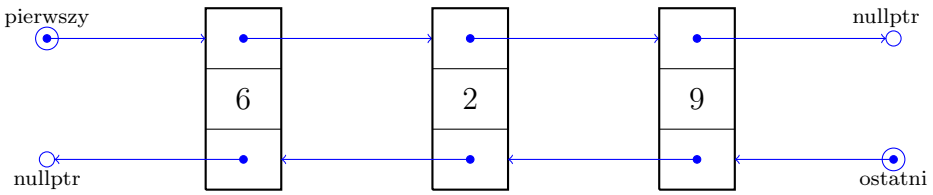
ZAAWANSOWANE PROGRAMOWANIE KOMPUTEROWE WNE (2022)

KRZYSZTOF ZIEMIAŃSKI

7. Listy dwukierunkowe

Listy jednokierunkowe są proste, ale mają pewne wady. Trudno jest znaleźć element poprzedzający dany element: wymaga to przeglądnięcia całej listy od początku. Podobnie jest w przypadku, gdy chcemy skasować pewien element; nie wystarczy nam posiadanie wskaźnika do tego elementu, potrzebujemy wskaźnika do elementu poprzedniego. Czasem wygodniej jest więc zapisywać w elemencie nie tylko wskaźnik do następnego, ale również do poprzedniego elementu; takie listy to listy dwukierunkowe.

Poniżej przykładowa lista dwukierunkowa.



Każdy element zawiera wartość, wskaźnik do następnego elementu (w prawo) oraz wskaźnik do poprzedniego elementu (w lewo). Jeśli następnego (lub poprzedniego) elementu nie ma, odpowiedni wskaźnik ma wartość `nullptr`.

7.1. **Implementacja.** Listy jednostronne na poprzednich zajęciach były zrealizowane po prostu jako wskaźniki do elementów (typ `Lista` był synonimem `ElListy*`). Implementacja listy dwustronnych będzie bardziej skomplikowana, ale też bardziej zbliżona to profesjonalnych implementacji. Używamy trzech klas:

- `ElListy`: pojedynczy element listy; będzie to klasa prywatna używana tylko przez pozostałe dwie.
- `ListaDw`: obiekt reprezentujący listę; przechowuje wskaźniki do pierwszego i ostatniego elementu listy.
- `Iterator`: obiekt służacy do przeglądania i modyfikacji list.

Oto definicja klasy `ElListy`:

```
class ElListy {
private:
    ElListy(ElListy* _nast, ElListy* _popr, int _dane)
        : nast(_nast), popr(_popr), dane(_dane) {}
    ElListy* nast;
    ElListy* popr;
    int dane;
```

```
    friend class ListaDw;
    friend class Iterator;
};
```

Proszę zauważyć, że wszystkie pola i konstruktor są prywatne, tj. nie można ich używać poza klasą z wyjątkiem klas `ListaDw` i `Iterator`, którym na to jawnie zezwalamy.

Następnie definiujemy klasę `Iterator`:

```
class Iterator {
public:
    int& operator*() { return p->dane; }
    void operator++() { p=p->nast; }
    void operator--() { p=p->popr; }
    operator bool() { return p!=nullptr; }
    int usun();
    void dodaj(int w);
private:
    Iterator(ElListy* _p, ListaDw* _lista): p(_p), lista(_lista) {}
    ElListy* p;
    ListaDw* lista;
    friend class ListaDw;
};
```

Każdy iterator wskazuje pewien element pewnej listy: wskaźnik `p` to ten element, a `lista` to wskaźnik do listy po której się poruszamy. Możliwe też jest, że iterator wskazuje na "nic" w swojej liście (wtedy `p==nullptr`). Na iteratorze `iter` dozwolone są następujące operacje:

- `*iter` to wartość wskazywana, można na nią coś przypisać,
- `++iter` przesuwa operator na następny element (jeśli byliśmy na ostatnim elemencie, to na `nullptr`),
- `--iter` przesuwa operator na poprzedni element (jeśli byliśmy na pierwszym elemencie, to na `nullptr`),
- `if(iter) { ... }` wykona się tylko, jeśli iterator na coś wskazuje (jest automatyczna konwersja z typu `Iterator` na typ `bool`),
- `iter.usun()` usuwa wskazywany element (iterator przestawia się na następny),
- `iter.dodaj(n)` wstawia nowy element z wartością `n` po wskazywanym elemencie.

Konstruktor jest prywatny, co oznacza, że iteratory może tworzyć tylko zaprzyjaźniona klasa `ListaDw`.

Na koniec klasa `ListaDw`:

```
class ListaDw {
public:
    ListaDw(): pierwszy(nullptr), ostatni(nullptr) {}
    ~ListaDw() { while(!pusta()) usunPierwszy(); }
    void dodajPierwszy(int w);
```

```
    int usunPierwszy();
    bool pusta() { return pierwszy==nullptr; }
    Iterator poczatek() { return Iterator(pierwszy, this); }
    Iterator koniec() { return Iterator(ostatni, this); }
private:
    ElListy* pierwszy;
    ElListy* ostatni;
    friend class Iterator;
};
```

Lista pozwala na wykonanie pewnych operacji (np. można dodawać i usuwać elementy na początku i sprawdzać, czy lista jest pusta), ale najważniejsza jest możliwość tworzenia iteratorów: `poczatek()` zwraca iterator wskazujący na początek listy, a `koniec()` na koniec listy.

Dzięki temu, że iterator posiada konwersję na `bool` łatwo przeglądać listę do przodu:

```
for(Iterator i=lista.poczatek(); i; ++i)
    cout << *i << "->";
```

lub do tyłu:

```
for(Iterator i=lista.koniec(); i; --i)
    cout << *i << "<-";
```

7.2. **Zadania.**

(1) Dodać do klasy `ListaDw` metodę

```
void dlugosc();
```

która zwraca liczbę elementów listy.

(2) Napisać funkcję

```
bool rosnaca(ListaDw& l);
```

która zwraca `true` tylko wtedy, gdy elementy na liście są w kolejności rosnącej.

(3) Napisać funkcję

```
bool takieSame(ListaDw& l, ListaDw& m);
```

która zwraca `true` wtedy i tylko wtedy, gdy obie listy mają takie same elementy w takiej samej kolejności.

(4) Napisać funkcję

```
void ostatniBedaPierwszymi(ListaDw& l);
```

która przestawia ostatni element listy na początek.

(5) Dodać do klasy `ListaDw` metodę

```
void doklej(ListaDw& m);
```

która dokleja listę `m` na końcu listy. Kolejność elementów powinna być zachowana. Po wykonaniu tej operacji lista `m` powinna być pusta, a jej elementy wykorzystane w liście wywołującej metodę `doklej`.

(6) Napisać funkcję

```
void podwoj(ListaDw& l);
```

która podwaja każdy element (np. lista `2->3->5->` zamienia się na `2->2->3->3->5->5->`).

(7) Dodać do klasy `ListaDw` konstruktor kopiujący

```
ListaDw(const ListaDw& m);
```

który tworzy kopię listy. Oczywiście należy stworzyć nowe elementy listy.