ZAAWANSOWANE PROGRAMOWANIE KOMPUTEROWE WNE (2022)

KRZYSZTOF ZIEMIAŃSKI

5. Listy jednokierunkowe

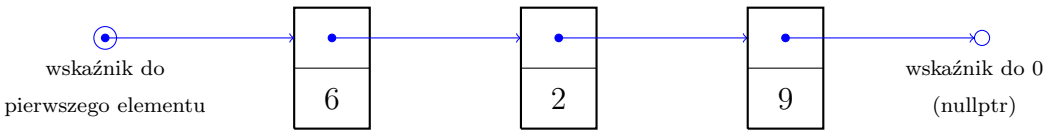
Listy to struktury danych umożliwiające przechowywanie elementów tego samego typu, przy czym kolejność tych elementów jest zachowana. Lista składa się z ciągu elementów; każdy element zawiera:

- *wartość*, tj. informację, którą chcemy przechować,
- *wskaźnik* wskazujący następny element listy.

Te informacje dotyczą najprostszych list, tzw. *list jednokierunkowych*; są też listy dwukierunkowe i cykliczne.

5.1. Listy jednokierunkowe — sposób działania.

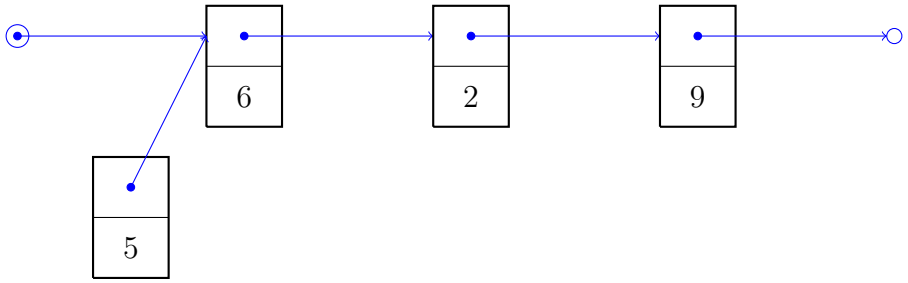
Lista przechowująca ciąg elementów $6, 2, 9$ wygląda tak:



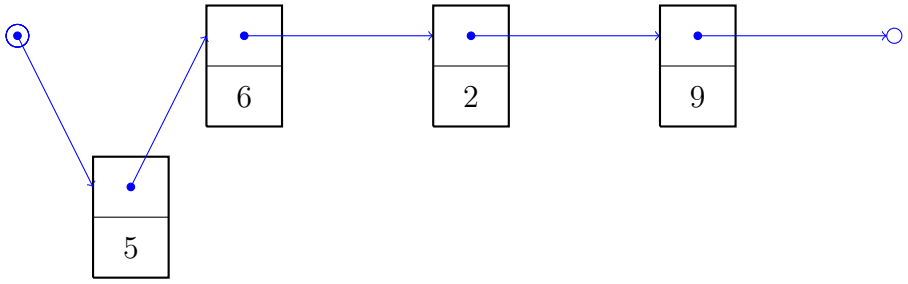
Możemy wykonywać na niej różne operacje, np. dodawać elementy (w różnych miejscach), usuwać elementy, przeglądać listę, itd. Kilka przykładowych operacji zilustrowanych jest poniżej.

Dodanie elementu na początku (o zadanej wartości, np. 5)

(1) Tworzymy nowy element z wartością 5, wskazujący na początek listy.

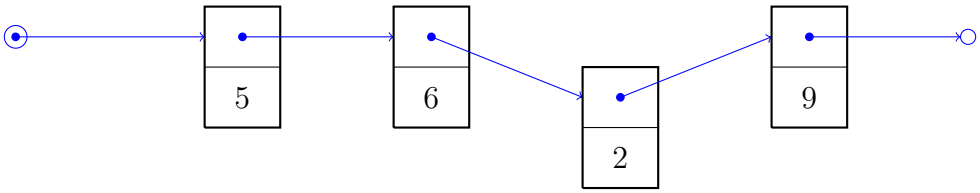


(2) Przepinamy wskaźnik do pierwszego elementu

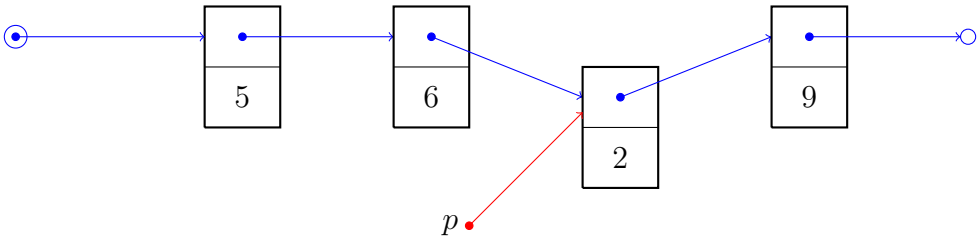


Uwaga: wskaźnik początku listy się zmienił!

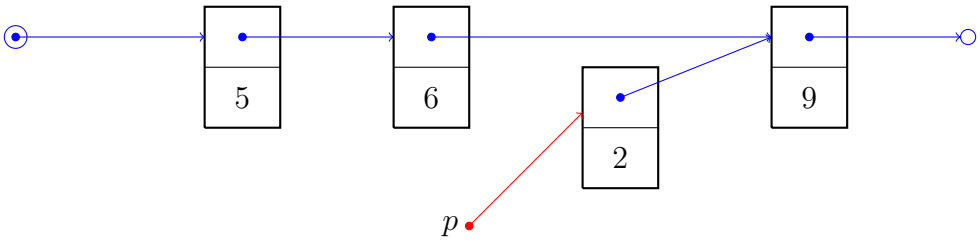
Usunięcie elementu. Na przykładzie usuwamy środkowy element (z wartością 2).



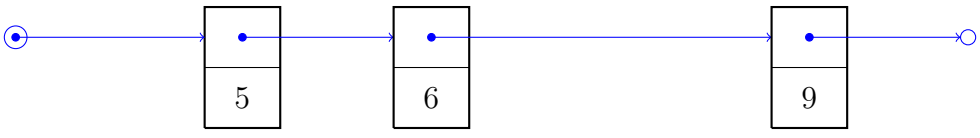
(1) Zapamiętujemy usuwany element przy pomocy zmiennej pomocniczej (wskaźnika).



(2) Przepinamy wskaźnik, tak, aby lista omijała usuwany element.



(3) Kasujemy element używając wskaźnika p.



5.2. **Jak tworzyć listy: wskaźniki.** Niebieskie strzałki na rysunkach powyżej to wskaźniki. Wskaźnik to zmienna, której wartością jest miejsce w pamięci gdzie przechowywana jest zmienna pewnego typu. Typem wskaźnika do typu `T` jest `T*`; np.

```
int* p;
```

jest deklaracją wskaźnika do typu `int`. Wskaźnik może przyjmować specjalną wartość `nullptr` (w starszych wersjach C++ jest to `NULL` lub po prostu `0`). Jeśli `p` jest wskaźnikiem, to wskazywaną przez niego wartość uzyskujemy przez `*p` (jest to tzw. *operator wyłuskania*); oczywiście jeśli `p==nullptr`, tzn. wskaźnik nie wskazuje na nic, próba "wyłuskania" wartości spowoduje błąd.

My będziemy potrzebować wskaźników do elementów listy. Definiujemy typ elementów listy:

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

Element listy ma dwa pola: `nast` jest wskaźnikiem do kolejnego elementu, a `dane` zawiera wartość. Możemy identyfikować listę i wskaźnik do jej pierwszego elementu. Dla wygody użyjemy definicji typu

```
typedef ElListy* Lista;
```

dzięki czemu będziemy mogli pisać `Lista` zamiast `ElListy*`: jest to bardziej czytelne. Teraz możemy stworzyć pustą listę za pomocą deklaracji

```
Lista l=nullptr;
```

Żeby dodać do listy elementy, musimy je najpierw stworzyć. Służy do tego operator `new`. Wywołanie `new T` tworzy (dynamicznie) nowy obiekt (zmienną) typu `T` i zwraca wskaźnik na nią wskazujący; proszę zauważyć, że do tego obiektu będziemy się mogli odwołać **tylko** za pomocą tego wskaźnika (musimy więc go koniecznie zapamiętać). Możliwe jest też podanie parametrów do konstruktora. Na przykład

```
Lista l=new ElListy(nullptr, 7);
```

tworzy nową jednoelementową listę z wartością 7 (oczywiście `Lista` to ten sam typ co `ElListy*`). Podobnie

```
Lista l;
// tu coś robimy z listą
l=new ElListy(l, 5);
```

dodaje nowy element (z wartością 5) na początku listy `l`.

5.3. **Odczyt elementów listy: operator `->`.** Jeśli `l` jest typu `Lista`, to

- `*l` jest pierwszym elementem listy,
- `(*l).dane` jest wartością zapisaną w pierwszym elemencie,
- `(*l).nast` jest wskaźnikiem do drugiego elementu listy
- `(*(*l).nast).dane` jest wartością zapisaną w drugim elemencie, itd.

Użycie nawiasów jest niezbędne, bo operator `.` (kropka) wiąże mocniej niż operator `*`. Zapis można uprościć przy pomocy operatora `->`: zapis `a->b` oznacza to samo, co `(*a).b`, czyli pole `b` obiektu wskazywanego przez wskaźnik `a`. Tak więc

- `l->dane` jest wartością zapisaną w pierwszym elemencie,
- `l->nast` jest wskaźnikiem do drugiego elementu listy
- `l->nast->dane` jest wartością zapisaną w drugim elemencie, itd.

5.4. **Przeglądanie listy.** Listy można przeglądać przy pomocy pętli `for`, np. fragment kodu powoduje wydrukowanie listy na ekranie.

```
Lista l;
// tu coś robimy z listą
for(ElListy* p=l; p!=nullptr; p=p->nast)
    cout << p->dane << "->";
```

Przypomina to przeglądanie tablic lub napisów. Inny sposób przeglądania list, oparty na rekurencji, jest użyty w pliku do zajęć.

5.5. **Kasowanie elementów.** Pierwszy element listy `l` można skasować pisząc po prostu

```
l=l->nast;
```

Pierwszy element "zniknie" z listy i nie będzie dalej widoczny. Nie jest to jednak dobra praktyka, bo pamięć zajmowana przez ten element będzie nadal zarezerowowana. Należy ją zwolnić przy pomocy operatora `delete`. Jeśli `p` jest wskaźnikiem, to instrukcja `delete p;` powoduje zniszczenie obiektu wskazywanego przez `p`. Poprawną metodą kasowania pierwszego elementu jest

```
Lista p=l->nast; // zapamiętujemy wskaźnik do pozostałej części listy
delete l;        // kasujemy pierwszy element
l=p;             // ustawiamy l na drugi element
```

5.6. **Listy: zalety i wady.** W porównaniu z tablicami, listy jednostronne mają zalety:

- Zarezerwowana pamięć jest zawsze proporcjonalna do liczby elementów na liście. W przypadku tablic potrzebujemy "zapasu" na nowe elementy a skasowanie elementu nie zwalnia pamięci (chyba, że chcemy przepisywać elementy do nowej tablicy, co jest kosztowne czasowo).
- Dodanie jednego elementu na początku wymaga stałej liczby operacji, niezależnej od rozmiaru listy. Wstawianie w innym miejscu również jest szybkie, po warunkiem, że wiemy gdzie wstawiać.

Niestety mają również wady — najpoważniejszą jest utrudniony dostęp do elementu o podanym indeksie. W dalszym ciągu kursu poznamy inne, bardziej skomplikowane struktury danych, które są tych wad pozbawione, przynajmniej w penym stopniu.

5.7. **Zadania.**

(1) Napisać funkcje

```
void dodajPierwszy(Lista& l, int a);
int usunPierwszy(Lista& l);
void dodajOstatni(Lista& l, int a);
int usunOstatni(Lista& l);
```

które dodają bądź usuwają elementy z listy. Funkcje usuwające elementy powinny zwrócić wartość zapisaną w usuwanym elemencie.

(2) Napisać funkcję

```
int dlugosc(Lista& l);
```

która zwraca długość listy `l`.

(3) Napisać funkcję

```
int indeks(Lista& l, int i);
```

która zwraca wartość przypisaną do `i`–tego elementu listy `l` (lub 0 jeśli takiego elementu nie ma).

(4) Napisać funkcję

```
int usunElementy(Lista& l, int w);
```

która usuwa wszystkie wystąpienia elementu z wartością `w` na liście `l`. Zwraca liczbę usuniętych elementów.

(5) Napisać funkcję

```
void sklej(Lista& l, Lista& m);
```

która dołącza listę `m` na końcu listy `l`. Nie należy tworzyć nowych elementów.

(6) Napisać funkcje

```
vector<int> stworzWektor(Lista& l);
Lista stworzListe(vector<int> v);
```

które zamieniają listę na wektor i na odwrót. Należy zachować kolejność elementów.

(7) Napisać funkcję

```
Lista tylkoDodatnie(Lista& l);
```

która zwraca nową listę zawierająca tylko dodatnie elementy z listy `l`. Zwracana lista powinna składać się z nowych elementów (nie powinna wykorzystywać elementów listy `l`). Kolejność elementów powinna być zachowana, należy też zadbać o to, aby złożoność była liniowa.

(8) Napisać funkcje

```
void dodajSort(Lista& l, int a);
void usunSort(Lista& l, int a);
```

które dodają lub usuwają element `a` tak, aby lista `l` zawsze pozostawała posortowana rosnąco.

(9) (*) Napisać funkcję

```
void sortuj(Lista& l);
```

która sortuje listę. Nie należy używać żadnych dodatkowych struktur (np. wektorów), ani tworzyć nowych elementów listy. Czy można uzyskać złożoność lepszą niż kwadratowa?