Dubbellänkad list: Labb 1

En dubbellänkad lista är ett ganska tungt sätt att hålla reda på saker men har fördelen att det går fort att lägga till och ta bort saker, även mitt i listan. Dubbellänkade listor finns i de flesta programspråk.

C#: class LinkedList<T>
Java: class LinkedList<T>
C++: template < class T, class Alloc = allocator<T> > class list;

1 Uppgiften

Det som ska implementeras är inte en kopia av std::list utan vi gör en både enklare och annorlunda version – frånvaron av itteratorer i vår lösning förändrar helt hur interfacen ser ut.

Observera att den kloka studenten först implementerar alla datastrukturer med hjälp av en typedef för att när den fungerar göra om den till en template, se mera i avsnitt 2.

Ingående klasser:

```
template <class T>
class List;

template <class T >
class Node {
     friend class List<T>;
private:
     Node *prev, *next;
public:
     T data;
     Node * Prev();
     Node * Next();
     Node * InsertAfter(List<T> * list, T data);
     Node * InsertBefore(List<T> * list, T data);
     Node():prev(), next(){}
};

template <class T>
class List {
     friend class Node<T>;
private:
     Node<T> *last, *first;
public:
     Node<T> * push_front(T data);
     Node<T> * push_back(T data);
     T  pop_front();
     T  pop_back();
     Node<T> * First();
     Node<T> * Last();
     bool List::Check(int count);
     List():first(),last(){}
     ~List();
};
```

Laborationen är att implementera ovanstående klasser samt fundera på om det behövs några konstruktorer och destruktorer (ni behöver dock inte implementera dem denna gång).

Observera att det är tillåtet (och kanske nödvändigt) att ändra i klassdeklarationerna ovan. De är bara givna för att ge en idé om hur klasserna bör se ut.

2 Typdef implementation

Av flera orsaker så är det krångligt att skriva template kod, inte minst för att kompilatorn ger betydligt sämre felmedelanden då.

Ett vanligt sätt är därför att man börjar med att skriva koden för en bestämd typ, så det är lämpligt att göra en dubbellänkad lista som lagrar heltal först och testa den. När den fungerar så gör man om den till template.
I c++ så kan man ge typer nya namn ”alias” med hjälp av typdef.

```
typedef int T;
```

gör att vi i fortsättningen kan skriva T och kompilatorn kommer att byta ut det mot int.
Klassdeklarationerna blir då:

```
typedef float T;

class Node {
     friend class List;
private:
     Node *prev, *next;
public:
     T data;
     Node * Prev();
     Node * Next();
     Node * InsertAfter(List * list, T data);
     Node * InsertBefore(List * list, T data);
     Node():prev(), next(){}
};

class List {
     friend class Node;
private:
     Node *last, *first;
public:
     Node * push_front(T data);
     Node * push_back(T data);
     T  pop_front();
     T  pop_back();
     Node * First();
     Node * Last();
     bool List::Check(int count);
     List():first(),last(){}
     ~List();
};
```

3 Testning.

På alla laborationer kommer det att krävas att ni testar ert program. För flera av laborationerna så kommer det att finnas ett minimalt testprogram givet. Observera dock att era program skall fungera korrekt i alla situationer, att de går igenom testprogrammet är ingen garanti för att de blir godkända! I vissa fall kommer det också att finnas krav som är svåra att testa i ett testprogram (t.ex. att det inte allokeras extra minne i en viss metod). Vi kommer inte att kräva en uttömmande testning men ni bör ha en Code coverage på nära 100%.

3.1 Hur göra ett lämpligt testprogram?

Att göra en uttömmande testning är ofta svårt! Vi kan lägga till och ta bort saker på olika sätt och alla ska ge rätt resultat. Ett sätt att förenkla testningen är att skriva en funktion som kontrollerar att datastrukturen ser ut på rätt sätt. I den länkade listan kan man t.ex.

- Kontrollera att alla pekarna ska peka på rätt sätt.
- Att det finns rätt antal noder i listan.

Ett problem är att om listan får fel struktur och går i cirkel så finns det risk för att testprogrammet också loopar. Det kan enkelt lösas genom att vi skickar med förväntat antal noder listan och bryter om det visar sig vara fler noder.
Observera att testfunktioner av denna typ nästan alltid måste skrivas som en del av den testade klassen då de behöver tillgång till den ”privata” strukturen, i vårt fall prev, next, first och last pekarna.

Observera att den givna testmetoden antar saker om datastrukturen som inte är helt självklara. T.ex. att en tom lista har både ”first” och ”last” = null_ptr vilket inte är nödvändigt.

```
template <class T>
bool List<T>::Check(int count) {
     //count anger hur många noder som förväntas i strukturen.
     //true betyder att allt var ok
     if ((count==0) ^ (first==nullptr))
          return false; //tom lista ska ha first=null och tvärtom.
     if  (first==nullptr)
          return (last==nullptr && count==0);
     if ((last==nullptr) || count==0)
          return false;
     //nu är first och last != null och count!=0)
     Node <T> * node=first;
     Node <T> * lastNode=nullptr;
     while (node!=nullptr && count>0) {
          if (lastNode!=node->prev)
               return false;
          count --;
          lastNode=node;
          node = node -> next;
     }
     return (lastNode==last) && count==0;
}
```

3.2 Main program:

Detta är bara ett litet minimalt program:

```
#include <iostream>
#include <assert.h>
#include "DDList.h"

int main() {
     List<float> myList;
     assert(myList.Check(0));
     myList.push_back(3.4f);
     assert(myList.Check(1));
     myList.push_back(3.5f);
     assert(myList.Check(2));
     myList.pop_back();
     myList.pop_back();
     myList.pop_back();
     assert(myList.Check(0));

}
```

Ni bör också kolla efter minnesläckor (programmet ovan ska inte ge någon minnesläcka).

4 Versioner

140902: Preliminär
140909: Definitiv: ~List tillagd.