Tips till Lab 2.1

Du kan göra lab 2.1 på lite olika sätt som du kan se i föreläsning 2. Du kan t.ex välja att göra en enkellänkad eller dubbellänkad, du kan också representera listhantaget på olika sätt. I detta dokument visar vi lite olika sätt du kan göra på, samt för- och nackdelar med dessa.

Att representera en listnod

Först behöver ni ha en typ för er listnod. Hur den ser ut beror på om ni vill göra en enkellänkad eller dubbellänkad lista.

Exempel på enkellänkad:

```
struct node
{
    Data data;
    struct node* next;
};
```

Vill ni göra en dubbellänkad lista blir det istället:

```
struct node
{
    Data data;
    struct node* next;
    struct node* previous;
};
```

Detta är alltså enbart en nod i listan, inte typen ni använder för själva listan.
En dubbellänkad lista är lite svårare att implementera eftersom varje operation måste ändra på både `next` och `previous`, men kan förenkla vissa operationer som t.ex att ta bort ett element i mitten.

Båda dessa representationer finns med i labbskelettet, välj en att kommentera bort.

Att representera en Lista

Själva listan kan också representeras på lite olika sätt och varje sett har sina för- och nackdelar.

1. Nodpekare

Det enklaste (och det som jag visat på föreläsningarna) är att representera en lista som en *nodpekare*.

Väljer ni att representera listan som en nodpekare så kommenterar ni bort följande typedef i labbskelettet:

```
typedef struct node* List;
```

Detta betyder att när man skriver typen `List` så använder man en nodpekare. En tom lista (utan noder) representeras då med hjälp av en `NULL`-pekare. En pekare till en lista blir då i praktiken en dubbelpekare till en nod.

Fördelar: denna representation är enkel och du kan mycket enkelt göra rekursiva funktioner med den.
Nackdelar: du behöver ofta hantera dubbelpekare (och är ett bra sätt att öva på dubbelpekare samt rekursion).

2. Struct

Ett annat sätt att representera en lista är med hjälp av en struct. Structen kan se lite olika ut beroende på vilken typ av information man vill spara om listan.

Exempel 1:

```
struct list
{
	struct node* head;
	int length;
};
```

I denna variant kan man spara listhuvudet och t.ex längden på listan. Använder man det behöver man komma ihåg att uppdatera "`length`" när man tar bort och lägger till element.

En tom lista i det här exemplet är en lista där `head==NULL` och `length == 0`.

Exempel 2:

```
struct list
{
	struct node* head;
	struct node* tail;
};
```

I denna variant sparar man en pekare till första och sista elementet i listan för enkel (och snabb) åtkomst till både slutet och början av listan. Man måste då komma ihåg att uppdatera båda dessa så de hela tiden pekar på första resp. andra elementet.

En tom lista i det här exemplet är en lista där `head==NULL` och `tail==NULL`.

Båda dessa varianter finns att avkommentera i labbskelettet.

Tänk på att om du väljer denna representation så kommer du troligen att behöva skriva ganska många hjälpfunktioner eftersom du kommer att behöva göra “samma sak” med en lista som med en nod.

Fördelar: du slipper hantera dubbelpekare, din implementation kan därför bli något mindre pekarintensiv.
Nackdelar: du kommer behöva hålla reda på “två typer” (en för listan, en för noderna), din kod kommer i allmänhet bli längre, och du kommer behöva skriva mer kod. Om du vill skriva rekursiva funktioner kommer du att behöva skriva fler hjälpfunktioner.

3. Ett litet exempel

Som ett exempel kan vi visa hur funktionen `createEmptyList()` kommer att se ut med de olika representationerna.

Om vi använder nodrepresentationen (List är en Node*) så ser funktionen ut som följer:

```
List createEmptyList(void)
{
      return NULL; // En NULL-pekare är en tom lista
}
```

Om vi använder structrepresentationen (`List` är en **`struct`**) så ser det istället ut som följer:

```
List createEmptyList(void)
{
      List list;  // Notera att list *inte* behöver allokeras dynamiskt!
      List.head = NULL;
      List.length = 0;
      return list; // Som du ser blir det lite mer att skriva
}
```

Som du också ser är det viktiga att din typ alltid heter just List för att testerna, testprogrammet och interfacet ska fungera.

Terminologi

I labben används termer som listnod, lista, element och data. Dessa kan vara röriga att hålla reda på så här ger jag en kort definition av dem:

Listnod

En listnod refererar till structen struct node. En nod kan aldrig vara tom. En nod innehåller alltid ett data och en eller två nodpekare (beroende på om listan är enkellänkad eller dubbellänkad). Noder är alltid dynamiskt allokerade och kan enbart kommas åt via pekare.

Lista

Listan refererar till er listrepresentation: en nodpekare eller en list-struct. En lista kan vara tom (inte peka på någon nod).

Element och/eller Data

När vi refererar till ett element så använder vi samma koncept som ett element i en array. Vi pratar om själva datat. Men när en funktion ska “lägga till ett element” så betyder det i praktiken att vi måste skapa en listnod som innehåller vårt data. Datat refererar till själva informationsdelen i en listnod.