

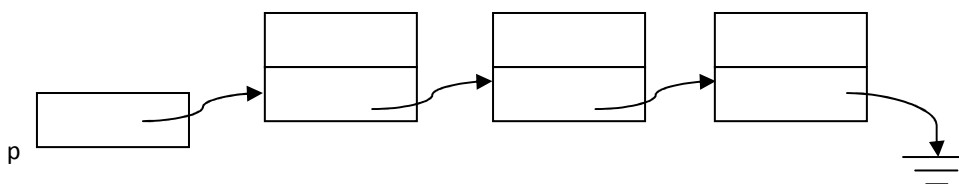
Zaczynając programować w dowolnym języku programowania jesteśmy zmuszeni do opanowania zasad posługiwania się podstawowymi typami danych. Na przykład w językach C/C++ posługujemy się m.in. typami: `int`, `float` czy `char` albo tablicami lub strukturami złożonymi z tych zmiennych. Zmienne zadeklarowane za ich pomocą nazywamy zmiennymi **statycznymi**. Ich zastosowanie łączy się z uzyskaniem wielu zalet, choćby spójność danych czy szybki dostęp do elementów. Posiadają jednak bardzo istotną wadę, mocno ograniczającą możliwość ich zastosowania: musimy z góry określić wielkość i ilość elementów. Jednakże wiele zagadnień charakteryzuje się zmiennością struktur danych podczas procesu obliczeniowego. W tym wypadku z pomocą przychodzą nam zmienne **dynamiczne**.

Niewątpliwą korzyścią wynikającą ze stosowania tych typów jest możliwość określenia wielkości zajmowanej przez nich pamięci dopiero w trakcie trwania programu. Taka deklaracja struktury danych powoduje jednak, iż kompilator nie jest w stanie określić konkretnych adresów dla poszczególnych zmiennych ani przydzielić na nie pamięci. W konsekwencji zmuszeni jesteśmy do **dynamicznego przydzielania pamięci**, tzn. przydzielania pamięci zmiennym w momencie użycia ich przez program (utworzenia nowej zmiennej w trakcie wykonania programu) zamiast przydzielania im stałej ilości pamięci podczas translacji programu. Kompilator przydziela wtedy stałą ilość pamięci na **adres** zmiennej umieszczanej dynamicznie w pamięci zamiast na samą zmienną. Zmienne, które zawierają adres i tym samym wskazują położenie innego obiektu lub zmiennej w pamięci, nazywamy **wskaźnikami**. W językach C/C++ zmienną wskaźnikową oznaczamy `*`.

Korzystając z dynamicznych struktur danych możemy między innymi operować na **listach**, **drzewach** czy **grafach**.

LISTY JEDNOKIERUNKOWE

Najprostszym sposobem pozwalającym grupować dowolną ilość danych (ograniczoną tylko rozmiarem dostępnej pamięci) jest lista jednokierunkowa (patrz rys. 1). W tym przypadku każdy element jest pojedynczym rekordem składającym się z co najmniej dwóch pól: pola wartości i wskaźnika do następnego elementu listy. Wartością wskaźnika umożliwiającą jednoznaczne zidentyfikowanie ostatniego elementu listy jest specjalna wartość **NULL**.



Rys. 1. Przykład listy jednokierunkowej

W przypadku list jednokierunkowych nie jest możliwy bezpośredni dostęp do dowolnego jej elementu. Ponieważ każdy element posiada jedynie wskaźnik do jego następnika, więc dotarcie do *n-tego* elementu listy wymaga wcześniej przejścia przez *n-1* poprzedników. W tym wypadku mówimy zatem o dostępie sekwencyjnym.

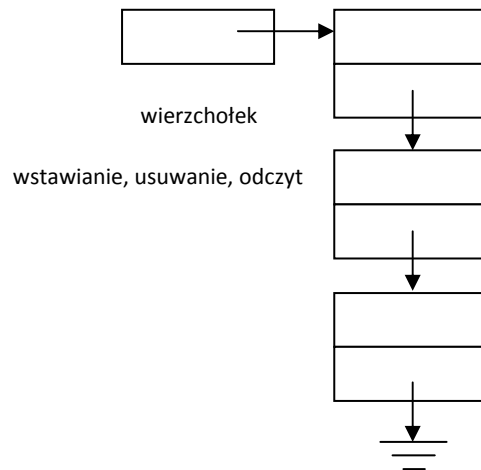
Oto najprostsze operacje, jakie możemy wykonać na liście jednokierunkowej:

- wstawienie elementu na początek listy,
- wstawienie elementu na koniec listy,

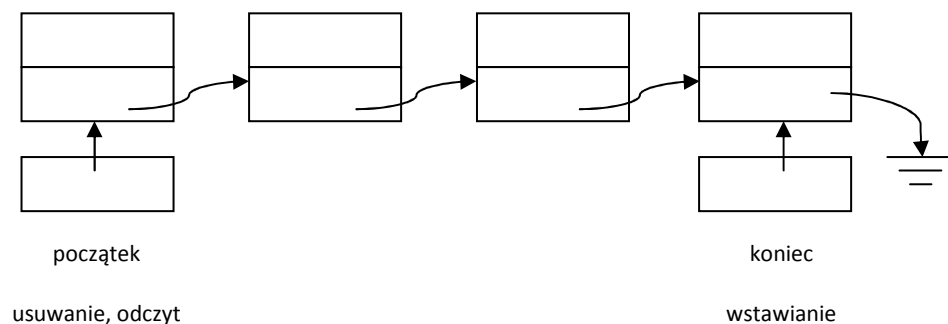
- usunięcie pierwszego elementu listy,
- odczyt pierwszego elementu listy,
- dostęp do dowolnego elementu listy.

W zależności od wyboru podstawowych operacji wykonywanych na liście możemy wyróżnić dwa podstawowe rodzaje struktur implementowanych na podstawie list jednokierunkowych:

- **stos**: możliwe jest wstawianie, usuwanie i odczyt pierwszego elementu listy - wierzchołek; tego typu struktura nazywana jest LIFO¹ (rys. 2),
- **kolejka** (pojedyncza, jednokierunkowa): wstawianie elementu na koniec listy, odczyt i usuwanie elementu pierwszego - struktura typu FIFO² (rys. 3).



Rys. 2. Struktura stosu



Rys. 3. Struktura kolejki

ZADANIE 1.

Napisać program odwracający plik tekstowy **plik.in**. Wynikiem wykonania programu ma być drugi plik tekstowy **plik.out**. Pierwszym wierszem pliku wyjściowego ma być ostatni wiersz pliku wejściowego, drugim - przedostatni itd. Zatożyć, że wiersze nie przekraczają 255 znaków³.

¹ ang. last in, first out

² ang. first in, first out

³ K.Jakubczyk: *Turbo Pascal i Borland C++. Przykłady.*, Helion, Gliwice 2002, s. 149

DEFINIOWANIE LISTY

Zdefiniujmy typ elementów listy wierszy pliku. Każdy taki element zawiera ciąg znaków pochodzący z kolejnej linii pliku tekstowego oraz wskaźnik do elementu następnego. Definicja ta będzie miała następujący wygląd:

```
struct LINIA          // 1
{
    char tekst[255];  // 2
    LINIA *nast;      // 3
} *lista;             // 4
```

Zaczynamy od zdefiniowania struktury **LINIA** (1). Definicja narzuca strukturę każdej zmiennej typu **LINIA** składającą się z dwóch pól: **tekst** (2) zawierającego linię tekstu pliku oraz **nast** (3) zawierającego wskaźnik do następnego elementu. * to symbol wskaźnika.

Wśród zmiennych globalnych zadeklarowaliśmy zmienną **lista** (4) typu **LINIA** wskazującą na początek listy wykorzystywanej w zadaniu.

DODAWANIE NOWEGO ELEMENTU NA POCZĄTEK LISTY

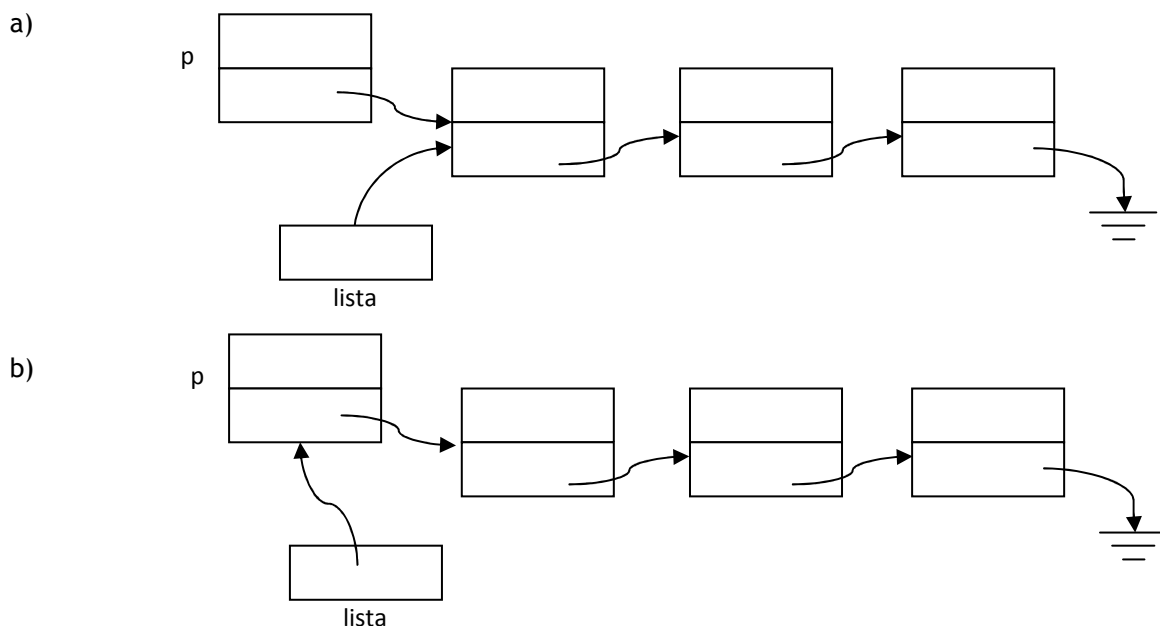
Wstawienie elementu **p** na początek listy **lista** wymaga wykonania dwóch operacji przypisania wartości zmiennym typu wskaźnikowego:

- nadanie polu **nast** wskazywanemu przez wskaźnik **p** wartości adresu przechowywanego przez wskaźnik do początku listy **lista** (rys. 4a);

```
p->nast = lista;
```

- uaktualnienie wartości zmiennej **lista** poprzez przypisanie jej wartości wskaźnika do elementu **p** (element **p** zostaje umieszczony na pierwszym miejscu listy **lista** - rys. 4b).

```
lista = p;
```



Rys. 4. Dodanie elementu **p** na początek listy **lista**

Wstawianie elementu *p* na początek listy *lista* możliwe jest wówczas, gdy element *p* istnieje (został wcześniej utworzony). W przeciwnym wypadku musimy utworzyć nowy element i uaktualnić jego pola (nadać im wartości początkowe). Operatorem przydzielającym pamięć oraz generującym wskaźnik do nowego elementu jest polecenie **new**.

```
p = new LINIA;
strcpy(p->tekst, s);
```

Procedura wczytującą kolejne linie tekstu i umieszczającą je na początku listy będzie zatem miała następujący wygląd:

```
void utworz()
{
    FILE *plik; // wskaźnik do pliku
    char s[255];
    LINIA *p; // wskaźnik pomocniczy
    lista = NULL; // początkowo lista jest pusta
    if ((plik=fopen("plik.in", "r"))==NULL) // próba otwarcia pliku
        printf ("Bład danych");
    else {
        while (fgets(s,255,plik)!=NULL) { // dopóki poprawne wczytanie
            // kolejnej linii tekstu
            p = new LINIA; // utwórz nowy element p
            strcpy(p->tekst,s); // skopiuj elementu tekst
            p->nast = lista; // wstaw element p
            lista = p; // na początek listy
        }
    }
    fclose(plik); //zamknij plik
}
```

Każda nowa wczytana linia tekstu zostaje wstawiona na początek listy. W wyniku tej operacji uzyskujemy strukturę listową zawierającą w pierwszym elemencie ostatnią linię tekstu znajdującego się w pliku, w drugim przedostatnią itp. Wystarczy teraz wypisać kolejne elementy listy do pliku **plik.out**, aby nasze zadanie zostało pomyślnie zakończone.

PRZEGLĄDANIE LISTY

Zmienna **lista** przechowuje adres pierwszego elementu listy w pamięci. Nie możemy modyfikować jej wartości, gdyż wiązałoby się to z utratą informacji na temat położenia pierwszego elementu, a co z tym się wiąże - również pozostałej części listy. Wprowadźmy więc zmienną pomocniczą **p** typu **LINIA** i z jej pomocą dokonajmy przeglądu elementów listy.

Przeглядanie listy będzie przebiegać w następujących etapach:

- a) zmienna **p** wskazuje na początek listy (rys. 5a);

```
LINIA *p=lista;
```

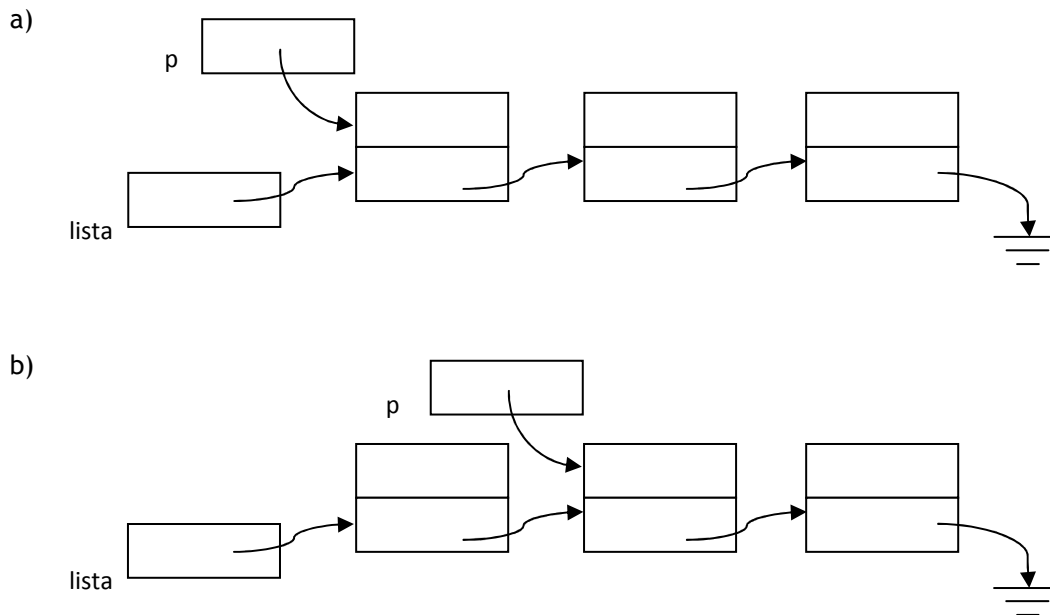
- b) wypisz pole **tekst** zmiennej wskazywanej przez **p**;

```
printf("%s", p->tekst);
```

- c) przejdź do kolejnego elementu (rys 5b);

```
p=p->nast;
```

- d) powtarzaj punkt b), dopóki nie osiągniesz końca listy (sygnalizowane przez wartość **NULL**).



Rys. 5. Przeglądanie listy jednokierunkowej

Zatem cała procedura wypisywania listy do pliku będzie wyglądać następująco:

```
void wypisz()
{
    FILE *plik; // wskaźnik do pliku
    LINIA *p; // wskaźnik pomocniczy
    plik=fopen("lista.out","w"); // otwarcie pliku do zapisu
    for (LINIA *p=lista;p;p=p->nast) // dopóki nie został osiągnięty
        // koniec listy
        fprintf(plik, "%s",p->tekst); // wypisz linię tekstu do pliku
    fclose(plik); // zamknij plik
}
```

USUWANIE LISTY

Dodając nowe elementy do listy zmuszeni byliśmy do dynamicznego przydzielania im pamięci przy każdorazowym utworzeniu nowego elementu za pomocą operatora **new**. Przed zakończeniem działania programu obowiązkiem programisty jest zwolnienie uprzednio przydzielonego obszaru pamięci. Do tego celu użyjemy operatora **delete**. Zauważmy, że nie wolno nam bezpośrednio usunąć początkowego elementu listy, wskazywanego przez **lista**, gdyż w takim wypadku utracilibyśmy wskaźniki do pozostałej części listy. Wykorzystamy zmienną pomocniczą **p** typu **LINIA**, aby bezpiecznie usunąć element z listy. Cała operacja będzie miała następujący przebieg:

- a) zmienna **p** wskazuje na początek listy (rys. 6a);

```
p = lista;
```

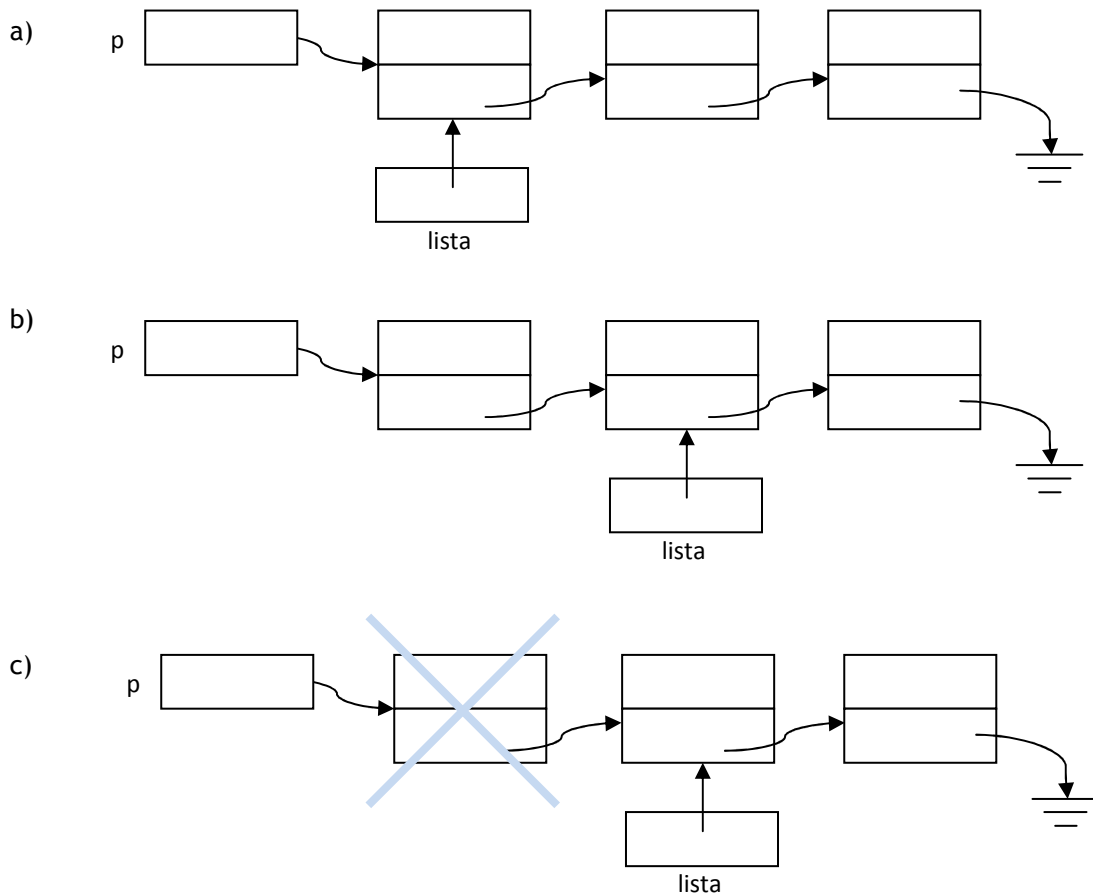
- b) **lista** wskazuje na element następny (rys 6b);

```
lista = lista->nast;
```

- c) usuń bezpiecznie element **p** (rys 6c);

```
delete p;
```

d) powtarzaj punkt a), dopóki nie osiągniesz końca listy (sygnalizowane przez wartość **NULL**).



Rys. 6. Usuwanie elementu z listy jednokierunkowej

Prześledźmy treść procedury usuwającą wszystkie elementy listy z pamięci:

```
void usun()  
{  
    LINIA *p; // wskaźnik pomocniczy  
    while (lista) { // dopóki nie został osiągnięty koniec listy  
        p = lista; // p wskazuje na początek listy  
        lista = lista->nast; // przejdź listą do następnego elementu  
        delete p; // usuń element  
    }  
}
```

TREŚĆ PROGRAMU

Główna część programu ma za zadanie wyłącznie wywołanie każdej z zadeklarowanych przez nas procedur:

```
int main()
{
    utworz();
    wypisz();
    usun();
    return 0;
}
```

Zauważmy, że zadeklarowanie pola typu `char tekst[255]` jest posunięciem dalece nieekonomicznym, gdyż każda linia tekstu zawiera różną ilość tekstu, przy tym niezwykle rzadko (o ile w ogóle) osiąga maksymalną długość. Można zatem rozważyć również dynamiczny przydział pamięci dla poszczególnych tablic znaków. Modyfikację pozostawiam czytelnikowi.