6 Gelinkte lijsten en manipulation

Oefening 36

```
typedef struct Knoop Knoop;
struct Knoop{
    int d;
    Knoop * opv;
};

int main(){
    Knoop * l = 0;
    Knoop * k = l;
    k = (Knoop*) malloc(sizeof(Knoop));
    return 0;
}
```

Teken de twee pointers `l` en `k`, en wat er precies gebeurt in bovenstaande code. Is `l` gewijzigd?

Oefening 37

```
typedef struct Knoop Knoop; /* zie vorige opgave */
int main(){
    Knoop * l = 0;
    voeg_vooraan_toe(7, ... );
    voeg_vooraan_toe(3, ... );
    print_lijst( ... );
    return 0;
}
```

Schrijf de nodige procedures, opdat bovenstaand hoofdprogramma zou werken. Op de puntjes geef je de gelinkte lijst `l` mee. Denk na over de manier waarop je dat doet.

1. De procedure `voeg_vooraan_toe` breidt de gegeven gelinkte lijst (tweede parameter) uit met één knoop: die komt vooraan, en bevat het gegeven getal (eerste parameter).
2. De procedure `print_lijst` implementeer je eerst niet-recursief.
3. Implementeer de procedure `print_lijst` daarna recursief.
4. Er ontbreekt een procedure aan het hoofdprogramma. Schrijf hoofding, implementatie en aanroep van deze procedure.

Oefening 38

```
int main(){
    srand(time(NULL));
    knoop * l = maak_gesorteerde_lijst_automatisch(10,100);
    print_lijst(l);
    printf("\nnu worden dubbels verwijderd: \n");
    verwijder_dubbels(...); /* aan te vullen */
    printf("\nna verwijderen van dubbels: \n\n");
    print_lijst(l);
    ... /* aan te vullen */
    return 0;
}
```

In bovenstaand hoofdprogramma wordt er automatisch een stijgend gesorteerde lijst aangemaakt, waar dubbele elementen in kunnen zitten. Implementeer de functie `maak_gesorteerde_lijst_automatisch(aantal,bovengrens)`. De eerste parameter geeft aan hoeveel elementen de lijst moet bevatten. De tweede parameter geeft aan wat het grootste getal zal zijn. Een tip: bouw de lijst op door vooraan telkens een iets kleiner getal dan het vorige toe te voegen. Daarvoor genereer je een getal uit de verzameling $\{0, 1, 2\}$ met de functie `rand()` uit `stdlib.h` en trek je dit getal af van het vorige toegevoegde getal (of van het getal `bovengrens` bij het toeveogen van de eerste knoop).

Schrijf de procedure `verwijder_dubbels(...)` die alle dubbels uit de gelinkte lijst verwijdert. Als enige parameter geef je de gelinkte lijst mee. Vul het hoofdprogramma verder aan zoals het hoort.

Oefening 39

Werk voort op voorgaande oefening. Het hoofprogramma wordt nu:

```
int main(){
    srand(time(NULL));
    knoop * m = maak_gesorteerde_lijst_automatisch(10,1000);
    knoop * n = maak_gesorteerde_lijst_automatisch(5,1000);
    printf("\nLIJST m:\n");    print_lijst(m);
    printf("\nLIJST n:\n");    print_lijst(n);
    printf("\nDeze worden gemerged. \n\n");

    knoop * mn = merge(...,...);

    printf("\nLIJST m:   \n"); print_lijst(m);
    printf("\nLIJST n:   \n"); print_lijst(n);
    printf("\nRESULTAAT: \n"); print_lijst(mn);
    ....   /* aan te vullen */
    return 0;
}
```

Schrijf de functie `merge` die twee gerangschikte lijsten als parameter neemt. Beide lijsten zullen op het einde leeg zijn; hun knopen zitten allemaal in de nieuwe (eveneens gerangschikte) lijst die teruggegeven wordt. Omdat de lijsten die je meegeeft gerangschikt moeten zijn (precondities van de functie), kan je de lijsten efficiënt samenvoegen. Neem hiervoor onderstaande code, die hetzelfde doet voor twee array's `a` en `b` als voorbeeld.

```
int * merge(const int * a, const int * b, int size_a, int size_b){
    int i = 0; /* indexeert a */
    int j = 0; /* indexeert b */
    int k = 0; /* indexeert c */
    int size_c = size_a + size_b;
    int * c = (int*)malloc(size_c * sizeof(int));

    while(i < size_a && j < size_b){
        if( a[i] < b[j] ){
            c[k++] = a[i++];
       }
       else{
            c[k++] = b[j++];
       }
    }

    while(i < size_a){
        c[k++] = a[i++];
    }
```

```
    while(j < size_b){
       c[k++] = b[j++];
    }

    return c;
}
```

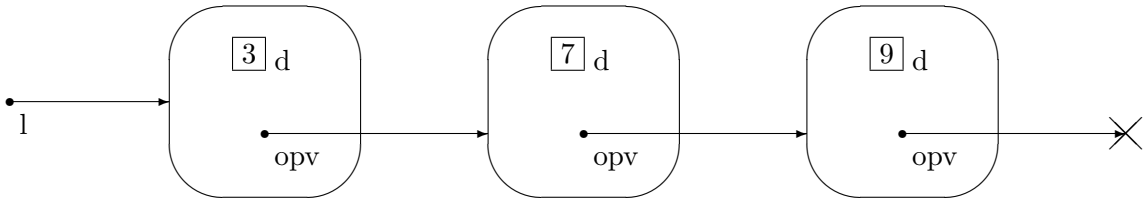
Oefening 40

Hieronder de procedure `voeg_getal_toe` die een knoop toevoegt aan een lijst die stijgend geordend is en moet blijven.

```
void voeg_getal_toe(int g, knoop **l) {
   knoop *h, *m;
   if (*l == 0 || g <= (*l)->getal) {
      h = (knoop*) malloc(sizeof(knoop));
      h->getal = g; h->next = *l; *l = h;
   }
   else {
      h = *l;
      while (h->next != 0 && h->next->getal < g)
         h = h->next;
      m = h->next;
      h->next = (knoop*) malloc(sizeof(knoop));
      h = h->next; h->getal = g; h->next = m;
   }
}
```

Je ziet dat er een opsplitsing nodig is in deelgevallen: een stuk code dat van toepassing is indien het toevoegen vooraan in de lijst gebeurt, en een stuk code dat alle andere gevallen behandelt. Dit dubbele werk willen we vermijden. Het is immers werk gespaard als je code kan schrijven die in elke situatie bruikbaar is.

De hulppointer `h` werd gedeclareerd als `knoop *` en duidt de knoop aan waar de nieuwe knoop aangehangen zal worden. Op tekening:



Als we het getal 4 willen toevoegen, zal `h` naar de eerste knoop wijzen. Als we het getal 10 toevoegen, zal `h` naar de laatste knoop wijzen. Als we het getal 2 willen toevoegen, zal `h` geen knoop hebben om naar te wijzen. (Vandaar de opsplitsing in gevallen in de code.)

Het valt echter op te merken dat je enkel `h->opv` nodig hebt in de code. De inhoud van de knoop waar `h` naar wijst (`h->d`) heb je niet nodig. Waarom dan naar de hele knoop (zowel `h->d` als `h->opv`) wijzen? Is toegang tot `h->opv` niet voldoende? Dat is de oplossing voor ons probleem: we laten een hulppointer `k` wijzen naar de horizontale pijlen (de links tussen de knopen). Willen we 4 toevoegen, dan wijst `k` naar de tweede horizontale pijl. Willen we 10 toevoegen, dan wijst `k` naar de laatste horizontale pijl. Willen we 2 toevoegen, dan wijst `k` naar de eerste horizontale pijl. En het behandelen van een speciaal geval is niet meer nodig!

Herschrijf de procedure `voeg_getal_toe(getal,lijst)` die een knoop met inhoud `getal` toevoegt aan een lijst die ondersteld wordt stijgend geordend te zijn (en te blijven). Dubbels zijn toegelaten. Gebruik géén `if-else` -structuur.

Oefening 41

Werk je sinds oefening 39 in een nieuw bestand? Heb je dan gedacht aan opkuisen van de lijst? Schrijf nu, als je dat nog niet deed, een *recursieve* versie van de procedure die een lijst volledig vrijgeeft en test uit. (Schrijf net voor het vrijgeven van een knoop, diens inhoud uit.)

Oefening 42

Schrijf een procedure `verwijder(x,lijst)` die — indien aanwezig — de knoop met inhoud `x` uit de stijgend gerangschikte lijst `lijst` verwijdert.

Oefening 43

Om een idee te hebben van wat er zoal gevraagd kan worden op de test van C, volgt hieronder een ongecensureerde testvraag. Oplossingen worden niet gepubliceerd. Uiteraard heeft het geen zin aan deze oefening te beginnen als je alle vorige niet onder de knie hebt. Veel puzzelplezier!

Gegeven een tekst zoals

```
De Dit Er zon is was schijnt een eens door goede een het zin. prinses. raam. STOP
```

Als je deze zin woord voor woord inleest en elk woord beurtelings in één van drie gelinkte lijsten stopt, krijg je volgende drie gelinkte lijsten:

```
De zon schijnt door het raam.
Dit is een goede zin.
Er was eens een prinses.
```

Schrijf een functie `geef_array_van_lijsten(int aantal)` die een opsomming van woorden (afgesloten met STOP) inleest en beurtelings in één van '`aantal`' gelinkte lijsten stopt. De array (met lengte '`aantal`') die deze lijsten bevat, wordt teruggegeven. De woorden worden ingelezen vanop het scherm en zullen niet meer dan 80 letters bevatten, maar neem niet meer geheugenplaats in dan nodig.

Let op: om netjes bij te houden waar je in de gelinkte lijsten gebleven bent, houd je een array bij van precies '`aantal`' hulppointers, die ervoor zorgen dat je elke gelinkte lijst achteraan kan uitbreiden met een nieuwe knoop, zonder telkens de reeds opgebouwde lijsten van voor af aan te moeten doorlopen.

Oefening 44

Schrijf onderstaande functies en procedures, waarbij je enkel gebruik maakt van bitoperatoren. Gebruik een `typedef` zodat je `unsigned int` korter kan noteren als `uint`.

1. De functie `bit_i(uint x, int i)` geeft het gehele getal 0 of 1 weer (type `int`), naargelang de inhoud van bit `i` van het getal `x`. Merk op: we zullen de laatste, minst significante bit, met het volgnummer 0 aanduiden. Test uit met enkele getallen.

2. Schrijf een procedure `schrijf(uint x, int lengte)` die gebruik maakt van bitoperatoren om het getal `x` uit te schrijven. Bij dit uitschrijven zullen er slechts `lengte` bits uitgeschreven worden (de minst significante). Je mag hierbij gebruik maken van de voorgaande functie `bit_i`. **Let op:** maak groepjes van 4 door tussen 4 bits steeds een statie toe te voegen. Test uit.
3. Schrijf de functie `eenbit(int i)` die de unsigned int teruggeeft die slechts één bit heeft aanstaan, nl. die met volgnummer `i`.
4. Schrijf de functie `aantal_eenbits(uint x)` die het aantal bits teruggeeft dat aanstaat in de binaire voorstelling van `x`. Werk niet via de functie `bit_i(x,i)`, dat is inefficiënt.
5. Schrijf de functie `bit_i_aangezet(uint x, int i)` die de unsigned int teruggeeft die gelijk is aan het gegeven getal x op hoogstens één bit na: de bit met volgnummer i staat aan.
6. Schrijf de functie `bit_i_uitgezet(uint x, int i)` die analoog werkt aan bovenstaande functie. Nu staat de i-de bit van het resultaat zeker uit.
7. Schrijf de functie `bit_i_gewisseld(uint x, int i)` die analoog werkt aan bovenstaande functie. Nu wordt de i-de bit veranderd van aan naar uit (of omgekeerd).
8. Schrijf de logische functie `zijn_gelijk(uint x, uint y)` die nul teruggeeft indien de parameters gelijk zijn. De `==`-operator mag niet gebruikt worden.
9. Schrijf de logische functie `is_even(uint x)` die 1 teruggeeft als het getal even is, anders 0. Gebruik enkel bitoperatoren, zelfs geen minteken.
10. Welke test wordt er uitgevoerd door de code `n&(n-1)==0` ?
11. Schrijf een functie `product(int a, int b)` die het product van `a` en `b` bepaalt zonder gebruik te maken van de vermenigvuldigingsoperator `*`. Tip: denk aan cijferrekenen (techniek uit lagere school).