Practicum C: Gelinkte Lijsten

Informatica werktuigen

Academiejaar 2014-2015

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 Inleiding: Gelinkte Lijsten

Een *gelinkte lijst* is een datastructuur waar 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.

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.

2 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 `lists.zip` wat een aantal bestanden bevat om jullie op weg te helpen:

`list.h` Het header bestand waar alle declaraties (types en functies) die jullie moeten gebruiken in staan. 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.

2.1 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 laatste getal in de lijst terug, en verwijderd dit uit de lijst.

`list_prepend` Voeg een getal vooraan de lijst toe.

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

`list_insert_sorted` Voeg een getal toe aan een gesorteerde lijst, zodat de lijst nog altijd gesorteerd blijft.

`list_print_reverse` Print de lijst in omgekeerde volgorde uit.

`list_remove_all` Verwijder alle voorkomens van een bepaalde waarde.

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.

2.2 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 };
```

In Figuur 1 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.

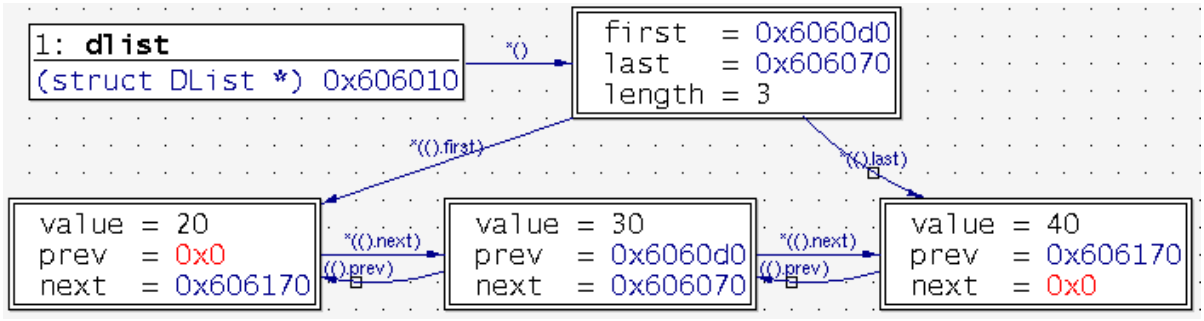
`dlist_length` Geeft de lengte van de lijst terug.

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

`dlist_append` Voeg een getal achteraan de lijst toe.

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

`dlist_remove` Verwijder de node op de gegeven index.



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

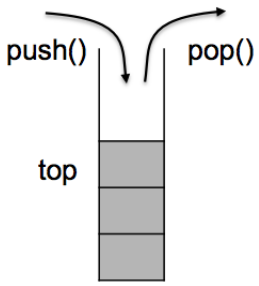
In de file `list.c` staat een gedetailleerde beschrijving van elke functie die jullie moeten implementeren. Opnieuw zijn al enkele functies gegeven. Bestudeer eerst hoe deze gegeven functies werken, en implementeer dan pas de overige operaties!

2.3 Stacks

Een stack (of stapel) is een simpele, veelgebruikte datastructuur. Er zijn slechts twee mogelijke 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 2). Dit wordt ook wel een LIFO-datastructuur genoemd (Last-In-First-Out).



Figuur 2: Een stack is een simpele datastructuur met het push'en en pop'en van elementen als de belangrijkste operaties.

Infix notatie	Postfix notatie
`101 + 44`	`101 44 +`
`(1 + 5) / 2`	`1 5 + 2 /`
`((1 - 2) - 5) + (15 / 5)`	`1 2 - 5 - 15 5 / +`
`5 / (2 - 2)`	`5 2 2 - /`
`-5 * 10`	`-5 10 *`
`70 / -7`	`70 -7 /`

Tabel 1: Voorbeelden van postfix expressies en hun overeenkomstige infix expressies.

Om een stack te implementeren gebruiken we de volgende datastructuur:

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

De implementatie van een stack lijk dus sterk op de implementatie van een gelinkte lijst. Het grootste verschil is dat we nu slechts enkele specifieke operaties moeten ondersteunen:

`stack_create` Maak een lege stack aan.

`stack_push` Plaats een nieuw element op de stack.

`stack_pop` Haal een element van de stack.

`stack_isempty` Test als de stack leeg is.

`stack_delete` Verwijder de stack.

Zorg ervoor dat de operaties `stack_push` en `stack_pop` zo efficiënt mogelijk zijn. Deze operaties moeten in constante tijd uitgevoerd kunnen worden. Aangezien er weinig verschil is tussen het implementeren van een stack en een gelinkte lijst, raden we je aan om eerst alle operaties van de gelinkte lijst te implementeren. Daarna kun je je op die code baseren, en zou het implementeren van de operaties vrij eenvoudig moeten zijn.

Eén van de toepassingen van een stack is het evalueren van een expressie in *postfix* notatie. Deze notatie is een alternatief op de meer gebruikelijke *infix* notatie. Bijvoorbeeld, `3 + 4` is een expressie in infix notatie, en in postfix notatie wordt dit `3 4 +`. Dus bij postfix notatie worden alle argumenten voor de operator geschreven. Enkele complexere voorbeelden zijn in Tabel 1 gegeven. Merk op dat in postfix notatie haakjes niet meer nodig zijn om de volgorde van

operaties aan te duiden.

Met behulp van een stack is het eenvoudig op een expressie in postfix notatie te evalueren. Het algoritme hiervoor werkt als volgt:

1. Splits de expressie (gegeven als een string) op in woorden (d.i., getallen en operatoren), en initialiseer een lege stack.
2. Voor elk woord:
   (a) Indien dit woord een operator is, pop twee argumenten, voer de operatie uit, en push het resultaat op de stack. Indien er geen twee getallen op de stack staan, geef je een error terug.
   (b) Indien het woord geen operator is, interpreteer dit woord als een getal, en zet het getal op de stack. Gebruik de functie `atoi` om in C een `char*` string om te zetten naar een `int`.
3. Eens alle woorden zijn verwerkt, staat er maar één getal op de stack. Dit is het resultaat van de postfix expressie. Indien de stack leeg is, of er staan twee of meer getallen op de stack, was de expressie ongeldig.

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

```
int evaluate(char* formula, int* result)
```

in de file `code.c` implementeren. Het argument `formula` is die postfix expressie die geëvalueerd moet worden. De functie geeft 1 terug indien het een geldige postfix expressie is, en 0 indien het een ongeldige expressie is. Het resultaat van de expressie wordt met behulp van de pointer `result` terug gegeven. 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.

3 Tips

Enkele handige tips:

- Als een functie het eerste/laatste element in de lijst aanpast, denk er dan aan om ook de `first`/`last` pointer te updaten.
- Denk eraan om geheugen vrij te geven zodat er geen memory leaks in je code voorkomen.
- **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.

4 Afspraken

Je practicum moet **ten laatste op maandag 15 december 2014** 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>_c.zip`,

waar `<VOORNAAM>` en `<NAAM>` 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 (GCC versie 4.8.2). 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!

Yolande Berbers
Jasper Bogaerts
Job Noorman
Thomas Winant
Raoul Strackx
Mathy Vanhoef
Wilfried Daniels