantal elementen te ver uit elkaar liggen, dan geeft `rehash_table_size` een nieuwe tabelgrootte terug. In de functie `rehash` wordt deze aanpassing van de tabelgrootte uitgevoerd, en dit uiteraard vóór het opnieuw hashen en opnieuw invoegen van de bestaande elementen. Vanzelfsprekend wordt hierbij het niet meer gebruikte geheugen, gealloceerd door de oude tabelrij, vrijgegeven.

Iedere keer als het aantal elementen in de hashtabel verandert (dus zowel wanneer er elementen toegevoegd of verwijderd worden) moet er opnieuw gehasht worden **indien** de functie `rehash_table_size` dit aangeeft. Hiertoe is de functie `rehash` gegeven, vergeet deze dus niet op te roepen waar nodig.

In **`hash_table.h`:**

- Vul de `struct hash_table` aan. Hou er rekening mee dat de tabelgrootte en het aantal toegevoegde elementen per hashtabel wordt bijgehouden.

In **`hash_table.c`:**

Implementeer de volgende functies:

- `hash_table* ht_create()`

  Deze functie creëert een hashtabel en geeft een pointer terug die wijst naar deze nieuw gealloceerde structuur. De array krijgt als tabelgrootte `DEFAULT_TABLE_SIZE`. Indien er niet genoeg geheugen voorhanden is, geeft deze functie `NULL` terug.

- `int ht_get_value(const hash_table* ht, const char* key, index** i)`

  Deze functie geeft het aantal waarden (indices) terug die in de hashtabel met de sleutel geassocieerd wordt. De bijhorende array `i` die de waarden zal bevatten wordt 'by reference' doorgegeven. Geeft `NOT_AN_ELEMENT` terug als de sleutel niet in de hashtabel voorkomt.

- `int ht_set_value(hash_table* ht, const char* key, index value)`

  Deze functie zorgt ervoor dat de hashtabel de index value met de sleutel associeert, en voegt dus m.a.w. de index toe in de array van indices (die hiertoe dus vergroot moet worden. Vergeet ook `nr_of_occurrences` niet aan te passen). Geeft het aantal indexes terug die na uitvoeren van deze operatie met de sleutel in de hashtabel worden geassocieerd. Als een nieuw element wordt toegevoegd aan de hashtabel wordt van de sleutel een diepe kopie gemaakt. Deze methode roept ook `rehash` op.

- `int ht_remove_entry(ht_entry_list** list)`

  Verwijdert een gegeven element, en geeft het aantal indices terug die met de sleutel in de hashtabel werden geassocieerd. Geeft hierbij uiteraard het geheugen vrij dat voor dit element gealloceerd was, en laat `*list` wijzen naar het volgende lijstelement. Wordt opgeroepen door `ht_remove` en `ht_destroy`.

- `int ht_remove(hash_table* ht, const char* key)`

  Deze functie verwijdert het element met een gegeven sleutel uit de hashtabel. Alle gemapte indices worden hierbij verwijderd. Geeft het aantal indices terug dat met de sleutel werd geassocieerd. Geeft `NOT_AN_ELEMENT` terug indien de sleutel niet in de hashtabel voorkwam. In dat geval wordt uiteraard niets verwijderd. Maakt gebruik van `ht_remove_entry`. Deze methode roept ook `rehash` op.

- `void ht_destroy(hash_table** ht)`

  Deze functie geeft het geheugen gealloceerd door de hashtabel volledig vrij en zet de pointer naar de hashtabel op `NULL` om een *dangling pointer* te vermijden. Maakt gebruik van `ht_remove_entry`.

- `void ht_print(const hash_table* ht)`

  Print op het scherm per rij van de hashtabel:

  - het volgnummer van de rij zelf
  - gevolgd door de sleutels en hun corresponderende waarden in die rij.

  Volgnummers die overeenstemmen met een lege rij worden niet geprint.

Merk op:

- dat de `ht_create` functie zelf een hashtabel aanmaakt en een pointer naar de nieuw gealloceerde structuur teruggeeft.
- dat het NIET de bedoeling is om in de functies `ht_get_value` en `ht_set_value` de tabel lineair te overlopen; gebruik hiertoe de specifieke eigenschappen van hashtabellen zoals uitgelegd in het eerste deel van de opgave.

Opgave B: implementatie van een indexer

In deze opgave wordt gevraagd enkele gegeven tekstbestanden te indexeren met behulp van een hashtabel en de structuren en functies gedefinieerd in message_indexer.h.

```
typedef struct message{
    char* text;
    char** word_index;
    int text_size;
    int nr_of_words;
} message;
```

Hierin is `text` de volledige tekst van het bericht, `word_index` een array van pointers naar het begin van elk woord in `text` (ondiepe kopieën dus), en is hun volgnummer in `text` hun index in de array. De `int`s houden de lengte van hun respectieve arrays bij.

Implementeer de volgende functies in `message_indexer.c`:

- `void init_message(message* m,char* filename)`

  Initialiseert een bericht door het inlezen van de tekst en het opslaan van de tekstgrootte. Maakt hierbij gebruik van `read_file_into_string` (gegeven). Hierbij worden `nr_of_words` en `word_index` op 0 gezet.

- `void free_message(message m)`

  Geeft het geheugen ingenomen door de opgegeven message vrij.

- `int index_message(hash_table* ht, message* m, int messagenr)`

  Indexeert de woorden van het opgegeven bericht (opgeslagen in `m->text`) in de opgegeven hashtabel. Hierbij wordt in `m` ook de `word_index` aangemaakt. Dit is nodig om bij het opzoeken van een woord de woorden van de context (zie `print_word_context` in main.c) snel en correct terug te geven. Het `messagenr` is het volgnummer van het bericht en wordt gebruikt om het veld `messagenr` van de index-entry voor dit woord in te vullen.

  Pseudocode voor deze methode:

```
Overloop de woorden in tekst van het bericht via een pointer
Zolang niet aan het einde van de tekst, doe:
      lees het volgende woord (woorden bestaan uit hoofd- of kleine
letters en eventueel koppeltekens, andere karakters mag je negeren)
      maak een index voor dit woord en stop het in de hashtabel
      (her)alloceer genoeg plaats voor de word_index array van het
bericht
      kopieer de pointer naar de start van het woord in de word_index
Geef het aantal geïndexeerde woorden terug
```

Merk op:

- dat de `init_message` functie een reeds aangemaakte `message struct` invult, dit in tegenstelling tot de `ht_create` methode van de hashtabel.
- dat de `word_index` ondiepe kopieën bijhoudt, dit zijn dus pointers naar plaatsen in `text`. Voor de woorden zelf moet dus geen plaats meer gealloceerd worden. Je moet echter wel plaats alloceren voor deze pointers zelf!

In het bestand main.c wordt afhankelijk van de opgegeven opdrachtlijnparameter de keuze tussen de modi van het programma gemaakt: als opdrachtlijnparameter moet een string meegegeven worden waarmee 1 van volgende 3 modi geselecteerd wordt:

- “debug_ht”: testen van de hashtabel met hardgecodeerde input
- “debug_mi”: testen van de message indexer met hardgecodeerde input
- “run”: voer de message indexer uit op een aantal berichten

De methode `print_word_context` print een aantal woorden, aangegeven door de constante `CONTEXTSIZE`, voor en na een opgegeven woord uit naar het scherm.