Practicum C: Gelinkte Lijsten

Informatica werktuigen

Academiejaar 2017-2018

Het doel van dit practicum is om meer vertrouwd te worden met pointers en geheugenbeheer in C. Bij het implementeren van datastructuren spelen beide een belangrijke rol, vandaar dat we hierop zullen focussen. Meer bepaald gaan jullie in dit practicum een (dubbel) gelinkte lijst en een stack implementeren.

1 Gedragscode

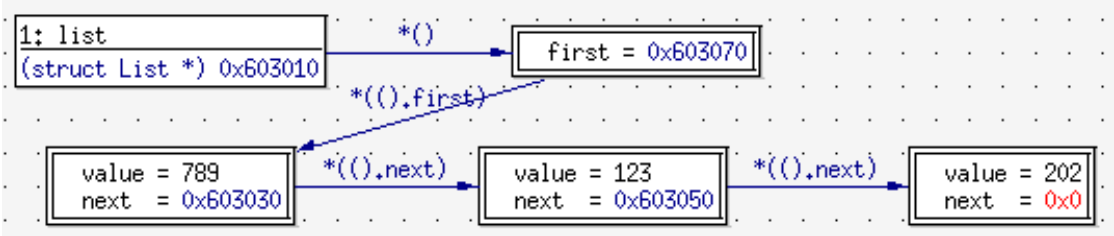
De practica worden gequoteerd, en het examenreglement is dan ook van toepassing. Soms is er echter wat onduidelijkheid over wat toegestaan is en niet inzake samenwerking bij opdrachten zoals deze.

De oplossing en/of programmacode die ingediend wordt moet volledig het resultaat zijn van werk dat je zelf gepresteerd hebt. Je mag je werk uiteraard bespreken met andere studenten, in de zin dat je praat over algemene oplossingsmethoden of algoritmen, maar de bespreking mag niet gaan over specifieke code of verslagtekst die je aan het schrijven bent, noch over specifieke resultaten die je wenst in te dienen. Als je het met anderen over je practicum hebt, mag dit er dus *nooit* toe leiden, dat je op om het even welk moment in het bezit bent van een geheel of gedeeltelijke kopie van het opgeloste practicum of verslag van anderen, onafhankelijk van of die code of verslag nu op papier staat of in elektronische vorm beschikbaar is, en onafhankelijk van wie de code of het verslag geschreven heeft (mede-studenten, eventueel uit andere studiejaren, volledige buitenstaanders, internet-bronnen, e.d.). Dit houdt tevens ook in dat er geen enkele geldige reden is om je code of verslag door te geven aan mede-studenten, noch om dit beschikbaar te stellen via publiek bereikbare directories of websites.

Elke student is verantwoordelijk voor de code en het werk dat hij of zij indient. Als tijdens de beoordeling van het practicum er twijfels zijn over het feit of het practicum zelf gemaakt is (bvb. gelijkaardige code, of oplossingen met andere practica), zal de student gevraagd worden hiervoor een verklaring te geven. Indien dit de twijfels niet wegwerkt, zal er worden overgegaan tot het melden van een onregelmatigheid, zoals voorzien in het onderwijs- en examenreglement (zie `http://www.kuleuven.be/onderwijs/oer/`).

2 Inleiding: Gelinkte Lijsten

Een *gelinkte lijst* is een datastructuur waarvoor het eenvoudig is om een element toe te voegen of te verwijderen. Een ander voordeel is dat de grootte van een gelinkte lijst dynamisch kan veranderen: er is geen limiet op het aantal elementen dat je aan de lijst kunt toevoegen. Dit staat in contrast met een *array*: deze is van een vaste grootte, waardoor het niet zo evident is om elementen toe te voegen of te verwijderen.



Figuur 1: Weergave van een enkelvoudige gelinkte lijst in `ddd`.

Zoals tijdens de les en de oefenzittingen, stellen we een (enkelvoudig gelinkte) lijst in C voor door de types `struct ListNode` en `struct List`:

```
1  struct ListNode {
2          int value;
3          struct ListNode *next;
4  };
5
6  struct List {
7          struct ListNode *first;
8  };
```

Het type `struct List` representeert de volledige gelinkte lijst, en `struct ListNode` representeert een node in de lijst. Elke node bevat een getal, en een pointer naar de volgende node. Een `NULL` pointer stelt het einde van de lijst voor. In Figuur 1 is een voorbeeld gegeven van een enkelvoudige gelinkte lijst die deze datastructuur gebruikt.

3 Opgave

Tijdens de les en oefenzitting hebben jullie al enkele operaties op gelinkte lijsten moeten implementeren. In dit practicum zal je een aantal extra operaties moeten implementeren. Daarnaast moeten jullie enkele basisoperaties op dubbel gelinkte lijsten en stacks implementeren. Op Toledo vinden jullie het bestand `practicumC.zip` dat de volgende bestanden bevat om jullie op weg te helpen:

`list.h` Het header bestand met alle declaraties (types en functies) die jullie moeten gebruiken. Aan dit bestand mag niets veranderd worden!

`list.c` Bevat lege definities van alle functies die jullie moeten implementeren. De implementatie van een aantal hulpfuncties die jullie van ons krijgen is hier ook terug te vinden. Bij de declaraties van de functies die jullie moeten implementeren, staat in commentaar heel precies uitgelegd wat er van de functies verwacht wordt. Baseer je op deze commentaar bij het implementeren van de functies. Je mag hier eventueel hulpfuncties aan toevoegen, maar het prototype van bestaande functies mag niet veranderd worden (m.a.w. de functie naam, return type, en argumenten mogen niet aangepast worden)!

`main.c` Bevat de `main` functie die een aantal eenvoudige testen uitvoert. Merk op dat deze testen niet volledig zijn! Het kan zijn dat deze testen lukken, maar er nog altijd een fout in je code zit. We raden je dan ook sterk aan om ook eigen testen te schrijven.

Je kan deze bestanden compileren het volgende commando:

```
gcc -g -std=c99 -Wall main.c list.c list.h -o main
```

Daarna kan je het programma starten met "`./main`". Om te vermijden dat je dit commando elke keer opnieuw moet typen, hebben we een Makefile bestand toegevoegd. Concreet betekent dit dat je in plaats van het bovenstaande gcc commando, gewoon "`make main`" kan uitvoeren. Dit zal dan het volledige gcc commando uitvoeren indien een van de bestanden is aangepast.

De rest van deze sectie zal dieper ingaan op een aantal aspecten van de opgave.

3.1 Gelinkte Lijsten

Deze sectie geeft een kort overzicht van de operaties die geïmplementeerd moeten worden. Zie het bestand `list.c` voor een gedetailleerdere beschrijving van elke functie.

`list_remove` Verwijder de node op de gegeven index.

`list_pop` Geef het eerste getal in de lijst terug, en verwijder dit uit de lijst.

`list_prepend` Voeg een getal vooraan de lijst toe.

`list_insert` Voeg een getal op een gegeven index toe.

Om jullie op weg te helpen hebben we al een aantal functies geïmplementeerd. Alle indices zijn nul gebaseerd, dus het eerste element heeft index 0.

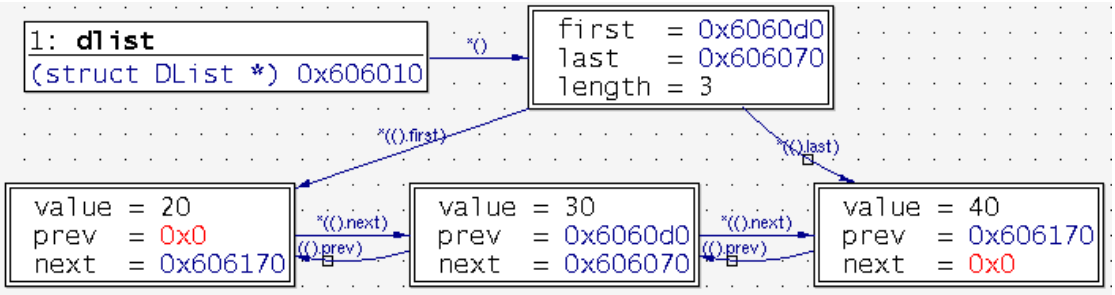
3.2 Dubbel Gelinkte Lijsten

Het nadeel van enkele gelinkte lijsten is dat we de lijst slechts in één richting kunnen doorlopen. Om de lijst in omgekeerde volgorde te doorlopen kunnen we, naast de `next` pointer, ook een `prev` pointer bijhouden in elke node. Verder zijn er ook nog de volgende twee nadelen aan onze huidige implementatie:

- Om de lengte van de lijst te berekenen moeten we heel de lijst doorlopen.
- Om een element achteraan toe te voegen moeten we heel de lijst doorlopen.

We kunnen de lengte van de lijst in een extra variabele bijhouden en deze updaten als we een element toevoegen of verwijderen. Daarnaast kunnen we niet alleen de pointer naar het eerste element bijhouden, maar ook een pointer naar het laatste element. Uiteindelijk kunnen we dus de volgende datastructuur gebruiken om de bovenvermelde problemen te vermijden:

```
1  struct DListNode {
2          int value;
3          struct DListNode *prev;
4          struct DListNode *next;
5  };
6
7  struct DList {
8          struct DListNode *first;
9          struct DListNode *last;
10         int length;
11 };
```



Figuur 2: Weergave van een dubbel gelinkte lijst in `ddd`.

In Figuur 2 is een voorbeeld gegeven van een dubbel gelinkte lijst die deze datastructuur gebruikt. Jullie moeten de volgende operaties op deze dubbel gelinkte lijst implementeren:

`dlist_print_reverse` Print de lijst in omgekeerde volgorde uit.

`dlist_length` Geeft de lengte van de lijst terug.

`dlist_get` Geeft het getal om de gegeven positie terug indien dit bestaat.

`dlist_append` Voeg een getal achteraan de lijst toe.

`dlist_remove` Verwijder de node op de gegeven index.

In de file `list.c` staat een gedetailleerde beschrijving van elke functie die jullie moeten implementeren. Opnieuw zijn al enkele functies gegeven. Bestudeer eerst hoe deze gegeven functies werken, en implementeer dan pas de overige operaties!

3.3 Stacks

Een stack (of stapel) is een simpele, veelgebruikte datastructuur. Er zijn twee essentiële operaties:

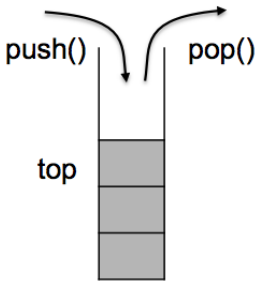
1. `push`: Plaatst een nieuw element op de stack
2. `pop`: Haalt, indien mogelijk, het laatst gepushte element van de stack

Een stack heeft dus de eigenschap dat het laatste element als eerste opnieuw wordt teruggegeven (zie Figuur 3). Dit wordt ook wel een LIFO-datastructuur genoemd (Last-In-First-Out).

Om een stack te implementeren gebruiken we de volgende datastructuur:

```
1 struct StackNode {
2         char *string;
3         struct StackNode *next;
4 };
5
6 struct Stack {
7         struct StackNode *top;
8 };
```

De implementatie van een stack lijkt dus sterk op de implementatie van een gelinkte lijst. Merk op dat bij de stack die je moet implementeren de elementen geen getallen zijn, maar strings. Geheugen voor deze strings moet gealloceerd worden met `malloc`. Het grootste verschil tussen lists en stacks dat we nu andere operaties ondersteunen:



Figuur 3: Een stack is een simpele datastructuur met het `push`'en en `pop`'en van elementen als de belangrijkste operaties.

Bestandspad	Vereenvoudigd bestandspad
`/home/user`	`/home/user`
`/home/user/..`	`/home`
`/home/user/../root`	`/home/root`
`/somedir/../home/user`	`/home/user`
`/../home/user`	`NULL`
`/home/../../somedir`	`NULL`

Tabel 1: Voorbeelden van originele en vereenvoudigde bestandspaden. Er wordt NULL teruggegeven indien het originele bestandspad ongeldig was.

`stack_create` Maak een lege stack aan.

`stack_push` Plaats een nieuwe string op de stack.

`stack_pop` Haal een string van de stack.

`stack_isempty` Test of de stack leeg is.

`stack_delete` Verwijder de stack.

`stack_print` Print de inhoud van de stack.

`stack_reverse` Draai de volgorde van alle elementen in de stack om.

`stack_join` Voeg alle strings in de stack samen, gescheiden door het gegeven scheidingsteken.

Zorg ervoor dat de operaties `stack_push` en `stack_pop` zo efficiënt mogelijk zijn. De uitvoeringstijd voor deze operaties moet onafhankelijk zijn van het huidige aantal elementen op de stack.

Eén van de toepassingen van een stack is het vereenvoudigen van een bestandspad. Bijvoorbeeld, het bestandspad `"/home/user1/.."` kan vereenvoudigd worden naar `"/home"`. Dus bij het vereenvoudigd bestandspad worden alle voorkomens van `".."` weggewerkt. We geven enkele complexere voorbeelden in Tabel 1.

Met behulp van een stack is het eenvoudig om bestandspad te vereenvoudigen. Het algoritme hiervoor werkt als volgt:

1. Splits het bestandspad (gegeven als een string) op in de verschillende directory-namen (deze worden gescheiden met een “/”), en initialiseer een lege stack.

2. Voor elke directory-naam:

   (a) Indien de directory-naam gelijk is aan "..", dan pop je de vorige directory-naam van de stack. Indien de stack leeg is, is het originele bestandspad ongeldig en geeft je NULL terug.

   (b) Anders zet je de directory-naam op de stack.

3. Eens het volledige bestandspad is verwerkt, kan je alle directory-namen in de stack combineren. Het resultaat is dan het vereenvoudigde bestandspad.

Jullie moeten dit algoritme in C implementeren. Meer bepaald moet je de functie:

```
char * realpath(const char *path)
```

in de file `list.c` implementeren. Het argument `path` is het bestandspad wat vereenvoudigd moet worden. De functie geeft het vereenvoudigde bestandspad als returnwaarde terug. De functie geeft `NULL` terug indien `path` geen geldige bestandspad is (zie de voorbeelden in Tabel 1). Gebruik de stack operaties die je in dit practicum hebt geïmplementeerd. Je mag ook je (dubbel) gelinkte lijst(en) gebruiken indien je dat nodig acht.

4 Tips

Enkele handige tips:

- Als een functie het eerste of laatste element in de lijst aanpast, denk er dan aan om ook de `first`/`last` pointer te updaten.
- Denk eraan om dynamisch geheugen, verkregen met behulp van de functie `malloc`, éénmalig vrij te geven met behulp van de `free` functie. Zo vermijd je memory leaks in je code.
- Wanneer je geheugen voor een string alloceert, denk er dan aan dat je ook plaats moet voorzien voor de terminating `'\0'`-char.
- Gebruik `gdb` of `ddd` om je implementatie te debuggen. Je kan ook `printf` statements toevoegen tijdens het schrijven van je code (verwijder deze wel in je ingestuurde code).
- **Test je functies in detail.** De meegeleverde testen zijn niet volledig. Schrijf dus ook je eigen testen en let vooral op randgevallen zoals, het eerste/laatste element verwijderen, een onbestaande index doorgeven, bewerkingen op de lege lijst, enz.

5 Afspraken

Je practicum moet **ten laatste op maandag 18 december 2017** ingeleverd worden. Dien op die dag voor 12:00 's middags een ZIP-bestand in met je code. Geef het ZIP-bestand een naam volgens deze conventie:

`iw_<VOORNAAM>_<NAAM>_<R-NUMMER>.zip`,

waar `<VOORNAAM>`, `<NAAM>`, en `<R-NUMMER>` uiteraard vervangen worden. In je ZIP-bestand moeten de bestanden `list.c`, `list.h`, en `main.c` zitten. Denk eraan dat jullie het bestand `list.h` niet mogen aanpassen! Het ZIP-bestand mag geen directories bevatten.

Je oplossing zal gecontroleerd worden via het uitvoeren via een aantal automatische testen. Zorg er daarom voor dat je oplossing werkt in de PC klassen van gebouw 200A. Je mag tijdens het oplossen van het practicum uiteraard een andere compiler gebruiken maar wat je indient *moet* werken met GCC; anders wordt het niet bekeken!

Voor vragen en problemen kan je altijd terecht op het forum voor dit practicum op Toledo. Dit kan je bereiken via `http://toledo.kuleuven.be`

Hou er ook rekening mee dat er een demo van dit practicum volgt waarbij je een aantal aanpassingen aan je code zal moeten aanbrengen. De datum wanneer deze demo zal doorgaan zal later bekend gemaakt worden.

Plagiaat (waaronder het delen van code met andere studenten) is uiteraard niet toegestaan tijdens dit practicum en zal behandeld worden als fraude.

Veel succes!