# 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. T.ex.

C#: class LinkedList<T>
C++: template<class T> class list;

Den främsta anledningen till att alls använda länkade listor är att det går fort att sätta in och ta bort element i listan - O(1). I arrays/vektorer tar det O(n) att sätta in något, man måste flytta på alla objekt som står senare i listan. Den vanliga lösningen är en så kallad extern lista där noderna innehåller pekare/referenser till data, t.ex. i C#

```
class Node<T>{
    Node *next, *prev;
    T refToData;      //en referens till data (i C++ skulle det varit en pekare).
…
}
```

I C++, std::list, så är det i stället en intern lagring:

```
template <class T>
class Node<T>{
    Node* next, prev;
    T data;
…
}
```

Vi kommer dock att göra på ett tredje sätt där vi förutsätter att själva vår data klass ärver från en Link klass så att data klassen självt har ärvt next och prev. Detta görs med CRTP (Curiously Recurring Template Pattern). En node klass ska därför ärva från Link med sig själv som parameter: `class Node : public Link<Node> {…}`

## 1 Exempel på möjlig klassdeklaration

```
template <class T>
class List;

template <class T>
class Link {
    Link* next; Link* prev;
    friend class List<T>;
public:
    Link();
    virtual ~Link() = default;
    T* Next();
    T* Prev();

    T* InsertAfter(T* TToInsert);
    T* InsertBefore(T* TToInsert);
    T* DeleteAfter();
    template<class Arg>
    T* FindNext(const Arg& searchFor);
    //FindNext use the function Match in the T class to see if it is a hit

    virtual std::ostream& Print(std::ostream& cout) { return cout; }
};
```

```
template <class T>
class List :public Link<T> {
public:
    List();
    T* First();
    T* Last();
    T* PushFront(T* item);
    T* PopFront();
    T* PushBack(T* item);
    template<class Arg>
    T* FindFirst(const Arg& searchFor) { return FindNext(searchFor); }

    friend std::ostream& operator<<(std::ostream& cout, List& list) {
        return list.Print(cout);
    }
    void Check();
private:
    std::ostream& Print(std::ostream& cout);
};
```

Ni ska sedan lägga till en klass `class Node :public Link<Node> …` som är noderna som verkligen ska läggas i listan.

## 1.1 Ett enkelt exempel på en datanode.

```
class Node: public Link<Node> {
        float val;
public:
        Node(float v): val(v) {}
        //Match is used in the FindNext funktionen
        bool Match(float rhs) { return val=rhs; }
}
```

## 1.2 Förklaringar och specificeringar

Listan äger sina noder, dvs. när listan tas bort ska också alla noder tas bort.
Idén med FindFirst och FindNext är att man ska kunna skriva en loop som går igenom alla objekt i listan och hittar det som stämmer ,ed argumentet t.ex.

```
for (Node* item = myListFindFirst(3.14f); item != nullptr; item = item.FindNext(3.14f) {
        cout << item.val; //This will not work, val is private
}
```

## 1.3 Interfacet är inte klart!

Det saknas bl.a const deklarationer, några metoder måste dessutom göras i två version, en med const och en utan.

### 1.4 Förklaring till interfacet.

Det finns ett litet testprogram som är en hjälp för att förstå hur listan kan användas, observera att detta program bara testar en del av funktionerna, att er lösningen klarar testprogrammet innebär inte alls att den är klar!

`bool List::Check();` This function verifies that the list is correct, it is only for testing purposes. For example, it can be implemented as

```
template<class T>  void List<T>::Check() {
    const Link<T>*node = this, *nextNode = next;
    do {
        assert(node->next == nextNode && nextNode->prev == node);
        node = nextNode;
        nextNode = nextNode->next;
    } while (node != this);
}
```

Men observera att det kan vara något i er implementation som gör att det fungerar annorlunda!

## 2 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! Ni kan också få problem därför att testprogrammet är felaktigt, ibland så förutsätter mina testprogram ett visst implementationssätt som inte krävs i labben. Detta innebär att ni mycket väl kan bli godkända även om testprogrammet inte fungerar.
Ganska ofta kan man underlätta testningrn genom att skriva in kontrollkod och asserts.
I denna labb skulle man t.ex. sist i varje metod i Link kunna skriva:

```
    assert(next->prev == this && prev->next == this);
```

så man ser att länkarna har blivit rätt.
Ni SKALL också kolla efter minnesläckor.