Rozdział 5.

Typy i struktury danych

Nikogo nie trzeba chyba przekonywać o wadze tematu, który zostanie poruszony w tym rozdziale. Od wyboru właściwej w danym momencie struktury danych może zależeć wszystko: szybkość działania programu, zakres obsługiwanych przypadków podstawowych, łatwość i możność modyfikacji, czytelność zapisu algorytmów, przenośność i... w konsekwencji dobre samopoczucie programisty.

Typy podstawowe i złożone

Każdy, kto poznał jakikolwiek język programowania, został niejako zmuszony do opanowania zasad posługiwania się tzw. typami podstawowymi (wbudowanymi).

Przykładowo w C++ dysponujemy m.in. następującymi typami:

- `int` i `long` (liczby całkowite, np. deklaracja `int zmienna_typu_int=5;` pozwala na przechowywanie w zmiennej `zmienna_typu_int` wartości całkowitych, tutaj przykładowo 5),
- `float` i `double` (liczby zmiennopozycyjne),
- `char` (znaki, np. `char znak='a'`),
- `bool` (przydatny do deklarowania zmiennych logicznych, przyjmujących wartości `true` i `false`),
- typy wskaźnikowe[1] (z „gwiazdką", służące do przechowywania adresów, np. deklaracja `int *zm;` oznacza, że `zm` będzie wskazywała na zmienną typu całkowitego. „Wskaźnik" przechowuje *adres*, aby „wyłuskać" z niego wskazywaną wartość, trzeba go poprzedzić symbolem gwiazdki (przykładowo `zm` zawiera adres, a `*zm` — wartość, oczywiście pod warunkiem, że zmienna `m` została wcześniej zainicjowana, np. poprzez instrukcję `zm=&zmienna_typu_int`). Operator `&` wyłuskuje ze zmiennej jej adres i stanowi, jak widać, przeciwieństwo operatora *.

Typy podstawowe stanowią niezbędne cegiełki, z których budujemy bardziej złożone struktury danych i warto je dobrze opanować, aby np. rozumieć, że * może oznaczać nie tylko mnożenie. Ponadto każdy typ podstawowy jednoznacznie determinuje zbiór dozwolonych wartości (zakres) oraz operacji (np. dodawanie, mnożenie itp.).

---

[1] Dodatek A omawia także tzw. referencje, które są nieco wygodniejsze w użyciu od wskaźników, choć nie do końca mogą je zastąpić w rzeczywistych programach.

W C++ niektóre typy danych mają charakter obiektowy (np. string, vector), co pozwala na łatwe manipulowanie ich zawartością (np. dodawanie obiektów) oraz izoluje nas od kwestii technicznych, takich jak np. przydzielanie i zwalnianie pamięci.


Ostrzeżenie

Lista typów podstawowych oraz ich precyzja jest w C++ określona przez standard tego języka, ale ich konkretna *implementacja* (np. liczba bitów zajmowanych przez zmienną określonego typu) — już od konkretnej technologii. Jeśli zatem nasze programy używają niskopoziomowych cech sprzętu, takich jak rozmiar komórki pamięci, to tracą przenośność! Minimalne i maksymalne wartości poszczególnych typów danych znajdziesz w plikach limits.h i float.h.

Zakres dozwolonych wartości jest szczególnie istotny dla algorytmów matematycznych. Przykładowo w Visual C++ znajdziemy w pliku limits.h następujące informacje:

```
#define INT_MIN  (-2147483647 - 1)  /* minimum (signed) int value  */
#define INT_MAX   2147483647         /* maximum (signed) int value */
```

Jeśli zatem użyjesz zmiennej `int` do wyliczania przypadków pewnego algorytmu, który teoretycznie może się nie zmieścić w powyższym zakresie, to musisz albo użyć zmiennej innego typu, albo zastosować sztuczki podobne do tych podanych np. w rozdziale 13. w punkcie „Kodowanie danych i arytmetyka dużych liczb".

Nawet początkujący programista C++ używa bez problemu nieco bardziej złożonych struktur danych, do jakich należą tablice i rekordy. Tablice i rekordy są na tyle proste, że nie będą one stanowić przedmiotu naszych głębszych rozważań. (Jeśli jednak masz kłopoty ze zrozumieniem składni C++, to zerknij do dodatku A po konkretne przykłady). Warto jednak przytoczyć ogólne cechy tych „nieco bardziej złożonych" struktur danych, aby mieć świadomość ich zalet i wad:

- Tablice (np. `int t[N]`) pozwalają na przechowywanie obok siebie (dosłownie) zbioru `N` zmiennych tego samego typu, adresowanych przez podanie indeksu od `0` do `N-1` (`t[0]`, `t[1]`, ... `t[N-1]`). Dostęp do danych jest zatem naturalny i swobodny: znając indeks konkretnej wartości, można do niej sięgnąć poprzez adresowanie bezpośrednie. Wadą klasycznych tablic jest oczywiście konieczność rezerwacji pewnego stałego obszaru pamięci. Tablice w C++ mogą być wielowymiarowe i zawierać nie tylko zmienne typu podstawowego, ale i rekordy danych.
- Rekordy (w C++ definiowane przez słowo kluczowe `struct`) tworzą nowy typ podstawowy, który pozwala na grupowanie w swego rodzaju „paczce" kilku zmiennych innego typu. Dzięki temu łatwo jest zgrupować np. informacje adresowe, zarobki, numer PESEL (są to tzw. pola) itp. w jednym miejscu. Dostęp do pól jest bardzo naturalny (notacja z kropką — bardziej rozbudowane przykłady w C++ znajdują się w dodatku A), pola mogą też zawierać typy złożone (np. tablice) oraz wskaźniki.


Uwaga

W C++ występuje też słowo kluczowe `union`, zaszłość z języka C pozwalająca na oszczędne wykorzystanie pamięci komputera w pewnych specyficznych sytuacjach poprzez definiowanie tzw. unii, ale ceną jest niska czytelność tworzonego kodu. Na pierwszy rzut oka definicja unii jest bowiem podobna do definicji rekordu, ale jest to tylko pozór: pola w unii „nachodzą" na siebie w pamięci i programista musi ściśle kontrolować w programie, co tak naprawdę z niej odczytuje. Jeśli zależy Ci na czytelności... nie używaj unii, gdyż jest to konstrukcja języka bardzo myląca i w zasadzie zbędna w większości zastosowań.

Ciągi znaków i napisy w C++

Czytelnik ma prawo być nieco zdziwiony pojawieniem się tego punktu, gdyż ma on charakter ściśle związany z C++ i dość rzadko bywa wzmiankowany w podręcznikach algorytmiki (chlubnym wyjątkiem jest **[Sed92]**). Tymczasem wyodrębnienie dyskusji o napisach i ciągach znaków ma

sens, gdyż umiejętność manipulowania napisami w programach ułatwia rozwiązanie wielu zagadnień. Bez sprawnego przetwarzania napisów nie da się napisać sensownej aplikacji biznesowej, więc postanowiłem w tym wydaniu książki pokazać kilka przykładów, które powinny wyjaśnić tajemnicze konstrukcje spotykane w kodzie C++.

Pierwsze, co warto zrozumieć, to fakt, iż ciąg znaków w C++ jest tablicą... o zmiennej długości. Ta pozornie niewielka różnica jest dość istotna z punktu widzenia systemowego: zwykła tablica jest od początku statycznie określona w programie, posiada znaną nam długość i wiemy, jak czytać poszczególne elementy (np. znaki, liczby, rekordy). Ciąg znaków zachowuje się teoretycznie dość podobnie, ale ponieważ jego długość może się zmieniać w trakcie działania programu, to kompilator potrzebuje dodatkowej informacji, znacznika końca, którym jest specjalny kod zero (zapisywany jako '\0').

W C++ spotyka się często deklaracje zbliżone do:

```
char *s1;
char  s2[100];
char *s3="Kod błędu 100";
```

W pierwszym przypadku deklarowany jest wskaźnik, który ewentualnie można przypisać do jakiejś zmiennej tekstowej (np. do argumentów wywołania funkcji main, innej zmiennej tekstowej), ale sam w sobie *nie rezerwuje on pamięci* do przechowywania znaków! Dopiero druga forma pozwala na bezpieczne składowanie znaków i swobodny dostęp do poszczególnych znaków poprzez indeksowanie, bez łamania reguł bezpieczeństwa w zakresie odczytywania pamięci. Trzecia forma jest równoważna zadeklarowaniu wskaźnika s3, przydzieleniu w pamięci miejsca na 14 znaków (tekst + znacznik końca, czyli 0) i zainicjowaniu zawartości napisem „Kod błędu 100".

Popatrzmy na prosty program przykładowy, który odczytuje ciąg znaków, zapisuje go do tablicy i odczytuje jej zawartość, znak po znaku, włącznie ze znacznikiem końca, automatycznie dołożonym przez program.


Listing

znaki.cpp

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
char s[100]; // na razie tu nic nie ma!
cout << "Podaj słowo:";
cin >> s;
for (int i=0; i<=strlen(s); i++) // strlen = długość ciągu
   cout << "Znak [" << i << "]=" << s[i] << ".kod: "
        << (int) s[i] << endl;
}
```

Oto przykładowe wykonanie programu:

```
Podaj słowo:pies
Znak [0]=p.kod: 112
Znak [1]=i.kod: 105
Znak [2]=e.kod: 101
Znak [3]=s.kod: 115
Znak [4]= .kod: 0
```

Gdybyśmy w tym kodzie zastąpili `char s[100]` przez `char *s`, to program zgłosiłby błąd dostępu do pamięci!

Konieczność badania długości ciągów znakowych, kontrolowania ewentualnego przepełniania „buforów" i rezerwowania pamięci była zawsze koszmarem wielu programistów C+, prowadząc do wielu pomyłek wykrywalnych dopiero podczas wykonywania programu. Z tego powodu

w odświeżonej wersji standardu C++ pojawił się wygodny, obiektowy typ danych o nazwie `string`, umożliwiający łatwe operowanie łańcuchami znaków, bez nużącego używania tablic i wskaźników i całej tej męczącej kontroli pamięci.

Popatrzmy na prostą ilustrację użycia obiektów klasy `string`.


Listing

string.cpp

```
int main()
{
string s1, s2="ma kota";        // deklaracja + inicjalizacja
s1 = "ala ";                    // przypisanie wartości
string s3 = s1 + s2 + "\n";     // sklejanie łańcuchów
cout << "s3=" << s3;
s3.erase();                     // zerujemy ciąg znaków
cout << "s3=" << s3;
}
```

Program po uruchomieniu wyświetli:

```
s3=ala ma kota
------
s3=
s2: znak [0]=m,kod: 109
s2: znak [1]=a,kod: 97
s2: znak [2]= ,kod: 32
s2: znak [3]=k,kod: 107
s2: znak [4]=o,kod: 111
s2: znak [5]=t,kod: 116
s2: znak [6]=a,kod: 97
```

Ten prosty kod pokazuje, jak łatwe i naturalne jest tworzenie i modyfikowanie ciągów znaków. Klasa `string` posiada oczywiście wiele ciekawych metod poza np. `erase`, a przedefiniowane (tzn. przeciążone) operatory standardowe pozwalają na przypisywanie i naturalne sklejanie ciągów znaków. Oprócz tego oczywiście pozostaje pełny dostęp do zawartości ciągów, tak jakby to była klasyczna tablica znakowa.


Uwaga

W tej książce w zależności od potrzeb będę stosował klasyczne (tablicowe) podejście i podejście obiektowe, z wykorzystaniem klasy `string`. Aby zostać sprawnym programistą C++ musisz niestety poznać oba te podejścia, gdyż do tej pory napisano bardzo dużo dobrych programów (często przeniesionych z C), które używają głównie podejścia klasycznego.

Abstrakcyjne struktury danych

Prawdziwa przygoda ze strukturami danych rozpoczyna się, dopiero gdy dostajemy do ręki tzw. listy, drzewa binarne, grafy itd. Wraz z nimi rozszerzają się znacznie możliwości rozwiązania programowego wielu ciekawych zagadnień; zwiększa się wachlarz potencjalnych zastosowań informatyki. Ponieważ z tymi strukturami związane są pewne reguły użycia, często określa się je jako *abstrakcyjne*, co w C++ przekłada się na... programowanie obiektowe, które doskonale pozwala implementować takie nieistniejące w naturze (czytaj: kompilatorach) twory!

Struktury danych są olbrzymim ułatwieniem dla programistów, gdyż pozwalają na uporządkowanie informacji zapamiętywanych w komputerach w formie łatwej do zrozumienia dla człowieka. Jak wykażemy w dalszej części książki, są one nie tylko formą organizacji danych, ale i ciekawym narzędziem do rozwiązywania skomplikowanych problemów algorytmicznych, np.:

- *Listy* ułatwiają tworzenie elastycznych baz danych.
- *Drzewa binarne* mogą posłużyć do analizy symbolicznej wyrażeń arytmetycznych.
- *Grafy* ułatwiają rozwiązanie wielu zagadnień z dziedziny tzw. sztucznej inteligencji.

Listy i drzewa binarne omówimy w tym rozdziale, materiał dotyczący grafów został, ze względu na jego znaczenie i objętość, wyodrębniony w rozdziale 10.

W kolejnych podrozdziałach zostaną przedstawione najważniejsze struktury danych i sposoby posługiwania się nimi. Jednocześnie przykłady ilustrujące ich użycie zostały tak wybrane, aby zasugerować niejako ewentualną dziedzinę zastosowań.

Ten rozdział jest bogato ilustrowany kodem w C++, którego poziom skomplikowania może okazać się dużym wyzwaniem dla początkujących adeptów tego języka (np. szablony klas, wskaźniki do funkcji, funkcje zaprzyjaźnione, przeciążanie operatorów...). Ponieważ nowoczesne implementacje języka C++ zawierają bogatą kolekcję gotowych klas realizujących nawet złożone struktury danych, to nie jest wymagane, aby każdy potrafił projektować od zera wszystko, co jest dostępne w postaci gotowej. Warto jednak co najmniej zrozumieć, w jaki sposób projektuje się nieco bardziej złożone klasy, aby nabrać wprawy w używaniu pewnych technik programowania specyficznego dla C++. Zapraszam zatem do lektury zarówno osoby zainteresowane samymi strukturami danych, jak i programistów pragnących wyłącznie podejrzeć gotowe rozwiązania niejako „od kuchni".

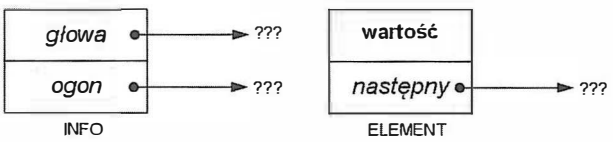
Listy jednokierunkowe

Lista jednokierunkowa jest oszczędną pamięciowo strukturą danych, pozwalającą grupować dowolną — ograniczoną tylko ilością dostępnej pamięci — liczbę elementów: liczb, znaków, rekordów itd. Jest to duża zaleta w porównaniu z tablicami, których rozmiar może być, co prawda, określany dynamicznie, ale przydział dużego, „liniowego" obszaru pamięci podczas wykonywania programu nie zawsze musi się zakończyć sukcesem. Nietrudno sobie bowiem wyobrazić, że o wiele bardziej prawdopodobne jest bezproblemowe przydzielenie przez system operacyjny 50 000 razy pamięci na pojedyncze dane (powiedzmy 1 kB), niż zarezerwowanie „za jednym zamachem" miejsca na tablicę o rozmiarze 50 000 kB. Ponadto tablica, przynajmniej na początku, i tak będzie prawie w ogóle nieużywana!

Użycie listy wymaga zarezerwowania pewnych dodatkowych informacji wskaźnikowych, które oczywiście także zajmują miejsce w pamięci, nie przechowując w sobie bezpośrednio danych „biznesowych". Ten narzut jednak jest niewielki i w dużych programach może być pomijalny, zwłaszcza dla wielkich rekordów informacyjnych.

Do budowy listy jednokierunkowej używane są dwa typy komórek pamięci (rysunek 5.1)[2]:

- Rekord natury informacyjnej (oznaczenie `INFO`), zawierający dwa wskaźniki: *do początku* listy i *do końca* listy. Wskaźnik jest adresem komórki w pamięci komputera i przechowywany jest w zmiennej typu wskaźnikowego.
- Rekord o charakterze roboczym (oznaczenie `ELEMENT`). Zawiera pole *wartości* (tutaj wpisujemy zawartość informacyjną, np. liczbę, ciąg znaków, zbiór atrybutów itd.) i wskaźnik na następny element listy.



Rysunek 5.1. *Typy rekordów używanych podczas programowania list*

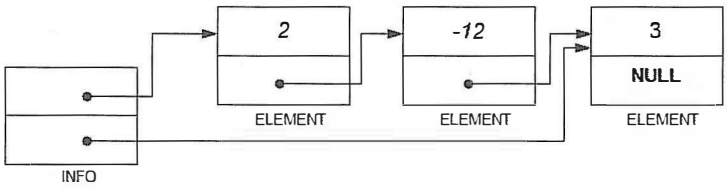
[2] Na rysunku pogrubiona kropka oznacza wartość typu „wskaźnik".

Zauważyłem, że w większości opisów struktur listowych nie wzmiankuje się zazwyczaj rekordu informacyjnego (nie jest on bezpośrednio elementem struktury danych) — jest to oczywisty błąd. Kosztem kilku bajtów pamięci[3] uzyskujemy bowiem ciągły dostęp do bardzo istotnych operacji i ułatwiamy ogromnie operację podstawową: dołączenie nowego elementu na koniec listy (jeśli nie wstawiamy na koniec listy, to zawsze możemy przyłączyć nowy element na początek listy, ale tracimy wówczas informację o kolejności przybywania danych!).

Pola: `głowa`, `ogon` i `następny` są wskaźnikami[4], natomiast `wartość` może być czymkolwiek — liczbą, znakiem, rekordem, etc. W przykładach znajdujących się w tej książce dla uproszczenia operuje się głównie wartościami typu całkowitego, co nie umniejsza bynajmniej ogólności wywodu. Ewentualne przeróbki tak uproszczonych algorytmów należą już raczej do kosmetyki niż do zmian o charakterze zasadniczym.

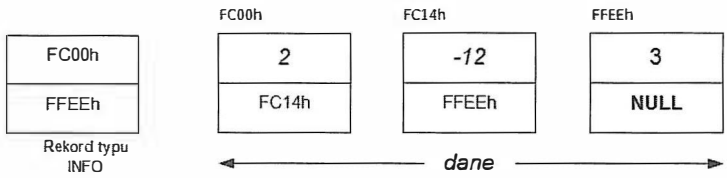
Idea jest zatem następująca: jeżeli lista jest pusta, to struktura informacyjna zawiera dwa wskaźniki `NULL`. Warto pamiętać, że w ogólnym przypadku `NULL` nie jest bynajmniej równy zeru — jest to pewien adres, na który na pewno żadna zmienna nie wskazuje (taka jest ogólna idea wskaźnika `NULL`, niestety wielu programistów o tym nie pamięta). Pierwszy element listy jest złożony z jego własnej wartości (informacji do przechowania) oraz ze wskaźnika na drugi element listy. Drugi zawiera własne pole informacyjne i, oczywiście, wskaźnik na trzeci element listy itd. Miejsce zakończenia listy zaznaczamy także poprzez wartość specjalną, wskaźnik `NULL`. Spójrzmy na rysunek 5.2 przedstawiający listę złożoną z trzech elementów: 2, -12, 3.

Rysunek 5.2. *Przykład listy jednokierunkowej (1)*



Rysunek 5.3 jest dokładnym odbiciem swojego poprzednika — z tą tylko różnicą, że — w miejsce strzałek symbolizujących „wskazywanie" — są użyte konkretne wartości liczbowe adresów komórek pamięci. Liczba szesnastkowa, umieszczona nad rekordem, jest adresem w pamięci operacyjnej komputera, pod którym zostało mu przydzielone miejsce przez standardową procedurę `new`[5].

Rysunek 5.3. *Przykład listy jednokierunkowej (2)*



Wróćmy jeszcze do analizy rekordów składających się na listę. Pole `głowa` struktury informacyjnej wskazuje na komórkę zawierającą 2 — pierwszy element listy, czyli — wyrażając się jaśniej — zawiera adres, pod którym w pamięci komputera jest zapamiętany rekord.

---

[3] Wielkość zmiennej wskaźnikowej zależy od potrzeb adresowych wymaganych dla używanego modelu pamięci i determinuje ją zasadniczo system operacyjny.

[4] Fakt wskazywania na coś jest symbolizowany dalej przez strzałki.

[5] Ze względów historycznych warto może przypomnieć, że w klasycznym języku C trzeba było w celu przydzielania pamięci używać funkcji bibliotecznych `calloc` i `malloc`. W C++ instrukcja `new` wykonuje dokładnie to samo, lecz o wiele czytelniej, i stanowi już element języka.

Pole ogon struktury informacyjnej wskazuje na komórkę zawierającą 3 (ostatni element listy). Pola te służą do przeglądania elementów listy i do dołączania nowych. Oto jak może wyglądać procedura przeglądająca elementy listy, np. w poszukiwaniu wartości x (komórka informacyjna nazywa się info):

```
adres_tmp=info.głowa
dopóki (adres_tmp != NULL) wykonuj
  {
   jeśli(adres_tmp.wartość == x) to
        {
         Wypisz "Znalazłem poszukiwany element"
         opuść procedurę
         }
    w przeciwnym przypadku
         adres_tmp=adres_tmp.następny
   }
wypisz "Nie znalazłem poszukiwanego elementu"
```

W dalszej części rozdziału będziemy przeplatać opis algorytmów w pseudojęzyku programowania (takim jak wyżej) z gotowym kodem C++; kryterium wyboru będzie czytelność procedur. Oczywiście nawet jeśli prezentacja algorytmu zostanie dokonana w pseudokodzie, to wersja dyskietkowa będzie zawierała pełne wersje w języku C++, gotowe do kompilacji i uruchomienia.

Realizacja struktur danych listy jednokierunkowej

Poniższa implementacja struktur potrzebnych do programowej obsługi listy jednokierunkowej jest dokładnym odzwierciedleniem rysunku 5.2 i nie należy się tu spodziewać szczególnych niespodzianek. Osoby, które nie znają jeszcze zbyt dobrze składni języka C++, powinny dobrze zapamiętać sposób deklaracji typów danych rekurencyjnych (tzn. zawierających wskaźniki do elementów swojego typu). Różni się on bowiem odrobinę od sposobu używanego na przykład w Pascalu (patrz również dodatek A).


Listing

lista.h

```
// definicje typów danych
enum szukanie {PORAZKA=0, SUKCES=1};
typedef struct rob
{
int wartosc;
struct rob *nastepny; // wskaźnik do następnego elementu
}ELEMENT;

// początek deklaracji klasy LISTA
class LISTA
{
public:
  // nagłówki:
    friend LISTA& operator +(LISTA&,LISTA&); // sumuje dwie listy
    friend void fuzja(LISTA &x,LISTA &y);
    void wypisz();            // wypisuje zawartość listy
    int szukaj(int x);        // szuka elementu x na liście
    void dorzucl(int x);      // dorzuca bez sortowania
    void dorzuc2(int x);      // dorzuca z sortowaniem
    LISTA& operator --(int); // usuwa ostatni element listy

// kilka prostych metod:
    bool  pusta()             // czy lista coś zawiera?
    {
    return (inf.glowa==NULL);
    }
```

```
    void zeruj()              // zeruje listę bez wykonywania „delete"
    {
    inf.glowa=inf.ogon=NULL;
    }

    LISTA()     // konstruktor
    {
    inf.glowa=inf.ogon=NULL;
    }
    ~LISTA()
    {
// destruktor, który używa przedefiniowanego operatora --
    while (!pusta()) (*this)--;
    }
private:
// struktura informacyjna zapewni dostęp do listy
typedef struct
{
ELEMENT *glowa;
ELEMENT *ogon;
}INFO;
INFO inf;
}; //koniec deklaracji klasy LISTA
```

Pole `wartość` w naszym przykładzie jest typu `int`, ale w praktyce może to być bardzo złożony rekord informacyjny (np. zawierający *imię*, *nazwisko*, *wiek* itd.).

Kilka prostych metod klasy LISTA zdefiniowaliśmy już w pliku nagłówkowym. Jeśli metoda jest trywialna i charakteryzuje się małymi rozmiarami, to często definiuje się ją wprost w ciele klasy. Przykładem może być miniaturowa funkcja usługowa `pusta`, która pomimo swej prostoty ma szansę być dość często używana w praktyce.

Klasa `LISTA` nie jest zbyt rozbudowana, jednak zawiera kilka rozwiązań, które wymagają dość szczegółowego komentarza. Osoby słabiej znające C++ mogą mieć kłopot ze zrozumieniem deklaracji:

```
LISTA& operator --(int);
```

Jest to oczywiście nagłówek metody, która przedefiniuje operator `--`, ale do czego służy ten dodatkowy parametr typu `int`? Otóż wynika on z wymogów normalizacyjnych języka C++. Parametr ten jest sztuczny i wyłącznie informuje kompilator o tym, że przedefiniujemy w tej metodzie operator „przyrostkowy".

Kwestią otwartą pozostaje wybór ewentualnego utajnienia typów danych (patrz deklaracje `public` i `private`); programista musi sam podjąć odpowiednią decyzję, mając na uwadze takie aspekty, jak: *sens* ujawniania/ukrywania atrybutów, parametry „sprawnościowe" metod, późniejsze dziedziczenie, etc. Propozycje przedstawione w tym rozdziale w żadnym razie nie pretendują do miana rozwiązań wzorcowych — takie bowiem nie istnieją wobec nieskończonej w zasadzie liczby nowych sytuacji i problemów, z którymi może się w praktyce spotkać programista. Staraniem autora było raczej pokazanie istniejącej różnorodności, a nie przekonywanie do jednych rozwiązań przy jednoczesnym pomijaniu innych.

W następnych paragrafach zostaną przedstawione wszystkie metody, które były wyżej wzmiankowane jedynie poprzez swoje nagłówki.

Tworzenie listy jednokierunkowej

Niewątpliwie najwyższa już pora na przedstawienie sposobu dołączania elementów do listy. Posłuży nam do tego celu kilka funkcji o mniejszym lub większym stopniu skomplikowania. Potrzeba sprawdzania, czy jakieś elementy już są zapamiętane na liście, wystąpi przykładowo w funkcji `dorzucl`, która dołącza nowy element do listy.

Podczas dokładania nowego elementu możliwe są dwa podejścia: albo będziemy traktować listę jako zwykły worek do gromadzenia danych nieuporządkowanych (będzie to wówczas naukowy sposób na zwiększanie bałaganu), albo też przyjmiemy założenie, że nowe elementy dokładane będą w liście we właściwym, ustalonym przez nas porządku — na przykład sortowane od razu w kierunku wartości niemalejących.

Pierwszy przypadek jest trywialny — odpowiadająca mu procedura `dorzucl` jest przedstawiona poniżej.

Listing

lista.cpp

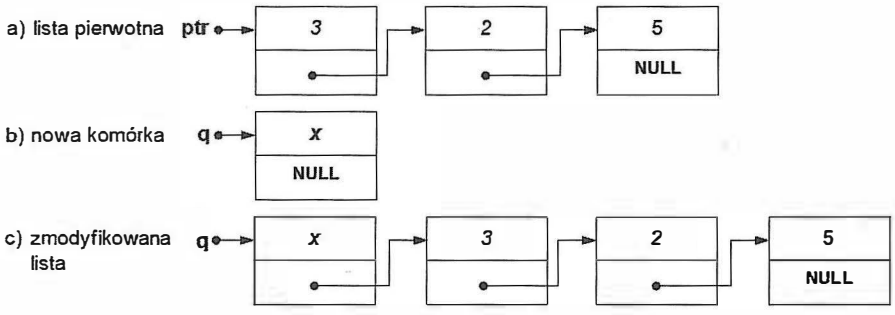
```
// Fragment pliku lista.cpp, na razie bez funkcji main:
#include <iostream>
using namespace std;
#include "LISTA.H"              // bez tej dyrektywy kompilator nie zrozumie
                                // nowego typu danych LISTA
void LISTA::dorzucl(int x)      // dołączamy rekord na koniec listy
{                               // bez sortowania; operator :: jest
   ELEMENT *q=new ELEMENT;      // niezbędny, bowiem definiujemy kod
   q->wartosc=x;                // metody poza „cialem" klasy
   q->nastepny=NULL;
   if (pusta())                 // lista pusta?
      inf.glowa=inf.ogon=q;
      else
// coś jest w liście
      {
         (inf.ogon)->nastepny=q; // pole następny jest
         inf.ogon=q;             // wskaźnikiem, stąd -> a nie . (kropka)
      }
}
```

Działanie funkcji `dorzucl` jest następujące: w przypadku pustej listy oba pola struktury informacyjnej są inicjowane wskaźnikiem na nowo powstały element. W przeciwnym wypadku nowy element zostaje podpięty do końca, stając się tym samym ogonem listy.

Oczywiście możliwe jest dokładanie nowego rekordu przez *pierwszy* element listy (wskazywanej zawsze przez pewien łatwo dostępny wskaźnik, powiedzmy: `ptr`), stawałby się on wówczas automatycznie głową listy i musiałby zostać zapamiętany przez program, aby nie stracić dostępu do danych:

```
ELEMENT      *q=new ELEMENT;     // a)
              q->wartosc=x;      // b)
              q->nastepny=ptr;   // c)
```

Kod ten może być zilustrowany schematem z rysunku 5.4.



Rysunek 5.4. *Dołączanie nowego elementu listy na jej początek*

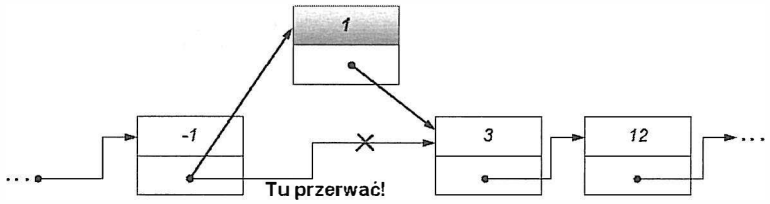

Ostrzeżenie

W tym i dalszych przykładach przyjmowane jest założenie, że przydział pamięci ZAWSZE kończy się sukcesem. W rzeczywistych programach jest to przypuszczenie dość niebezpieczne i warto sprawdzać, czy istotnie po użyciu instrukcji używającej `new`, np. `ELEMENT *q=new ELEMENT;` wartości `q` nie zostało przypisane `NULL`! Z uwagi na chęć zapewnienia klarowności prezentowanych algorytmów tego typu kontrola zostanie w książce pominięta; podczas realizacji „prawdziwego" programu takie niedopatrzenie może się okazać dość przykre w skutkach.

Sposób podany powyżej jest poprawny, ale pamiętajmy, że dokładając nowe elementy zawsze na *początek* listy, tracimy istotną czasami informację na temat *kolejności* nadchodzenia elementów!

O wiele bardziej złożona jest funkcja dołączająca nowy element w takie miejsce, aby całość listy była widziana jako posortowana (tutaj: w kierunku wartości niemalejących). Ideę przedstawia rysunek 5.5, gdzie możemy zobaczyć sposób dołączania liczby `1` do już istniejącej listy złożonej z elementów `-1`, `3` i `12`.

Rysunek 5.5. *Dołączanie elementu listy z sortowaniem*

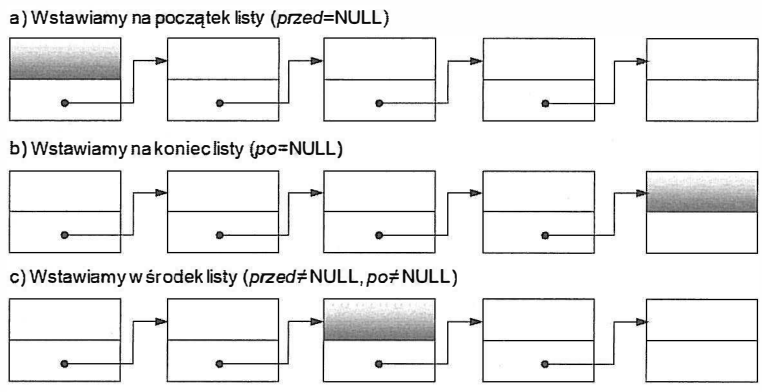


Nowy element (narysowany pogrubioną kreską i zacieniowany) może zostać wstawiony na początek *(a)*, koniec *(b)* listy, jak również gdzieś w jej środku *(c)*. W każdym z tych przypadków w istniejącej liście trzeba znaleźć miejsce wstawienia, tzn. zapamiętać dwa wskaźniki: element, *przed którym* mamy wstawić nową komórkę, i element, *za którym* mamy to zrobić. Do zapamiętania tych istotnych informacji posłużą nam zmienne `przed` i `po`.

Następnie, gdy dowiemy się, gdzie jesteśmy, możemy dokonać wstawienia nowego elementu do listy. Sposób, w jaki tego dokonamy, zależy oczywiście od miejsca wstawienia i od tego, czy lista przypadkiem nie jest jeszcze pusta. Krótko mówiąc, realizacja jest niestety dość złożona. Pewne skomplikowanie funkcji `dorzuc2` wynika z połączenia w niej poszukiwania miejsca wstawienia z samym dołączeniem elementu. Równie dobrze można by te dwie czynności rozbić na osobne funkcje — nie zostało to jednak uczynione w obecnej wersji.

Istnieją 3 przypadki „współrzędnych" nowego elementu w liście, symbolicznie przedstawione na rysunku 5.6 (zakładamy, że lista już coś zawiera).

Rysunek 5.6. *Wstawianie nowego elementu do listy — analiza przypadków*



W zależności od ich wystąpienia zmieni się sposób dołączenia elementu do listy. Oto pełny tekst funkcji dorzuc2, która swoje działanie opiera właśnie na idei przedstawionej na rysunku 5.6:

```
void LISTA::dorzuc2(int x)     // dołączamy rekord na właściwe miejsce
{                              //z sortowaniem
ELEMENT *q=new ELEMENT;        // tworzymy nowy element listy
q->wartosc=x;
// Poszukiwanie właściwej pozycji na wstawienie elementu
if (pusta())
   {
   inf.glowa=inf.ogon=q;
   q->nastepny=NULL;
   }
   else //szukamy miejsca na wstawienie
      {
      ELEMENT *przed=NULL,*po=inf.glowa;
   //zmienna wyliczeniowa
      enum {SZUKAJ,ZAKONCZ} stan=SZUKAJ;
      while ((stan==SZUKAJ) && (po!=NULL))
         if (po->wartosc>=x)
            stan=ZAKONCZ;// znaleźliśmy właściwe miejsce!
         else              // przemieszczamy się w poszukiwaniach
            {              // właściwego miejsca
            przed=po;      //wskaźniki „przed" i „po"
            po=po->nastepny; // zapamiętają miejsce wstawiania
            }
      if (przed==NULL)     // wstawiamy na początek listy
         {
         inf.glowa=q;
         q->nastepny=po;
         } else
         if (po==NULL)     // wstawiamy na koniec listy
            {
            inf.ogon->nastepny=q;
            q->nastepny=NULL;
            inf.ogon=q;
            }
         else              // wstawiamy gdzieś w środku
            {
            przed->nastepny=q;
            q->nastepny=po;
            }
        }
}
```

Kolejne ważne, choć skrajnie nieskomplikowane, metody są niemalże identyczne koncepcyjnie. W celu znalezienia w liście pewnego elementu x należy przejrzeć ją za pomocą zwykłej pętli while:

```
int LISTA::szukaj(int x)
{
ELEMENT *q=inf.glowa;
while (q != NULL)
   {
   if (q->wartosc==x)
      return SUKCES;
   q=q->nastepny;
   }
return PORAZKA;
}
```

Identyczną strukturę posiada metoda wypisz, służąca do wypisywania zawartości listy:

```
void LISTA::wypisz()
{
ELEMENT *q=inf.glowa;
```

```
if (pusta()) cout << "(lista pusta)";
else
   while (q != NULL)
   {
   cout << q->wartosc << "   ";
   q=q->nastepny;
   }
cout << "\n";
}
```

Pora na nieco trudniejsze fragmenty kodu.

Zaczniemy od operacji usuwania *ostatniego* elementu listy, do której to czynności zatrudniliśmy przedefiniowany operator dekrementacji (`--`). Tak jak już wcześniej wspominałem, dodatkowy parametr typu `int` wynika z wymogów normalizacyjnych języka C++, jest sztuczny i wyłącznie informuje kompilator o tym, że przedefiniujemy w tej metodzie operator „przyrostkowy".

Funkcja, która się za nim ukrywa, jest relatywnie prosta: jeśli na liście jest tylko jeden element, to modyfikacji ulegnie zarówno pole `glowa`, jak i pole `ogon` struktury informacyjnej. Oba te pola, po uprzednim usunięciu jedynego elementu listy, zostaną zainicjowane wartością `NULL`.

Nieco trudniejszy jest przypadek, gdy lista zawiera więcej niż jeden element. Należy wówczas odszukać *przedostatni* jej element, aby móc odpowiednio zmodyfikować wskaźnik `ogon` struktury informacyjnej. Znajomość przedostatniego elementu listy umożliwi nam łatwe usunięcie ostatniego jej elementu. Poniżej jest zamieszczony pełny tekst funkcji wykonującej to zadanie.

```
LISTA& LISTA::operator --(int) // int jest parametrem sztucznym
{                              // (operator -- będzie przyrostkowy)
if (inf.glowa==inf.ogon)       // jeden element (lub lista pusta)
   {
   delete inf.glowa;
   inf.glowa=inf.ogon=NULL;
   } else
   {
   ELEMENT *temp=inf.glowa;
   while ((temp->nastepny) != inf.ogon) // szukamy przedostatniego
      temp=temp->nastepny;              // elementu listy...
      inf.ogon=temp;
      delete temp->nastepny;            // ... i usuwamy go
      temp->nastepny=NULL;
   }
return (*this);  // zwracamy zmodyfikowany obiekt
}
```

Obiekt jest zwracany poprzez swój adres, czyli może posłużyć jako argument dowolnej dozwolonej na nim operacji. Przykładowo: możemy utworzyć wyrażenie `(l2--)--.wypisz()`. Mimo groźnego wyglądu działanie tej instrukcji jest trywialne: pierwsza dekrementacja zwraca prawdziwy, fizycznie istniejący obiekt, który jest poddawany od razu drugiej dekrementacji. Rezultat tej ostatniej — jako pełnoprawny obiekt — może aktywować dowolną metodę swojej klasy, czyli przykładowo sprawdzić swoją zawartość za pomocą funkcji `wypisz`.

Przy okazji omawiania operatora dekrementacji spójrzmy jeszcze na inne jego zastosowanie. W definicji klasy został zawarty jej destruktor. Przypomnijmy, że destruktor jest specjalną funkcją wywoływaną automatycznie podczas niszczenia obiektu. To niszczenie może być bezpośrednie, np. za pomocą operatora `delete`:

```
LISTA *p=new LISTA;      // tworzymy nowy obiekt...
...
delete p;                // ...i niszczymy go!
```

lub też pośrednie, w momencie gdy obiekt przestaje być dostępny. Przykładem tej drugiej sytuacji niech będzie następujący fragment programu:

```
if (warunek)
{
LISTA p; // tworzymy obiekt lokalny
...      //widoczny tylko w tej instrukcji if
}
```

Obiekt p zadeklarowany w ciele instrukcji `if` jest dla niej całkowicie lokalny. Żaden inny fragment programu nie ma prawa dostępu do niego. Z takim tymczasowym obiektem wiąże się czasem dość sporo pamięci zarezerwowanej tylko dla niego. Otóż gdyby nie było destruktora, programista nie miałby wcale pewności, czy ta pamięć została w całości zwrócona systemowi operacyjnemu. Celowo podkreślam, że w całości, bowiem automatyczne zwalnianie pamięci jest możliwe tylko w przypadku tych zmiennych, które są z założenia lokowane na stosie. Dotyczy to np. zwykłych pól obiektu, ale nie jest możliwe w przypadku struktur dynamicznych, które są nierzadko rozsiane po dość sporym obszarze pamięci komputera. Tak jest w przypadku list, drzew, tablic dynamicznych, etc. W takim przypadku programista musi sam napisać jawny destruktor, który znając[6] doskonale sposób, w jaki pamięć została przydzielona obiektowi, będzie ją umiał prawidłowo zwrócić.

Tak też się dzieje w naszym przykładzie. Destruktor ma zaskakująco prostą budowę:

```
~LISTA()
// destruktor, który używa przedefiniowanego operatora --
{
while (!pusta()) (*this)--;
}
```

Jest to zwykła pętla `while`, która tak długo usuwa elementy z listy, aż stanie się ona pusta. Mimo iż nie jest to optymalny sposób na zwolnienie pamięci, został jednak zastosowany w celu ukazania możliwych zastosowań wskaźnika `this`, który — jak wiemy — wskazuje na własny obiekt. Linia `(*this)--` oznacza dla danego obiektu wykonanie na sobie operacji dekrementacji. Obiekt ulegający z pewnych powodów destrukcji (typowe przypadki zostały wzmiankowane wcześniej) wywoła swój destruktor, który zaaplikuje na sobie tyle razy funkcję dekrementacji, aby całkowicie zwolnić pamięć wcześniej przydzieloną liście.

Kolejna porcja kodu do omówienia dotyczy redefinicji operatora + (plus). Naszym celem jest zbudowanie takiej funkcji, która umożliwi *dodawanie* list w jak najbardziej dosłownym znaczeniu tego słowa. Chcemy, aby w wyniku następujących instrukcji:

```
LISTA x,y,z;    // tworzymy 3 puste listy.
x.dorzuc2(3); x.dorzuc2(2); x.dorzuc2(1);
y.dorzuc2(6); y.dorzuc2(5); y.dorzuc2(4);
z=x+y;
```

Lista wynikowa `z` zawierała wszystkie elementy list `x` i `y`, tzn.: 1, 2, 3, 4, 5 i 6 (posortowane!). Najprostszą metodą jest przekopiowanie wszystkich elementów z list `x` i `y` do listy `z` i jednoczesne aktywowanie na rzecz tej ostatniej metody `dorzuc2`. Zapewni to utworzenie listy już posortowanej:

```
LISTA& operator +(LISTA &x,LISTA &y)
{
LISTA *temp=new LISTA;
ELEMENT *q1=(x.inf).glowa; //wskaźniki robocze
ELEMENT *q2=(y.inf).glowa;
while (q1 != NULL) //przekopiowanie listy x do temp
   {
   temp->dorzuc2(q1->wartosc);
   q1=q1->nastepny;
   }
```

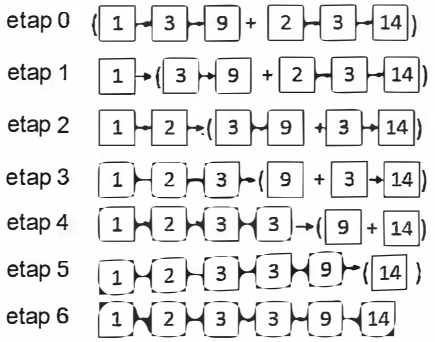
[6] De facto to *my* go znamy i dzielimy się tą cenną wiedzą z destruktorem. Występujące tu i ówdzie personifikacje są nie do uniknięcia w tego typu opisach!

```
while (q2 != NULL) // przekopiowanie listy y do temp
   {
   temp->dorzuc2(q2->wartosc);
   q2=q2->nastepny;
   }
return (*temp);
}
```

Czy jest to najlepsza metoda? Chyba nie, chociażby z uwagi na niepotrzebne dublowanie danych. Ideałem byłoby posiadanie metody, która wykorzystując fakt, iż listy są już posortowane[7], dokona ich zespolenia ze sobą (tzw. *fuzji*) przy użyciu wyłącznie istniejących komórek pamięci, bez tworzenia nowych. Inaczej mówiąc, będziemy zmuszeni do manipulowania wyłącznie wskaźnikami i to jest jedyne narzędzie, jakie dostaniemy do ręki!

Na rysunku 5.7 możemy przykładowo prześledzić, jak powinna być wykonywana fuzja pewnych dwóch list `x=(1,3,9)` i `y=(2,3,14)`, tak aby w jednej z nich znalazły się wszystkie elementy `x` i `y` — oczywiście posortowane (w naszym przykładzie niemalejąco).

Rysunek 5.7.
Fuzja list na przykładzie



Najmniejszym z dwóch pierwszych elementów list jest `1` i on też będzie stanowił zaczątek nowej listy. Następnikiem tego elementu będzie fuzja dwóch list: `x' = (3,9)` i `y=(2,3,14)`.

Jak dokonać fuzji list `x'` i `y`? Dokładnie tak samo: bierzemy element `2`, który jest najmniejszym z dwóch pierwszych elementów list `x'` i `y`... Można tak *rekurencyjnie* kontynuować aż do momentu, gdy natrafimy na przypadki elementarne: jeśli jedna z list jest pusta, to fuzją ich obu będzie oczywiście ta druga lista. Na tej zasadzie jest skonstruowana procedura `fuzja(ob1,ob2)`, która wywołana z dwoma parametrami `ob1` i `ob2` zwróci w liście `ob1` sumę elementów list `ob1` i `ob2`. Lista `ob2` jest w wyniku tej operacji zerowana, choć jej całkowite usunięcie pozostaje ciągle w gestii programisty (taki jest nasz wybór — równie dobrze można by to zrobić od razu).

Nasze zadanie wykonamy w dość nietypowy dla C++ sposób, który ma na celu ukazanie zakresu możliwych zastosowań tzw. *funkcji zaprzyjaźnionych* z klasą. Przypomnijmy, iż są to funkcje (lub procedury), które — nie będąc metodami danej klasy — mają dostęp do zastrzeżonych pól `private` i `protected`) obiektu, którego adres został im przekazany jako jeden z parametrów wywołania. Ponieważ nie są to metody, nie mogą być wywoływane w ramach notacji z kropką, a ponadto obiekt, na który mają działać, musi im zostać przekazany w sposób jawny — na przykład poprzez swój adres.

Fuzję list wykonamy w dwóch etapach. Najpierw przygotujemy prostą funkcję, która — otrzymując dwie posortowane listy `a` i `b` — zwróci jako wynik listę będącą ich fuzją. Rekurencyjny zapis tego procesu jest bardzo prosty i zbliżony stylem do rozwiązywania problemów listowych w takich językach jak LISP lub PROLOG:

[7] Zakładamy tym samym użycie metody `dorzuc2` podczas tworzenia listy.

```
ELEMENT *sortuj(ELEMENT *a,ELEMENT *b)
{
if (a==NULL)
return b;
if (b==NULL)
return a;
if (a->wartosc<=b->wartosc)
   {
   a->nastepny=sortuj(a->nastepny,b);
   return a;
   }else
   {
   b->nastepny=sortuj(b->nastepny,a);
   return b;
   }
}
```

Dysponując już funkcją `sortuj`, możemy zastosować ją w procedurze `fuzja`. Będąc zaprzyjaźnioną z klasą `LISTA`, może ona dowolnie manipulować prywatnymi komponentami list `x` i `y`, które zostały jej przekazane w wywołaniu[8].

```
void fuzja(LISTA &x,LISTA &y)
{// a i b muszą być posortowane
ELEMENT *a=x.inf.glowa,*b=y.inf.glowa;
ELEMENT *wynik=sortuj(a,b);
x.inf.glowa=wynik;
if(x.inf.ogon->wartosc <= y.inf.ogon->wartosc)
   x.inf.ogon=y.inf.ogon;
else x.inf.ogon=x.inf.ogon;
   y.zeruj();
}
```

Celowo znacznie rozbudowana funkcja `main` ilustruje sposób korzystania z opisanych wyżej funkcji. Do obu list są dołączane elementy tablic, następnie ma miejsce testowanie niektórych metod oraz sortowanie dwóch list poprzez ich fuzję.

```
int main()
{
LISTA l1,l2;
const int n=6;
int tabl[n]={2,5,-11,4,14,12};
// każdy element tablicy zostanie wstawiony do listy
cout << "\nL1 = ";
for (int i=0; i<n; l1.dorzuc2(tabl[i++]));
   l1.wypisz();
int tab2[n]={9,6,77,1,7,4};
cout << "L2 = ";
for (int i=0; i<n; l2.dorzuc2(tab2[i++]));
   l2.wypisz();
cout << "Efekt poszukiwań liczby 14 w l1: " << l1.szukaj(14) << endl;
cout << "Efekt poszukiwań liczby 0 w  l1: " << l1.szukaj(0) << endl;
cout << "Oto lista będąca sumą dwóch poprzednich\nL3 = ";
   LISTA l3=l1+l2;
   l3.wypisz();
cout << "Listy L1 i L2 pozostały bez zmian:\nL1 = ";
   l1.wypisz();
cout << "L2 = ";
   l2.wypisz();
cout << "Lista L1 bez dwóch ostatnich elementów:\nL1 = ";
   (l1--)--.wypisz();
cout << "Efekt fuzji L1 z L2:\n";
   fuzja(l1,l2);
```

[8] Patrz też dodatek A, w którym zostało omówione na innym przykładzie pojęcie funkcji zaprzyjaźnionej.

```
cout << "L1 = ";
   l1.wypisz();
cout << "L2 = ";
   l2.wypisz();
   l1.dorzuc2(80);l1.dorzuc2(8);
cout << "dorzucamy do L1 liczby 80 i 8\nL1 = ";
   l1.wypisz();
}
```

Oto wyniki uruchomienia programu:

```
L1 = -11   2   4   5   12   14
L2 = 1   4   6   7   9   77
Efekt poszukiwań liczby 14 w l1: 1
Efekt poszukiwań liczby 0 w l1: 0
Oto lista będąca sumą dwóch poprzednich
L3 = -11   1   2   4   4   5   6   7   9   12   14   77
Listy L1 i L2 pozostały bez zmian:
L1 = -11   2   4   5   12   14
L2 = 1   4   6   7   9   77
Lista L1 bez dwóch ostatnich elementów:
L1 = -11   2   4   5
Efekt fuzji L1 z L2:
L1 = -11   1   2   4   4   5   6   7   9   77
L2 = (lista pusta)
dorzucamy do L1 liczby 80 i 8
L1 = -11   1   2   4   4   5   6   7   8   9   77   80
```

Listy jednokierunkowe — teoria i rzeczywistość

Oprócz pięknie brzmiących rozważań teoretycznych istnieje jeszcze twarda rzeczywistość, w której mają wykonywać się nasze pieczołowicie przygotowane programy[9].

Spójrzmy obiektywnie na listy jednokierunkowe pod kątem ich wad i zalet:

- *Wady*: nienaturalny dostęp do elementów, niełatwe sortowanie, utrudniona analiza zawartości i ocena wielkości listy.
- *Zalety*: efektywne zużycie pamięci, elastyczność.

Przeanalizujmy szczególnie uważnie zagadnienie sortowania danych, będących elementami listy. Wyobrażamy sobie zapewne, że posortowanie w pamięci struktury danych, która nie jest w niej rozłożona liniowo (tak jak ma to miejsce w przypadku tablicy), jest dość złożone.

Lista, do której nowe elementy są wstawiane, już na samym początku konsekwentnie w określonym porządku służy, oprócz swojej podstawowej roli gromadzenia danych, także do ich porządkowania. Jest to piękna właściwość: sama struktura danych dba o sortowanie! W sytuacji, gdy istnieje tylko jedno kryterium sortowania (np. w kierunku wartości niemalejących pewnego pola `x`), możemy mówić o ideale. Cóż jednak mamy począć, gdy elementami listy są rekordy o bardziej skomplikowanej strukturze, np.:

```
struct
{
char imie[20];
char nazwisko[30];
int wiek;
int kod_pracownika;
}
```

Raz możemy zechcieć dysponować taką listą uporządkowaną alfabetycznie, wg nazwisk, innym razem będzie nas interesował wiek pracownika itd. Czy należy w takim przypadku dysponować dwiema wersjami tych list — co pochłania cenną pamięć komputera — czy też może

[9] Wbrew wszelkim przesłankom nie jest to definicja systemu operacyjnego.

zdecydujemy się na sortowanie listy w pamięci? Jednak uwaga: to drugie rozwiązanie zajmie z kolei cenny czas procesora!

Poruszony powyżej problem był na tyle charakterystyczny dla wielu rzeczywistych programów, że zostało do jego rozwiązania wymyślone pewne sprytne rozwiązanie, które postaram się dość szczegółowo omówić.

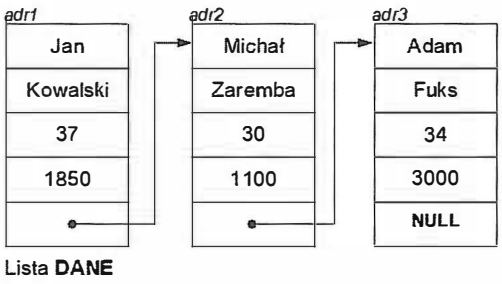
Pomysł polega na uproszczeniu i skomplikowaniu zarazem tego, co poznaliśmy wcześniej. Uproszczenie polega na tym, że *rekordy zapamiętywane w liście nie są w żaden sposób wstępnie sortowane*. Inaczej mówiąc, do zapamiętywania możemy użyć odpowiednika jakże prostej funkcji `dorzuc1` (patrz strona 97). Słowo „odpowiednik" pasuje tutaj najlepiej, bowiem niezbędne okaże się wprowadzenie kilku kosmetycznych zabiegów związanych z ogólną zmianą koncepcji.

Obok listy danych będziemy ponadto dysponować kilkoma listami wskaźników do nich. List tych będzie tyle, ile sobie zażyczymy kryteriów sortowania.

Jak nietrudno się domyślić, jeśli nie zamierzamy sortować listy danych (a jednocześnie chcemy mieć dostęp do danych posortowanych!), to podczas wstawiania nowego adresu do którejś z list wskaźników musimy dokonać jej sortowania. Zadanie jest zbliżone do tego, które wykonywała funkcja `dorzuc2`, z tą tylko różnicą, że dostęp do danych nie odbywa się w sposób bezpośredni.

Podczas sortowania list wskaźników dane nie są w ogóle ruszane — przemieszczaniu w listach będą ulegały wyłącznie same wskaźniki! Na tym etapie ma prawo to wszystko brzmieć dość enigmatycznie, pora zatem na jakiś konkretny przykład. Popatrzmy w tym celu na rysunek 5.8.

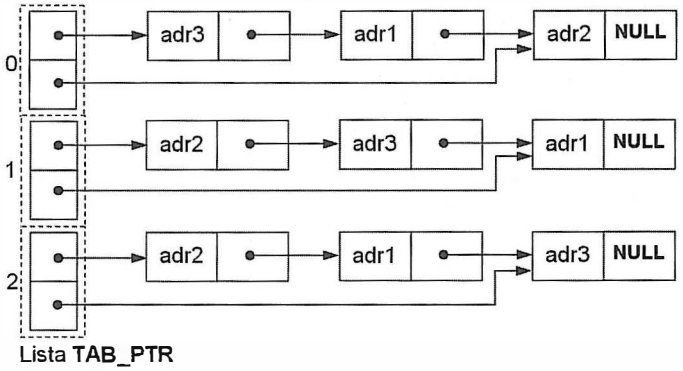
Rysunek 5.8. *Sortowanie listy bez przemieszczania jej elementów (1)*



Zawiera on listę o nazwie `DANE`, zbudowaną z kilku rekordów, które stanowią zaczątek miniaturowej bazy danych o pracownikach pewnego przedsiębiorstwa. Przyjmijmy dla uproszczenia, że jedyne istotne informacje, które chcemy zapamiętać, to: imię, nazwisko, pewien kod i oczywiście zarobek. Na rysunku są zaznaczone symbolicznie adresy rekordów: `adr1`, `adr2` i `adr3`, przydzielone przez funkcję `dorzuc1`.

Rysunek 5.9 zawiera już kilka nowości w porównaniu z tym, co mieliśmy okazję do tej pory poznać.

Rysunek 5.9. *Sortowanie listy bez przemieszczania jej elementów (2)*



Tablica TAB_PTR zawiera rekordy informacyjne (tzn. wskaźniki głowa i ogon) do list złożonych z adresów rekordów z listy DANE — w naszym przypadku zakładamy *3* listy wskaźników i będą one oczywiście zawierać adresy adr1, adr2 i adr3 (chwilowo na liście znajdują się trzy elementy; w miarę dokładania nowych elementów do listy DANE będą ulegały odpowiedniemu wzrostowi listy wskaźników).

Rozmiar tablicy TAB_PTR jest równy liczbie kryteriów sortowania: patrząc od góry, możemy zauważyć, że listy są posortowane kolejno wg nazwiska, kodu i zarobków.

Podsumujmy informacje, które można odczytać z rysunków 5.8 i 5.9:

- Nieposortowana baza danych, która jest zapamiętana w liście o nazwie DANE, zawiera w danym momencie *3* rekordy.
- Tablica wskaźników TAB_PTR zawiera *3* rekordy informacyjne (poznane już poprzednio), których pola głowa i ogon umożliwiają dostęp do trzech list wskaźników. Każda z tych list jest posortowana wg innego kryterium sortowania:
  - Lista wskazywana przez TAB_PTR[0] jest posortowana alfabetycznie wg nazwisk pracowników (Fuks, Kowalski i Zaremba).
  - Analogicznie TAB_PTR[1] klasyfikuje pracowników wg pewnego kodu używanego w tej fabryce (Zaremba, Fuks i Kowalski).
  - Ostania lista, TAB_PTR[2], grupuje pracowników wg ich zarobków.

Poniżej jest przedstawiona nowa wersja klasy LISTA, uwzględniająca już propozycje przedstawione na rysunku 5.8. Aby umożliwić sensowną prezentację w postaci programu przykładowego, pewnemu uproszczeniu uległa struktura danych zawierająca informacje o pracowniku: ograniczymy się tylko do nazwiska i zarobków. (Rozbudowa tych struktur danych nie wniosłaby koncepcyjnie nic nowego, natomiast zagmatwałaby i tak dość pokaźny objętościowo listing).

Struktury danych prezentują się w nowej wersji następująco:

```
typedef struct rob
   {
   char nazwisko[100];
   long  zarobek;
   struct rob *nastepny;          // wskaźnik do
   }ELEMENT;                      // następnego elementu

typedef struct rob_ptr            // struktura robocza listy
{                                 // wskaźników
ELEMENT *adres;
struct rob_ptr *nastepny;
}LPTR;
```

Olbrzymich zmian jak na razie nie ma i uważny Czytelnik mógłby się słusznie zapytać, dlaczego nie zostały wykorzystane mechanizmy dziedziczenia, aby maksymalnie wykorzystać już napisany kod? Powód jest prosty: poprzednia wersja klasy LISTA służyła w zasadzie do ukazania mechanizmów i podstawowych algorytmów związanych z listami jednokierunkowymi; jej zastosowanie praktyczne było w związku z tym raczej nikłe i użycie jej jako klasy bazowej byłoby nieco sztuczne. W C++ klasy bazowe powinny być jak najbardziej uniwersalne, tak aby ich użycie do dziedziczenia nie miało sztucznego charakteru.

Obecnie prezentowana wersja struktury listy jednokierunkowej charakteryzuje się bardzo dużą elastycznością użytkowania i to właśnie ona mogłaby ewentualnie posłużyć jako klasa bazowa w dalszej hierarchii dziedziczenia (o ile Czytelnik w istocie będzie w ogóle potrzebował mechanizmów dziedziczenia, mechanizm ten nie powinien być stosowany jako sztuka dla sztuki).

Oto nowa wersja klasy LISTA:


Listing

lista2.h

```
const int n=5;                  // rozmiar tablic przykładowych
const int kryteria_sort=2;      // liczba kryteriów sortowania

typedef struct rob
{
   char nazwisko[100];
   long  zarobek;
   struct rob *nastepny;        // wskaźnik do
}ELEMENT;                       // następnego elementu

typedef struct rob_ptr          // struktura robocza listy
{                               // wskaźników
   ELEMENT *adres;
   struct rob_ptr *nastepny;
}LPTR;
class LISTA
{
public:
   LISTA();                     // konstruktor
  ~LISTA();                     // destruktor
   void dorzuc(ELEMENT *);      // dołącz nowy element q
   void wypisz(char);           // wypisz zawartość listy
   int  usun(ELEMENT*,int(*decyzja)(ELEMENT *,ELEMENT*));
   // usuń element, który jest zgodny z wzorcową komórką podaną
   // jako parametr
private: // prywatne struktury:
   typedef struct               // struktura informacyjna listy danych
   {
   ELEMENT *glowa;
   ELEMENT *ogon;
   }
   INFO;
typedef struct    // struktura informacyjna listy wskaźników
   {
   LPTR *glowa;
   LPTR *ogon;
   }LPTR_INFO;

   LPTR_INFO inf_ptr[kryteria_sort];
   INFO info_dane;
   // metody „wewnętrzne", nieudostępnione publicznie:
LPTR_INFO *odszukaj_wsk(LPTR_INFO*,ELEMENT*,
     int(*)(ELEMENT*,ELEMENT*));
ELEMENT *usun_wsk(LPTR_INFO*, ELEMENT*, int(*)(ELEMENT*,ELEMENT*));
int usun_dane(ELEMENT*);
void dorzuc2(int, ELEMENT*, int(*decyzja)(ELEMENT*,ELEMENT*));
void wypisz1(LPTR_INFO*);
};
```

Tajemnicze metody prywatne, podane wyżej bez żadnego opisu, zostaną szczegółowo omówione w następnych paragrafach.

Analizując procedury i funkcje do obsługi list, można zauważyć, że operacje odszukiwania pewnego elementu wg podanego wzorca (np. „odszukaj pracownika, który zarabia 1 200 zł") i wyszukiwania miejsca na wstawienie nowego elementu różniły się nieznacznie. Od tego spostrzeżenia do gotowej realizacji programowej jest już tylko jeden krok. Aby go zrobić, musimy dobrze zrozumieć zasady operowania wskaźnikami do funkcji[10] w C++, bowiem ich użycie pozwoli na

[10] Miłośnicy i znawcy języka LISP mogą opuścić ten paragraf.

eleganckie rozwiązanie kilku problemów. Zdając sobie sprawę, że wskaźniki do funkcji są relatywnie rzadko stosowane, niezbędne wydało mi się przypomnienie sposobu ich stosowania w C++. Jest to ukłon głównie w stronę programistów pascalowych, bowiem w ich ulubionym języku ten mechanizm w ogóle nie istnieje.

Przedstawiony poniżej przykład ilustruje sposób użycia wskaźników do funkcji w C++.


Listing

wsk_fun.cpp

```
int do_2(int a)
{
return a*a;
}
int do_4(int a)
{
return a*a*a*a;
}

int wzor(int x,int(*fun)(int))
{
return fun(x);
}
int main()
{
cout << "10 do potęgi 2:"<< wzor(10,do_2) << endl;
cout << "10 do potęgi 4:"<< wzor(10,do_4) << endl;
}
```

Funkcja `wzor` zwraca — w zależności od tego, czy zostanie wywołana jako `wzor(10,do_2)` czy też `wzor(10,do_4)` — odpowiednio: `100` lub `10000`. Mamy tu do czynienia z podobnym fenomenem, jak w przypadku tablic, gdzie nazwa (tablicy, funkcji) jest jednocześnie wskaźnikiem do niej. Bezpośrednią konsekwencją jest dość naturalny sposób użycia, pozwalający na uniknięcie typowych dla C++ operatorów * (operator „wyłuskania" wartości) i & (operator adresowy).

Inny przykład: pewna procedura `f`, która otrzymuje jako parametr liczbę `x` (typu `int`) i wskaźnik do funkcji o nazwie *g* (zwracającej typ `double` i operującej trzema parametrami: `int`, `double`, i `char*`), może zostać zadeklarowana w następujący sposób:

```
void f(int x, double(*g)(int, double, char *))
{
   k=g(12.5,345,"1984");
   cout << k << endl;
}
```

Zakres stosowania wskaźników do funkcji jest dość szeroki i przyczynia się do uogólnienia wielu procedur i funkcji.

Powróćmy teraz do odsuniętych chwilowo na bok list i zajmijmy się problemem wstawiania nowego elementu do listy uprzednio posortowanej. Chcemy znaleźć dwa adresy: `przed` i `po` (patrz rysunek 5.6 na stronie 98), które umożliwią nam takie zmodyfikowanie wskaźników, aby cała lista była widziana jako posortowana. W tym celu zmuszeni jesteśmy do użycia pętli `while` poznanej na stronie 99:

```
while((stan==SZUKAJ) && (po!=NULL))
   if (po->zarobek >= x)
       stan=ZAKONCZ;
   else
       {
       przed=po;
       po=po->nastepny;
       }
```

Gdybyśmy zaś chcieli usunąć pewien element listy, który spełnia przykładowo warunek, że pole zarobek wynosi 1200 zł, to również będą nam potrzebne wskaźniki przed i po. Odnajdziemy je w sposób następujący:

```
while((stan==SZUKAJ) && (po!=NULL))
   if (po->zarobek == 1200)
       stan=ZAKONCZ;
   else
       {
       przed=po;
       po=po->nastepny;
       }
```

Różnica pomiędzy tymi dwiema pętlami while tkwi wyłącznie w warunku instrukcji if-else. Idea naszego rozwiązania jest zatem następująca: napiszemy uniwersalną funkcję, która posłuży do odszukiwania wskaźników przed i po w celu ich późniejszego użycia do dokładania elementów do listy, jak również do ich usuwania. Funkcja ta powinna nam zwrócić oba wskaźniki — posłużymy się do tego celu strukturą LPTR_INFO (patrz listing LISTA2.H), umawiając się, że pole glowa będzie odpowiadało wskaźnikowi przed, a pole ogon — wskaźnikowi po.

Łatwo jest zauważyć, że operacje poszukiwania, wstawiania, etc. rozpoczynamy od listy wskaźników, z której zdobędziemy adres rekordu danych (adres ten zostanie zapamiętany w polu adres struktury LPTR, która stanowi element składowy listy wskaźników — patrz rysunek 5.9). Dopiero po zmodyfikowaniu wszystkich list wskaźników (a może ich być tyle, ile przyjmiemy kryteriów sortowania) należy zmodyfikować listę danych. Pracy jest — jak widać — mnóstwo, ale jest to cena za wygodę późniejszego użytkowania takiej listy! Pocieszeniem niech będzie fakt, że po jednokrotnym napisaniu odpowiedniego zestawu funkcji bazowych będziemy mogli z nich później wielokrotnie korzystać bez konieczności przypominania sobie, jak one to robią. Przejdźmy już do opisu realizacji funkcji odszukaj_wsk, która zajmie się poszukiwaniem wskaźników przed i po, zwracając je w strukturze LPTR_INFO:

- Wskaźnik inf do struktury informacyjnej listy wskaźników; adres początku znajduje się w polu głowa, a adres końca w polu ogon.
- Wskaźnik q do pewnego fizycznie istniejącego rekordu danych. Jest to albo nowy rekord, który chcemy dołączyć do listy, albo po prostu pewien szablon poszukiwań.
- Wskaźnik decyzja do funkcji porównawczej, która zostanie włożona do instrukcji if w pętli while.


Listing

lista2.cpp

```
LISTA::LPTR_INFO* LISTA::odszukaj_wsk(LISTA::LPTR_INFO *inf,ELEMENT *q,
                                      int(*decyzja)(ELEMENT *q1,ELEMENT *q2))
{
LPTR_INFO *res=new LPTR_INFO;
res->glowa=res->ogon=NULL;
if (inf->glowa==NULL)
   return (res);    // lista pusta!
else
   {
   LPTR *przed,*pos;
   przed=NULL;
   pos=inf->glowa;
   enum {SZUKAJ,ZAKONCZ} stan=SZUKAJ;
   while ((stan==SZUKAJ) && (pos!=NULL))
      if (decyzja(pos->adres,q))
         stan=ZAKONCZ;        // znaleźliśmy miejsce, w którym element
      else                    // istnieje (albo ma być wstawiony)
         {                    // przemieszczamy się w poszukiwaniach
```

```
            przed=pos;
            pos=pos->nastepny;
            }
      res->glowa=przed;
      res->ogon=pos;
      return (res);
      }
}
```

Przykładowo, jeśli chcemy odszukać i usunąć pierwszy rekord, który w polu `nazwisko` zawiera „Kowalski", to należy stworzyć tymczasowy rekord, który będzie miał odpowiednie pole wypełnione tym nazwiskiem (pozostałe nie będą miały wpływu na poszukiwanie):

```
ELEMENT *f=new ELEMENT;
strcpy(f->nazwisko,"Kowalski");
```

Podobna uwaga należy się pozostałym kryteriom poszukiwań — wg zarobków, imienia, etc. Jeśli poszukiwanie zakończy się sukcesem, to w polu `ogon` zostanie zwrócony adres fizycznie istniejącego rekordu, który odpowiadał wzorcowi naszych poszukiwań. W przypadku gdyby element taki nie istniał, powinny zostać zwrócone wartości `NULL`. Znajomość wskaźników `przed` i `po` umożliwi nam zwolnienie komórek pamięci zajmowanych dotychczas przez rekord danych, jak również odpowiednie zmodyfikowanie całej listy, tak aby wszystko było na swoim miejscu.

Innym przykładem zastosowania funkcji jest dołączenie nowego elementu do listy. Trzeba wówczas stworzyć nowy rekord, wypełnić jego pola i umieścić go na końcu listy. Następnie należy adres tego elementu wstawić do list wskaźników posortowanych wg zarobków, nazwisk czy też dowolnych innych kryteriów. W każdej z tych list miejsce wstawienia będzie inne, czyli za każdym razem różne mogą być wartości wskaźników `przed` i `po`, które zwróci funkcja `odszukaj_wsk`.

Zastosowanie funkcji `odszukaj_wsk` jest, jak widać, bardzo wszechstronne. Taka elastyczność możliwa była do osiągnięcia tylko i wyłącznie poprzez użycie wskaźników do funkcji — we właściwym miejscu i o właściwej porze.

Oto garść funkcji decyzyjnych, które mogą zostać użyte jako parametr:


Listing

lista2.h

```
int alfabetycznie(ELEMENT *q1,ELEMENT *q2)
{ // czy rekordy q1 i q2 są uporządkowane alfabetycznie?
   return (strcmp(q1->nazwisko,q2->nazwisko)>=0);
}
int wg_zarobkow(ELEMENT *q1,ELEMENT *q2)
{ // czy rekordy q1 i q2 są uporządkowane wg zarobków?
   return (q1->zarobek>=q2->zarobek);
}
int ident_nazwiska (ELEMENT *q1,ELEMENT *q2)
{ // czy rekordy q1 i q2 mają identyczne nazwiska?
   return (strcmp(q1->nazwisko,q2->nazwisko)==0);
}
int ident_zarobki (ELEMENT *q1,ELEMENT *q2)
{ // czy rekordy q1 i q2 mają identyczne zarobki?
   return(q1->zarobek==q2->zarobek);
}
```

Dwie pierwsze funkcje z powyższej listy służą do jej porządkowania, pozostałe ułatwiają proces wyszukiwania elementów. Oczywiście w rzeczywistej aplikacji bazy danych o pracownikach analogiczne funkcje byłyby nieco bardziej skomplikowane — wszystko zależy od tego, jakie kryteria wyszukiwania lub porządkowania zamierzamy zaprogramować oraz jak skomplikowane struktury danych wchodzą w grę.

Po tak rozbudowanych wyjaśnieniach działanie funkcji odszukaj_wsk nie powinno stanowić już dla nikogo tajemnicy.

Na 95 stronie mieliśmy okazję zapoznać się z funkcją pusta informującą, czy lista danych coś zawiera. Nic nie stoi na przeszkodzie, aby do kompletu dołożyć jej kolejną wersję, badającą w analogiczny sposób listę wskaźników[11]:

```
inline int pusta(LPTR_INFO *inf)
{
return (inf->glowa==NULL);
}
```

Ponieważ użyliśmy dwukrotnie tej samej nazwy funkcji, nastąpiło w tym momencie jej *przeciążenie*; podczas wykonywania programu właściwa jej wersja zostanie wybrana w zależności od typu parametru, z którym zostanie wywołana (wskaźnik do struktury INFO lub wskaźnik do struktury LPTR_INFO).

Mając już komplet funkcji pusta, zestaw funkcji decyzyjnych i uniwersalną funkcję odszukaj_wsk, możemy pokusić się o napisanie brakującej procedury dorzuc1, która będzie służyła do dołączania nowego rekordu do listy danych z jednoczesnym sortowaniem list wskaźników. Załóżmy, że będą tylko dwa kryteria sortowania danych, co implikuje, iż tablica zawierająca „wskaźniki do list wskaźników" będzie miała tylko dwie pozycje (patrz rysunek 5.9).

Adres tej tablicy, jak również wskaźniki do listy danych i do nowo utworzonego elementu, zostaną obowiązkowo przekazane jako parametry:

```
void LISTA::dorzuc(ELEMENT *q)
{                                   // rekord dołączamy bez sortowania
if (info_dane.glowa==NULL)          // lista pusta
   info_dane.glowa=info_dane.ogon=q;
else                    // coś jest w liście
   {
   (info_dane.ogon)->nastepny=q;
   info_dane.ogon=q;
   }
// dołączamy wskaźnik do rekordu do listy posortowanej alfabetycznie
dorzuc2(0,q,alfabetycznie);
// dołączamy wskaźnik do rekordu do listy posortowanej wg zarobków
dorzuc2(1,q,wg_zarobkow);
}
```

Funkcja jest bardzo prosta, głównie z uwagi na tajemniczą procedurę o nazwie dorzuc2. Oczywiście nie jest to jej poprzedniczka ze strony 99, choć różni się od tamtej doprawdy niewiele:

```
void LISTA::dorzuc2(int nr,ELEMENT *q,int(*decyzja)(ELEMENT *q1, ELEMENT *q2))
{
LPTR *wsk=new LPTR;
wsk->adres=q; // wpisujemy adres rekordu q
// Poszukiwanie właściwej pozycji na wstawienie elementu
if (inf_ptr[nr].glowa==NULL) // pusta
   {
   inf_ptr[nr].glowa=inf_ptr[nr].ogon=wsk;
   wsk->nastepny=NULL;
   }
else //szukamy miejsca na wstawienie
   {
   LPTR *przed,*po;
   LPTR_INFO *gdzie=odszukaj_wsk(&inf_ptr[nr].q,decyzja);
```

[11] Funkcja typu inline oznacza tak naprawdę jej wywołanie w formie makra, tj. w miejsce jej wystąpienia zostanie wstawiony cały kod w niej zawarty, wydłużając kod wynikowy. Zaletą inline jest znaczące polepszenie czytelności programu, w którym zbiory prymitywnych grup funkcji (np. wielokrotnych przypisań) są zastępowane jednym „aliasem". Ponadto nie jest to wywołanie funkcyjne, dzięki czemu unikamy całego bagażu związanego z obsługą przez stos.

```
   przed=gdzie->glowa;
   po=gdzie->ogon;
   if (przed==NULL)  // wstawiamy na początek listy
      {
      inf_ptr[nr].glowa=wsk;
      wsk->nastepny=po;
      } else
           if (po==NULL) // wstawiamy na koniec listy
              {
              inf_ptr[nr].ogon->nastepny=wsk;
              wsk->nastepny=NULL;
              inf_ptr[nr].ogon=wsk;
              }else     // wstawiamy gdzieś „w środku"
                 {
                 przed->nastepny=wsk;
                 wsk->nastepny=po;
                 }
           }
}
```

W celu zrozumienia dokonanych modyfikacji właściwe byłoby porównanie obu wersji funkcji dorzuc2, aby wykryć różnice, które między nimi istnieją. „Filozoficznie" nie ma ich wiele — w miejsce sortowania danych sortujemy po prostu wskaźniki do nich.

Funkcja zajmująca się usuwaniem rekordów wymaga przesłania m.in. fizycznego adresu elementu do usunięcia. Mając tę informację, należy wyczyścić zarówno listę danych, jak i listy wskaźników:

```
int  LISTA::usun(ELEMENT *q,int(*decyzja)(ELEMENT *q1,ELEMENT *q2))
{
//usuwa calkowicie informacje z obu list (wskaźników i danych)
ELEMENT *ptr_dane;
for (int i=0; i<kryteria_sort; i++)
   ptr_dane=usun_wsk(&inf_ptr[i].q,decyzja);
if (ptr_dane==NULL)
   return(0);
else
   return  usun_dane(ptr_dane);
}
```

Funkcja usun_wsk zajmuje się usuwaniem wskaźników danego elementu z list wskaźników — jakakolwiek byłaby ich liczba. Czytelnik może zauważyć z łatwością, że raz jeszcze mamy tu do czynienia z bardzo podobnym do poprzednich schematem algorytmu.

Można nawet odważyć się na stwierdzenie, że listing jest zamieszczany wyłącznie gwoli formalności! Elementarna kontrola błędów jest zapewniana przez wartość zwracaną przez funkcję: w normalnej sytuacji winien to być różny od NULL adres fizycznego rekordu przeznaczonego do usunięcia.

```
ELEMENT* LISTA::usun_wsk(LPTR_INFO *inf,ELEMENT *q, int(*decyzja)(ELEMENT *q1,ELEMENT *q2))
{
if (inf->glowa==NULL)    // lista pusta, czyli nie ma czego usuwać!
   return NULL;
else                     //szukamy elementu do usunięcia
   {
   LPTR *przed,*pos;
   LPTR_INFO *gdzie=odszukaj_wsk(inf,q,decyzja);
   przed=gdzie->glowa;
   pos=gdzie->ogon;
   if (pos==NULL)
      return NULL;  // element nieodnaleziony

   if (pos==inf->glowa)          // usuwamy z początku listy
      inf->glowa=pos->nastepny;
   else
```

```
        if (pos->nastepny==NULL) // usuwamy z końca listy
           {
           inf->ogon=przed;
           przed->nastepny=NULL;
        } else     // usuwamy gdzieś „ze środka"
           przed->nastepny=pos->nastepny;
        ELEMENT *ret=pos->adres;
        delete pos;
        return ret;
     }
}
```

Funkcja usun_dane jest zbudowana wg podobnego schematu, co funkcja usun_wsk. Ponieważ przyjmowane jest założenie, że *element, który chcemy usunąć, istnieje*, programista musi zapewnić dokładną kontrolę poprawności wykonywanych operacji. Tak się dzieje w naszym przypadku — ewentualna nieprawidłowość zostanie wykryta już podczas próby usunięcia wskaźnika i wówczas usunięcie rekordu po prostu nie nastąpi.

```
int LISTA::usun_dane(ELEMENT *q)
{// założenie: q istnieje!
ELEMENT *przed,*pos;
przed=NULL;
pos=info_dane.glowa;
while ((pos!=q) && (pos!=NULL)) // szukamy elementu „przed"
   {
   przed=pos;
   pos=pos->nastepny;
   }
if (pos!=q)
   return(0); // element nieodnaleziony?!
if (pos==info_dane.glowa)          // usuwamy z początku listy
   {
   info_dane.glowa=pos->nastepny;
   delete pos;
} else
   if (pos->nastepny==NULL) // usuwamy z końca listy
   {
   info_dane.ogon=przed;
   przed->nastepny=NULL;
   delete pos;
   }else     // usuwamy gdzieś „ze środka"
      {
      przed->nastepny=pos->nastepny;
      delete pos;
      }
   return(1);
}
```

Pomimo wszelkich prób uczynienia powyższych funkcji bezpiecznymi kontrola w nich zastosowana jest ciągle bardzo uproszczona. Czytelnik, który będzie zajmował się implementacją dużego programu w C++, powinien bardzo dokładnie kontrolować poprawność operacji na wskaźnikach. Programy stają się wówczas co prawda mniej czytelne, ale jest to cena za mały, lecz jakże istotny szczegół: ich poprawne działanie.

Poniżej znajduje się rozbudowany przykład użycia nowej wersji listy jednokierunkowej. Jest to dość spory fragment kodu, ale zdecydowałem się na jego zamieszczenie, biorąc pod uwagę względne skomplikowanie omówionego materiału — ktoś nieprzyzwyczajony do sprawnego operowania wskaźnikami miał prawo się nieco zgubić. Zakładam zatem, że szczegółowy przykład zastosowania może mieć duże znaczenie dla ogólnego zrozumienia całości.

Dwie proste funkcje wypisz1 i wypisz zajmują się eleganckim wypisaniem na ekranie zawartości bazy danych w kolejności narzuconej przez odpowiednią listę wskaźników:

```
void LISTA::wypisz1(LPTR_INFO *inf)
{ // wypisujemy zawartość posortowanej listy wskaźników (oczywiście nie interesuje nas
LPTR *q=inf->glowa; // wypisanie wskaźników, gdyż są to adresy), lecz
while (q != NULL)   //informacji, na które one wskazują
   {
   cout << setw(9)<<q->adres->nazwisko<< " zarabia "<<setw(4)<<
          q->adres->zarobek<<"zł\n";
   q=q->nastepny;
   }
cout << "\n";
}

void LISTA::wypisz(char c)
{
if (c=='a')   // alfabetycznie
   wypisz1(&inf_ptr[0]);
else
   wypisz1(&inf_ptr[1]);
}
```

Funkcja `main` testuje wszystkie nowo poznane mechanizmy:

```
int main()
{
LISTA l1;
char *tabl[n]={"Bec","Becki","Fikus","Pertek","Czerniak"};
int   tab2[n]={1300,1000,1200,2000,3000};
for (int i=0; i<n; i++)
   {
   ELEMENT *nowy=new ELEMENT;      // tworzymy fizycznie nowy rekord...
   strcpy(nowy->nazwisko,tabl[i]);
   nowy->zarobek= tab2[i];
   nowy->nastepny=NULL;
   l1.dorzuc(nowy);   // ...i dorzucamy go do listy
   }
cout << "\n*** Baza danych posortowana alfabetycznie ***\n";
l1.wypisz('a');
cout << "*** Baza danych posortowana wg zarobków ***\n";
l1.wypisz('z');
ELEMENT *f=new ELEMENT;
         f->zarobek=2000;

cout << "Wynik usunięcia rekordu pracownika zarabiającego 2000zł=
"<< l1.usun(f, ident_zarobki) <<endl;
delete f;
cout << "*** Baza danych posortowana alfabetycznie ***\n";
l1.wypisz('a');
cout << "*** Baza danych posortowana wg zarobków ***\n";
l1.wypisz('z');
}
```

Uruchomienie programu powinno dać następujące wyniki:

```
*** Baza danych posortowana alfabetycznie ***
      Bec zarabia 1300zł
    Becki zarabia 1000zł
 Czerniak zarabia 3000zł
    Fikus zarabia 1200zł
   Pertek zarabia 2000zł
*** Baza danych posortowana wg zarobków ***
    Becki zarabia 1000zł
    Fikus zarabia 1200zł
      Bec zarabia 1300zł
   Pertek zarabia 2000zł
 Czerniak zarabia 3000zł
```

```
Wynik usunięcia rekordu pracownika zarabiającego 2000zł=1
*** Baza danych posortowana alfabetycznie ***
      Bec zarabia 1300zł
    Becki zarabia 1000zł
 Czerniak zarabia 3000zł
    Fikus zarabia 1200zł
*** Baza danych posortowana wg zarobków ***
    Becki zarabia 1000zł
    Fikus zarabia 1200zł
      Bec zarabia 1300zł
 Czerniak zarabia 3000zł
```

Tablicowa implementacja list

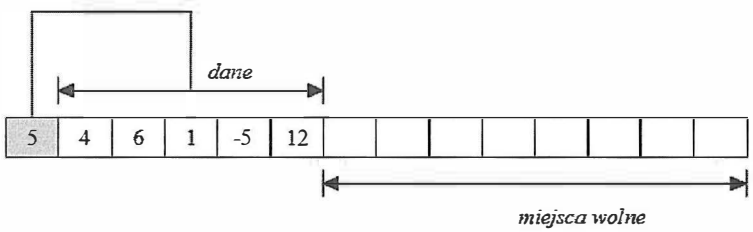
Programowanie w C++ zmusza niejako programistę do dobrego poznania operacji na dynamicznych strukturach danych, sprawnego żonglowania wskaźnikami, etc. Nie da się ukryć, że nie wszyscy wskaźniki lubią. Przyczyn tej niechęci należy upatrywać głównie w próbach programowania na przykład struktur listowych bez pełnego zrozumienia tego, co się chce zrobić. Efekty najczęściej są opłakane, a winę w takich przypadkach ponosi rzecz jasna „chłopiec do bicia", czyli sam język programowania. Tymczasem, podobnie zresztą jak i w życiu, to samo można zrobić na wiele sposobów — o czym niejednokrotnie zapominamy.

Tak też jest i z listami. Okazuje się, że istnieje kilka sposobów tablicowej implementacji list, niektóre z nich charakteryzują się nawet dość istotnymi zaletami, niemożliwymi do uzyskania w realizacji klasycznej (czyli tej, którą mieliśmy okazję poznać wcześniej). Olbrzymią wadą tablicowych wersji struktur listowych jest marnotrawstwo pamięci — przydzielamy przecież na stałe pewien obszar pamięci, powiedzmy dla 1 000 elementów — bo tyle w porywach będziemy potrzebowali miejsca. Gdyby natomiast nasz program używał listy o długości 200 elementów, to i tak obszar realnie zajmowany wynosiłby 1 000 (!). Jest to jednak cena nie do uniknięcia, płacimy ją za prostotę realizacji.

Klasyczna reprezentacja tablicowa

Jedną z najprostszych metod zamiany tablicy na listę jest umówienie się co do sposobu interpretacji jej zawartości. Jeśli powiemy sobie głośno (i nie zapomnimy o tym zbyt szybko), że *i*-temu indeksowi tablicy będzie odpowiadać *i*-ty element listy, to problem mamy prawie z głowy. To „prawie" wynika z tego, że trzeba się umówić, ile maksymalnie elementów zechcemy zapamiętać na liście. Oprócz tego konieczne będzie wybranie jakiejś zmiennej do zapamiętywania aktualnej liczby elementów wstawionych wcześniej do listy.

Ideę ilustruje rysunek 5.10, gdzie możemy zobaczyć tablicową implementację listy 5-elementowej złożonej z elementów: 4, 6, 1, -5 i 12:



Rysunek 5.10. *Tablicowa implementacja listy*

Programowa realizacja jest bardzo prosta — deklaracja przykładowej klasy nie powinna zawierać żadnych niespodzianek dla Czytelnika:


Listing

lista_tab.cpp

```
const int MaxTab=200;
class ListaTab
{
public:
   ListaTab() { tab[0]=0; } // konstruktor klasy
                            // metody zdefiniowane nieco dalej:
   void UsunElement(int k);
   void WstawElement(int x);
   void WstawElement(int x, int k);
   void WypiszListe();
private:
   int tab[MaxTab];          // tab[0] zarezerwowane!
};
```

Omówmy błyskawicznie wszystkie funkcje usługowe klasy. Przypuśćmy, że chcemy dysponować możliwością usunięcia *k*-tego elementu naszej listy. Po zbadaniu sensu takiej operacji (element musi istnieć!) wystarczy przesunąć zawartość tablicy o jeden w lewo od *k*-tej pozycji. Podczas przesuwania element nr k jest bezpowrotnie „zamazywany" przez swojego sąsiada:

```
void ListaTab::UsunElement(int k)
{ //usuwamy k-ty element listy, k >= 1
if((k>=1) && (k<=tab[0]))
   {
   for(int i=k;i<tab[0];i++)
      tab[i]=tab[i+1];
   tab[0]--;
   }
}
```

Wariantów przedstawionej wyżej funkcji może być dość sporo. Mam nadzieję, że Czytelnik w miarę swoich specyficznych wymagań będzie mógł je sobie stworzyć i dostosować do konkretnych potrzeb.

Co jednak z *dołączaniem* elementów do listy? Poniżej są omówione dwie wersje odpowiedniej funkcji: pierwsza wstawia na koniec listy, druga na *k*-tą jej pozycję. Oczywiście w przypadku tej drugiej funkcji niezbędne jest dokonanie odpowiedniego przesunięcia zawartości tablicy, podobnie jak w metodzie UsunElement:

```
void ListaTab::WstawElement(int x)
{//wstawiamy na koniec listy
if(tab[0]<MaxTab-1)
   tab[++tab[0]]=x;
}
//-----------------------------------------------
void ListaTab::WstawElement(int x,int k)
{//wstawiamy na k-tą pozycję listy
if((k>=1) && (k<=tab[0]+1) && (tab[0]<MaxTab-1))
   {
   for(int i=tab[0];i>=k;i--)
      tab[i+1]=tab[i];// robimy miejsce
   tab[k]=x;
   tab[0]++;
   }
}
```

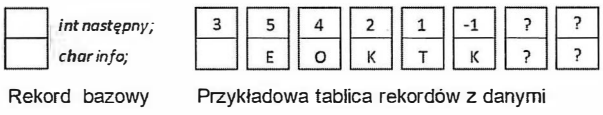
Zasady posługiwania się taką pseudolistą są już po stworzeniu wszystkich metod identyczne z zasadami pracy z prawdziwą listą jednokierunkową, dlatego też darujemy sobie cytowanie funkcji main.

Możliwe jest oczywiście takie zdefiniowanie klasy ListaTab, aby dołączanie elementów następowało już w porządku malejącym, rosnącym czy też wedle jakiegoś innego klucza — Czytelnik może odpowiednio rozbudować funkcje i metody w ramach nieskomplikowanego ćwiczenia.

Metoda tablic równoległych

W poprzednio poznanej implementacji list za pomocą zwykłej tablicy przypisaliśmy na sztywno *i*-temu elementowi tablicy *i*-ty element listy. W prostych zastosowaniach może to wystarczyć w zupełności, jednak rozwiązanie takie jest o wiele bardziej zbliżone ideowo do tablicy niż do listy. Prawdziwa lista powinna umożliwiać dość dowolne układanie elementów i sortowanie ich przy użyciu tylko i wyłącznie wskaźników. Chcieliśmy jednak od wskaźników, przydziałów pamięci, procedur `new` i `delete` uciec jak najdalej! Czyżby ich użycie było nieuniknione?

Odpowiedź na szczęście brzmi: NIE! Wszystko można w końcu zasymulować, więc czemu nie wskaźniki?! Popularna metoda polega na zadeklarowaniu tablicy rekordów składających się z pola informacyjnego `info` i pola typu całkowitego `następny`, które służy do odszukiwania elementu „następnego" na liście. Dobrze znane i klasyczne wręcz rozwiązanie. Idea jest przedstawiona na rysunku 5.11, gdzie można zobaczyć przykładową implementację listy służącej do przechowywania znaków, zawierającej w danym momencie pięć liter układających się w słowo „KOTEK".



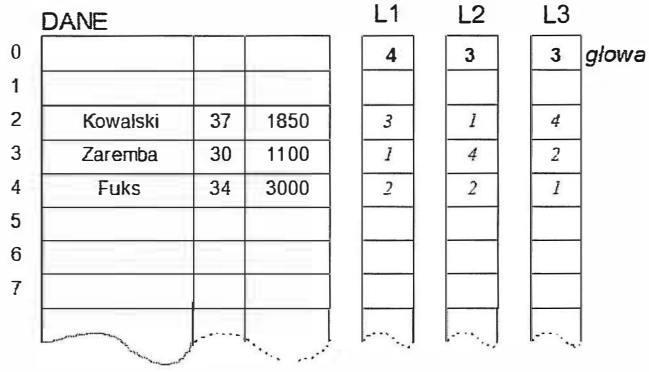
Rysunek 5.11. *Metoda „tablic równoległych" (1)*

Pierwszy element tablicy (tzn. ten z pozycji 0) pełni rolę wskaźnika początku listy. Jest to zatem zmienna typu `głowa`. Jeśli oznaczymy tablicę jako `t`, to `t[0].następny` zawiera indeks pierwszego rzeczywistego elementu listy. W naszym przykładzie jest to 3, zatem w t[3].info znajduje się pierwszy element listy — jest nim znak K. Aby dowiedzieć się, co następuje po K, musimy odczytać t[3].następny. Jest to 2 i tam też jest umieszczona kolejna litera słowa „KOTEK" — etc. Koniec listy jest zaznaczany umownie poprzez wartość -1 w polu następny.

Rozwiązanie to można uznać za eleganckie i elastyczne. Dopisanie funkcji, które obsługują taką strukturę danych, nie jest trudne. Występuje tu pełna analogia pomiędzy już wcześniej przedstawionymi funkcjami (np. obsługującymi listy jednokierunkowe), dlatego też zadanie ewentualnego opracowania ich pozostawiam Czytelnikowi.

Należy przy okazji zwrócić uwagę na jedną niedogodność: mamy tu do czynienia z bardzo ścisłym połączeniem samej „gołej" informacji z komórkami, które symulują wskaźniki. O ile w przypadku list był to zabieg niezbędny, to przy wykorzystaniu tablic możemy bez wahania oddzielić te dwie rzeczy. Inaczej rzecz ujmując, dobrze byłoby dysponować osobną tablicą na dane i osobną na wskaźniki. Dlaczego jednak nie pójść dalej i nie używać kilku tablic na wskaźniki?! Zbliżylibyśmy się wówczas do wersji zaprezentowanej na rysunku 5.8, otrzymując jednak o wiele prostsze w realizacji zadanie.

Na rysunku 5.12 jest przedstawiona minibaza danych zgrupowana w wyodrębnionej tablicy danych.



Rysunek 5.12. *Metoda „tablic równoległych" (2)*

Obok tablicy danych możemy zauważyć trzy osobne tablice wskaźników, które umożliwiają dostęp do danych widzianych jako listy posortowane wedle przeróżnych kryteriów. Tablica DANE zawiera rekordy z danymi, przy czym efektywne informacje zaczynają się od komórki dane[2] w górę. Dlaczego tak dziwnie? Otóż zabieg ten zapewnia nam odpowiedniość „1 do 1" tablicy danych i tablic wskaźników (L1, L2 i L3, które są w rzeczywistości zwykłymi tablicami liczb całkowitych).

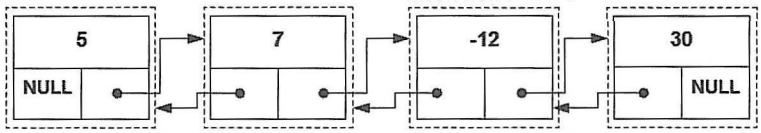
W tych tablicach bowiem komórki nr 0 i nr 1 są zarezerwowane odpowiednio na: wskaźnik początku listy i znacznik końca. Należy to rozumieć w ten sposób, że L1[0] zawierający liczbę 4 informuje nas, iż dane[4] są pierwszym rekordem na liście. A jaki jest rekord następny? Oczywiście L1[4]=2, co oznacza, że drugim rekordem na liście danych jest dane[2]. Postępując tak dalej, odtwarzamy całą listę: dane[4], dane[2], dane[3] — łatwo zauważyć, że jest to lista posortowana alfabetycznie wg nazwisk. Skąd jednak wiemy, że dane[3] jest ostatnim rekordem na liście? Otóż L1[3] zawiera 1, co stanowi wg naszej umowy znacznik końca listy.

Analogicznie postępując, możemy odkryć, że L2 jest listą posortowaną wg kodów *2*-cyfrowych, a L3 — wg zarobków. Tablicowa reprezentacja list, w której nastąpiło oddzielenie danych od wskaźników, pozwala na zapamiętanie w tym samym obszarze pamięci kilku list jednocześnie — o ile oczywiście ich elementy składowe w jakiś sposób się pokrywają. W aplikacjach, w których występuje taka sytuacja, jest to cenna właściwość przyczyniająca się do zmniejszenia zużycia pamięci. Ponadto wspomniana na samym początku tego paragrafu wada tablic, tzn. zajmowanie przez nie stałego obszaru, może być w łatwy sposób ominięta poprzez sprytne ukrycie dynamicznego zarządzania tablicą w definicji klasy (można np. wykorzystać tzw. wektory, czyli obiekty podobne do tablic, ale dynamicznie powiększające się w miarę potrzeb). Wektory są bardzo czytelnie objaśnione np. w **[Eck02]**.

Listy innych typów

Listy jednokierunkowe są bardzo wygodne w stosowaniu i zajmują stosunkowo mało pamięci. Mimo to operacje na nich niekiedy zajmują dużo czasu. Zauważyło ten fakt sporo ludzi i tym sposobem zostały wymyślone inne typy list, np.:

lista dwukierunkowa — komórka robocza zawiera wskaźniki do elementów: poprzedniego i następnego (rysunek 5.13):



Rysunek 5.13. *Lista dwukierunkowa*

- Pierwsza komórka znajdująca się w liście nie posiada swojego poprzednika; zaznaczamy to, wpisując wartość NULL do pola poprzedni.
- Ostatnia komórka znajdująca się w liście nie posiada swojego następnika; zaznaczamy to, wpisując wartość NULL do pola nastepny. Lista dwukierunkowa jest dość kosztowna, jeśli chodzi o zajętość pamięci, ale czasem szybkość działania jest ważniejsza niż małe zużycie pamięci — a właśnie szybkość działania jest zaletą listy dwukierunkowej.

Struktura wewnętrzna listy dwukierunkowej jest oczywista:

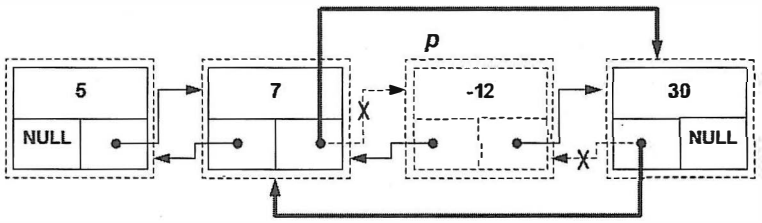
```
typedef struct rob
{
int  wartosc;
struct   rob *nastepny;
struct   rob *poprzedni;
}ELEMENT;
```

Załóżmy teraz, że podczas przeglądania elementów listy zapamiętaliśmy wskaźnik pozycji bieżącej p. (Przykładowo: szukaliśmy elementu spełniającego pewien warunek i na wskaźniku p nasze poszukiwania zakończyły się sukcesem). Jak usunąć element p z listy? Jak pamiętamy

z paragrafów poprzednich, do prawidłowego wykonania tej operacji niezbędna była znajomość wskaźników przed i po, wskazujących odpowiednio na komórki poprzednią i następną. W przypadku listy dwukierunkowej w komórce wskazywanej przez p te dwie informacje już się znajdują i wystarczy tylko po nie sięgnąć:

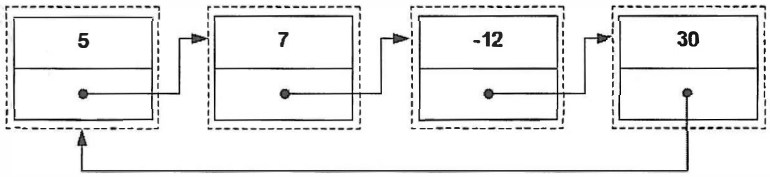
```
void usun2kier(ELEMENT *p)
{
if(p->poprzedni!=NULL)  // nie jest to element pierwszy
   p->poprzedni->nastepny=p->nastepny;
if(p->nastepny!=NULL)   // nie jest to element ostatni
   p->nastepny->poprzedni=p->poprzedni;
}
```

W zależności od konkretnych potrzeb można element p fizycznie usunąć z pamięci przez instrukcję delete lub też go w niej pozostawić do ewentualnych innych celów. Rysunek 5.14 jest odbiciem procedury usun2kier (potrzebne modyfikacje wskaźników są zaznaczone linią pogrubioną).



Rysunek 5.14. *Usuwanie danych z listy dwukierunkowej*

lista cykliczna — patrz rysunek 5.15 — jest zamknięta w pierścień: wskaźnik ostatniego elementu wskazuje „pierwszy" element.



Rysunek 5.15. *Lista cykliczna*

Pewien element określany jest jako „pierwszy" raczej umownie i służy wyłącznie do wejścia w „magiczny" krąg wskaźników listy cyklicznej.

Każda z przedstawionych powyżej list ma swoje wady i zalety. Celem tej prezentacji było ukazanie istniejących rozwiązań, zadaniem zaś Czytelnika będzie wybranie jednego z nich podczas realizacji swojego programu.

Stos

Stos jest ważnym pojęciem w informatyce i — w szczególności — w oprogramowaniu systemowym[12]. Choć takie zdanie brzmi bardzo groźnie, to chciałbym zapewnić, że nie kryje się za nim nic strasznego. Krótko mówiąc, jest to struktura danych, która ułatwia rozwiązanie wielu problemów natury algorytmicznej i w tę właśnie stronę wspólnie będziemy zdążać. Zanim dojdziemy do zastosowań stosu, spróbujmy go jednak zaimplementować w języku C++.

[12] Tym ostatnim zagadnieniem w tej książce nie będziemy się zajmowali.

Zasada działania stosu

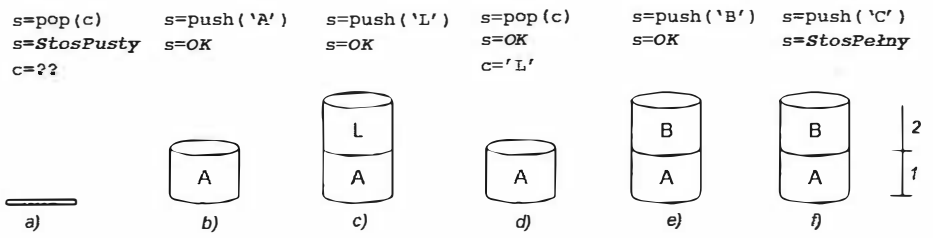
Stos jest strukturą danych, do której dostęp jest możliwy tylko od strony tzw. wierzchołka, czyli pierwszego wolnego miejsca znajdującego się na nim. Z tego też względu jego zasada działania jest bardzo często określana za pomocą angielskiego skrótu LIFO: *Last In First Out*, co w wolnym tłumaczeniu oznacza „ostatni będą pierwszymi". Do odkładania danych na wierzchołek stosu służy zwyczajowo funkcja o nazwie `push(X)`, gdzie `X` jest daną pewnego typu. Może to być dowolna zmienna prosta lub złożona: liczba, znak, rekord itd.

Podobnie, aby pobrać element ze stosu, używa się funkcji o nazwie `pop(X)`, która załadowuje zmienną `X` daną zdjętą z wierzchołka stosu. Obie te podstawowe funkcje oprócz swojego głównego zadania, które zostało wzmiankowane wyżej, zwracają jeszcze kod błędu[13]. Jest to stała typu całkowitego, która informuje programistę, czy czasem nie nastąpiła sytuacja niepożądana, np. próba zdjęcia czegoś ze stosu w momencie, gdy był on już pusty, lub też próba odłożenia na nim kolejnej danej w sytuacji, gdy brakowało w nim miejsca (brak pamięci). Programowe realizacje stosu różnią się między sobą drobnymi szczegółami (ostateczne słowo w końcu ma programista!), ale ogólna koncepcja jest zbliżona do opisanej wyżej.

Zasada działania stosu może zostać zatem podsumowana następującymi, prostymi regułami:

- Po wykonaniu operacji `push(X)` element `X` sam staje się nowym wierzchołkiem stosu, przykrywając poprzedni wierzchołek (jeśli oczywiście coś na stosie już było).
- Jedynym bezpośrednio dostępnym elementem stosu jest jego wierzchołek.
- Próba wstawienia czegoś na pełny stos powinna zakończyć się błędem.
- Próba pobrania elementu z pustego stosu powinna zakończyć się błędem.

Dla dokładniejszego zobrazowania zasady działania stosu i powyższych reguł proszę prześledzić kilka operacji dokonanych na nim i efekt ich działania — patrz rysunek 5.16.



Rysunek 5.16. *Stos i podstawowe operacje na nim*

Rysunek przedstawia stos służący do zapamiętywania znaków. Stałe symboliczne `StosPusty`, `OK` i `StosPełny` są zdefiniowane przez programistę w module zawierającym deklarację stosu jako struktury danych. Wyrażają się one w wartościach typu całkowitego (co akurat nie ma specjalnego znaczenia). Nasz stos ma pojemność dwóch elementów, co jest oczywiście absurdalne, ale zostało przyjęte na użytek naszego przykładu, aby zilustrować efekt przepełnienia.

Symboliczny stos znajdujący się pod każdą z sześciu grup instrukcji ukazuje zawsze stan po wykonaniu „swojej" grupy instrukcji. Jak można łatwo zauważyć, operacje na stosie przebiegały pomyślnie do momentu osiągnięcia jego całkowitej pojemności; wówczas stos zasygnalizował sytuację błędną.

Jakie są typowe realizacje stosu? Najpopularniejszym sposobem jest użycie tablicy i zarezerwowanie jednej zmiennej w celu zapamiętania liczby danych aktualnie znajdujących się na stosie. Jest to dokładnie taki sam pomysł, jak ten zaprezentowany na rysunku 5.10, z jednym

[13] Nie jest to bynajmniej obowiązkowe!

zastrzeżeniem: mimo iż wiemy, jak stos jest zbudowany od środka, nie zezwalamy nikomu na bezpośredni dostęp do niego. Wszelkie operacje odkładania i zdejmowania danych ze stosu muszą się odbywać za pośrednictwem metod push i pop. Jeśli zdecydujemy się na zamknięcie danych i funkcji służących do ich obsługi w postaci klasy[14], to wówczas automatycznie uzyskamy bezpieczeństwo użytkowania — zapewni je sama koncepcja programowania zorientowanego obiektowo. Taki właśnie sposób postępowania obierzemy.

Możliwych sposobów realizacji stosu jest mnóstwo; wynika to z faktu, iż ta struktura danych nadaje się doskonale do ilustracji wielu zagadnień algorytmicznych. Dla naszych potrzeb ograniczymy się do bardzo prostej realizacji tablicowej, która powinna być uważana raczej za punkt wyjścia niż za gotową implementację.

W związku z założonym powyżej celowym uproszczeniem definicja klasy STOS jest bardzo krótka:


Listing

stos.h

```
const int DLUGOSC_MAX=2;
enum stan {OK=0, STOS_PELNY=1, STOS_PUSTY=2};

template <class TypPodst>  class STOS
{
public:                          // szczyt = pierwsza WOLNA komórka
   STOS() { szczyt=0;}           // konstruktor
   void clear() { szczyt=0;} // zerowanie stosu
   int push(TypPodst x);
   int pop (TypPodst &w);
   int StanStosu();
private:
   TypPodst t[DLUGOSC_MAX];   // stos = t[0]...t[DLUGOSC_MAX-1]
   int szczyt;
}; // koniec definicji klasy STOS
```

Nasz stos będzie mógł potencjalnie służyć do przechowywania danych wszelakiego rodzaju, z tego też powodu celowe wydało się zadeklarowanie go w postaci tzw. *klasy szablonowej*, co zostało zaznaczone przez słowo kluczowe template.

Idea klasy szablonowej polega na stworzeniu wzorcowego kodu, w którym typ pewnych danych (zmiennych, wartości zwracanych przez funkcje itp.) nie zostaje precyzyjnie określony, ale jest zastąpiony pewną stałą symboliczną. W naszym przypadku jest to stała TypPodst.

Zaletą tego typu postępowania jest dość duża uniwersalność tworzonej klasy, gdyż dopiero we właściwym kodzie (np. funkcji main) określamy, że np. TypPodst powinien zostać zamieniony na np. float, char* lub jakiś złożony typ strukturalny. Wadą klasy szablonowej jest jednak dość dziwna składnia, której musimy się trzymać, chcąc zdefiniować jej metody. O ile jeszcze definicje znajdują się w ciele klasy (tzn. pomiędzy jej nawiasami klamrowymi), to składnia przypomina normalny kod C++. W momencie jednak gdy chcemy definicje metody umieścić poza klasą (tzn. poza nawiasami klamrowymi zamykającymi jej definicję), to otrzymujemy tego rodzaju dziwoląg[15]:

[14] Czyli dokonamy tzw. *hermetyzacji*.

[15] Oczywiście zawsze można się pocieszać, że ewentualnie mogłoby to zostać jeszcze bardziej skomplikowane. Ale żarty na bok, powyższe problemy wynikają z prostego faktu: C++ należy do grupy języków, których kompilatory muszą znać precyzyjnie typ danych, jakie wchodzą w grę podczas programowania, stąd też każdy zabieg służący uczynieniu go pozornie nieczułym na typy danych musi być nieco sztuczny. Warto wspomnieć przy okazji, że istnieją języki z zasady pozbawione pojęcia typu danych, np. *Smaltalk-80* (jest to język obiektowy o zupełnie innej filozofii niż C++, który wydaje się przy nim swego rodzaju *asemblerem obiektowym*).

```
template <class TypPodst> int STOS<TypPodst>::push(TypPodst x)
{
if (szczyt<DLUGOSC_MAX)
   {
   t[szczyt++]=x;
   return (OK);
   }
else
   return (STOS_PELNY);
}
```

Metoda push, bowiem to jej kod mamy przed oczami, jest bardzo prosta, co jest zresztą cechą wszelkich realizacji tablicowych. Nowy element x (jakikolwiek by był jego typ) jest zapisywany na szczycie stosu, który jest wskazywany w prywatnej dla klasy zmiennej szczyt. Następnie wartość szczytu stosu jest inkrementowana — to wszystko pod warunkiem, że stos nie jest już zapełniony!

Metoda pop wykonuje odwrotne zadanie — zdejmowany ze stosu element jest zapamiętywany w zmiennej w (przekazanej w wywołaniu przez referencję); zmienna szczyt jest oczywiście dekrementowana — pod warunkiem, że stos nie był pusty (z próżnego to nawet i programista nie... naleje?):

```
template <class TypPodst> int STOS<TypPodst>::pop(TypPodst &w)
{ //„w" zostanie „załadowane" wartością zdjętą ze stosu
if (szczyt>0)
  {
  w=t[--szczyt];
  return (OK);
  }else
      return (STOS_PUSTY);
}
```

Od czasu do czasu może zajść potrzeba zbadania stanu stosu bez wykonywania na nim żadnych operacji. Użyteczna może być wówczas następująca funkcja:

```
template <class TypPodst> int STOS<TypPodst>::StanStosu()
{
//zwraca informacje o stanie stosu
switch(szczyt)
   {
   case 0            :return (STOS_PUSTY);
   case DLUGOSC_MAX  :return (STOS_PELNY);
   default           :return (OK);
   }
}
```

Jakie są inne możliwe sposoby zdefiniowania stosu? Nie powinno dla nikogo stanowić niespodzianki, że logicznym następstwem użycia tablic są struktury dynamiczne, np. listy. Bezpośrednie wbudowanie listy do stosu, zamiast na przykład tablicy t, tak jak wyżej, byłoby jednakże nieefektywne — warto poświęcić odrobinę wolnego czasu i stworzyć osobną klasę od samego początku.

Chwilę uwagi należy jeszcze poświęcić wykorzystaniu stosu. Zasadniczą kwestią jest składnia użycia klasy szablonowej w funkcji main. Deklaracja stosu s, który ma posłużyć do przechowywania zmiennych typu np. char*, dokonuje się poprzez np.:

```
STOS<char*> s;
```

Podobnie dzieje się w przypadku każdego innego typu danych, wystarczy go tylko umieścić w nawiasach < >.


Listing

stos.cpp

```
int main()
{
STOS<char*>   s1; //stos do przechowywania tekstów
STOS<double>  s2; //stos do przechowywania liczb zmiennopozycyjnych
cout << "Odkładam na stos s1:\n";
for(int i=0; i<3;i++)
   {
   if (s1.StanStosu() != STOS_PELNY)
      {
      cout << "Próbuję włożyć:" << tab1[i] << ".";
      s1.push(tab1[i]);
      } else
      {
      cout << "Stos pełny, stop!\n";
      break;
      }
   }
for(int i=0; i<3;i++)
   {
   char *z;
   if (s1.pop(z)==OK)
      cout << "\nZdejmuję ze stosu s1: "<< z << endl;
   else
      {
      cout << "Stos pusty, koniec!\n";
      break;
     }
   }
}
```

Oto wyniki naszego programu (wersja na *ftp* jest nieco bardziej rozbudowana):

```
Odkładam na stos s1:
Próbuję włożyć:Ala. Próbuję włożyć:ma. Stos pełny, stop!
Zdejmuję ze stosu s1: ma
Zdejmuję ze stosu s1: Ala
Stos pusty, koniec!
```

Kolejki FIFO

Kolejki typu FIFO(ang. *First In First Out*, co w wolnym tłumaczeniu oznacza: *kto pierwszy, ten lepszy*) będą kolejnym omawianym typem danych. Struktura ta działa na zasadzie obsługi ogonka ludzi przed kasą sklepową, oczywiście zakładając brak wyjątków typu klient uprzywilejowany.

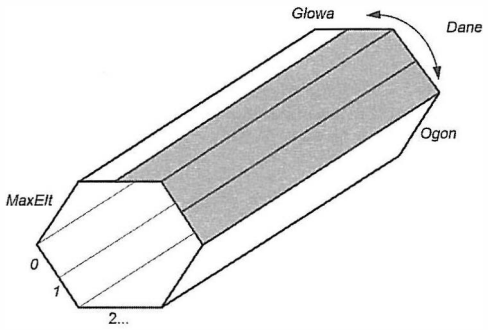
Podobnie jak i stos, jest to struktura danych o dostępie ograniczonym. Zakłada ona dwie podstawowe operacje:

- wstaw — wprowadź dane (klienta) na ogon kolejki;
- obsłuż — usuń dane (klienta) z czoła kolejki.

W porównaniu ze stosem kolejki są rzadziej stosowane w praktyce programowania. Pewne zagadnienia natury algorytmicznej dają się jednak relatywnie łatwo rozwiązywać właśnie przy użyciu tej struktury danych i to jest głównie powód niniejszej prezentacji.

Jak to zwykle bywa, możliwych implementacji kolejek jest co najmniej kilka. Realizacja efektywna czasowo za pomocą list jednokierunkowych jest zbliżona do tej wzmiankowanej przy okazji omawiania stosu. Nie będzie ona stanowiła przedmiotu naszej dyskusji, ograniczymy się jedynie do prostej implementacji tablicowej (zbliżonej zresztą ideowo do tablicowej realizacji stosu).

Tablicowa implementacja kolejki FIFO jest wyjaśniona na rysunku 5.17.



Rysunek 5.17. *Tablicowa realizacja kolejki FIFO — koncepcja*

Zawartość kolejki stanowią elementy pomiędzy *głową* i *ogonem* — te dwie zmienne będą oczywiście zmiennymi prywatnymi klasy FIFO. Dojście nowego elementu do kolejki wiąże się z inkrementacją zmiennej ogon i dopisaniem elementu u dołu „szarej strefy". Oczywiście w pewnym momencie może się okazać, że ogon osiągnął koniec tablicy — wówczas pojęcie dołu odwróci się i to dosłownie!

W takim przypadku cała szara strefa zawinie się wokół elementu zerowego tablicy. Obsługa „klienta" będącego aktualnie na początku kolejki wiąże się z zapamiętaniem elementu czołowego i z inkrementacją zmiennej głowa. Trzeba się ponadto umówić, jak interpretować stwierdzenie, że kolejka jest pusta?

Zamiast komplikować sobie życie specjalnymi testami zawartości tablicy, można po prostu założyć, że gdy głowa=ogon, to kolejka jest pusta. Tym samym trzeba zarezerwować jeden dodatkowy element tablicy, który nigdy nie będzie wykorzystany z uwagi na sposób działania metody wstaw. Po tych rozbudowanych wyjaśnieniach programowa realizacja kolejki nie powinna już stanowić żadnej niespodzianki dla Czytelnika:

Listing

kolejka.h

```
enum stan {OK, BLAD};
template <class TypPodst>  class FIFO
{
private:
   TypPodst *t;
   int glowa,ogon,MaxElt;
public:
   FIFO(int n)
   {
   MaxElt=n;
   glowa=ogon=0;
   t=new TypPodst[MaxElt+1];
   }

   void wstaw(TypPodst x)
   {
   t[ogon++]=x;
   if(ogon>MaxElt) ogon=0;
   }

   int obsluz(TypPodst &w)
   {

   if (glowa==ogon)
      return BLAD; // informacja o błędzie operacji
      w=t[glowa++];
```

```
        if(glowa>MaxElt) glowa=0;
            return OK;
    }
    bool pusta()
    {// czy kolejka jest pusta?
    if (glowa==ogon)
       return true; // kolejka pusta
    else
       return false;
    }
};
```

Podobnie jak w przypadku stosu zdefiniowaliśmy nowy typ danych w postaci klasy szablonowej. Umożliwia to łatwe definiowanie rozmaitych kolejek obsługujących różnorodne typy danych. Definicja klasy FIFO nie jest kompletna: brakuje w niej na przykład jawnego destruktora, ponadto kontrola operacji mogłaby być nieco bardziej rozbudowana... Te dodatki są jednak pozostawione Czytelnikowi jako proste ćwiczenie programistyczne.

Popatrzmy, jak wygląda w praktyce korzystanie z nowej struktury danych:


Listing

kolejka.cpp

```
#include <iostream>
using namespace std;
#include "kolejka.h"
static char *tab[]={"Kowalska","Fronczak","Becki","Pigwa"};
int main()
{
int i;
FIFO<char*> kolejka(5); // kolejka 5-osobowa
for(i=0; i<4; i++)
   kolejka.wstaw(tab[i]);
char *s;
for(i=0; i<5;i++)
   {
   int res=kolejka.obsluz(s);
   if (res==OK)
      cout <<  "Obsłużony został klient: "<<s<<endl;
   else
      cout <<  "Kolejka pusta!\n";
   }
}
```

Zasada obsługi kolejki (w krajach cywilizowanych) polega na uwzględnianiu w pierwszej kolejności osób, które zjawiły się na samym początku. Tak też jest w naszym przykładzie, o czym najdobitniej świadczą rezultaty wykonania programu:

```
Obsłużony został klient: Kowalska
Obsłużony został klient: Fronczak
Obsłużony został klient: Becki
Obsłużony został klient: Pigwa
Kolejka pusta!
```

Sterty i kolejki priorytetowe

W paragrafach poprzednich mieliśmy okazję zapoznać się m.in. z dwiema strukturami danych stanowiącymi swoje ideowe skrajności :

- *Kolejką* — usuwało się z niej w pierwszej kolejności „najstarszy" element.
- *Stosem* — usuwało się z niego w pierwszej kolejności „najmłodszy" element.

Były to struktury danych służące z zasady do zapamiętywania danych nieuporządkowanych, co zdecydowanie upraszczało wszelkie operacje! Kolejna zaś struktura danych, którą będziemy się zajmować — kolejki priorytetowe — działa wg zupełnie odmiennej filozofii, choć zachowuje ciągle zaletę operowania nieuporządkowanym zbiorem danych. (Stwierdzenie o nieuporządkowaniu jest prawdą w sensie globalnym — lokalnie fragmenty sterty są w pewien szczególny sposób uporządkowane, o czym przekonamy się już za moment). Dwie podstawowe operacje wykonywane na kolejkach priorytetowych polegają na:

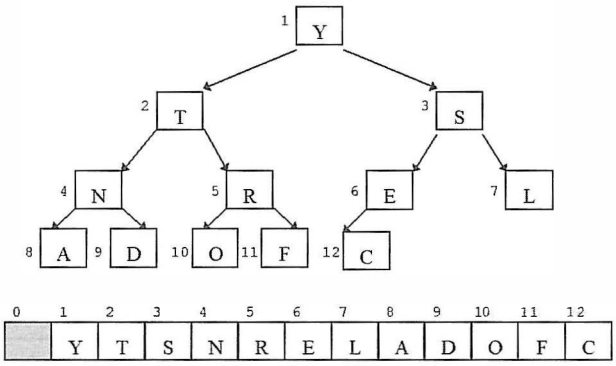
- wstawianiu nowego elementu;
- usuwaniu największego elementu [16].

Jednym z najłatwiejszych sposobów realizacji kolejek priorytetowych jest użycie struktury danych zwanej *stertą* (ang. *heap* — inna spotykana polska nazwa to stóg lub kopiec). O tej strukturze danych napomknęliśmy już przy okazji sortowania (patrz rozdział 4.), obecnie ten temat zostanie rozwinięty w nieco innym ujęciu.

Sterta jest swego rodzaju drzewem binarnym, które ze względu na szczególne własności warto omówić osobno. (Kwestia terminologiczna: zarówno sterta, jak i kolejki priorytetowe są strukturami danych, jednakże tylko kolejka priorytetowa ma charakter czysto abstrakcyjny).

Uporządkowanie elementów wchodzących w skład sterty można zaobserwować na rysunku 5.18 przedstawiającym *12*-elementową stertę. Jest to również przykład tzw. *kompletnego drzewa binarnego*. Stosując pewne uproszczenie definicyjne, można także powiedzieć, iż jest to „drzewo bez dziur". Jeśli spojrzeć na numery przypisane węzłom drzewa, to widać, że ich kolejność definiuje pewien charakterystyczny porządek wypełniania go: pod istniejące węzły przywieszamy maksymalnie po dwa nowe aż do ulokowania wszystkich *12* elementów. Można to oczywiście wyrazić nieco bardziej formalnie, ale zapewniam, że zdecydowanie mniej zrozumiale.

Rysunek 5.18. *Tablicowa realizacja kolejki FIFO — przykład*



Liniowy porządek wypełniania drzewa automatycznie sugeruje sposób jego składowania w tablicy[17]:

- Wierzchołek (czyli de facto korzeń, bo drzewo jest odwrócone) = 1.
- Lewy potomek *i*-tego węzła jest schowany pod indeksem `2*i`.
- Prawy potomek *i*-tego węzła jest schowany pod indeksem `2*i+1`.

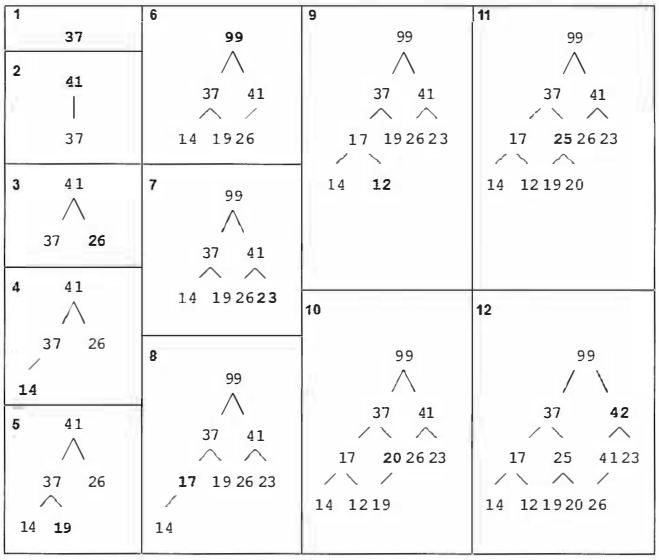
Dany węzeł może mieć od 0 do 2 potomków.

[16] Jeśli w kolejce priorytetowej będą składowane rekordy o pewnej strukturze, to jednym z pól rekordu będzie jego priorytet wyrażony w postaci liczby całkowitej dodatniej lub ujemnej. W naszych przykładach dla prostoty ograniczymy się tylko do przypadku składowania liczb całkowitych.

[17] Zerowa komórka tablicy nie jest używana do składowania danych.

Powyżej zdefiniowaliśmy sposób składowania danych, nic jednak nie powiedzieliśmy o zależnościach istniejących pomiędzy nimi. Otóż cechą charakterystyczną sterty jest to, iż *wartość każdego węzła jest większa*[18] *od wartości węzłów jego dwóch potomków — jeśli oczywiście istnieją*. Sposób organizacji drzewa (jak również w konsekwencji tablicy) ułatwia operacje wstawiania i usuwania elementów. Możemy bowiem nowy element bez problemu dopisać na końcu tablicy (co oczywiście zburzy nam ład wcześniej tam panujący), następnie za pomocą dość prostych modyfikacji tablicy przywrócić z powrotem tablicy (drzewu) własności sterty. Popatrzmy na przykładzie, w jaki sposób jest konstruowana sterta z elementów: 37, 41, 26, 14, 19, 99, 23, 17, 12, 20, 25 i 42 — dołączanych sukcesywnie do drzewa. Cały proces jest pokazany na rysunku 5.19.

Rysunek 5.19. *Konstrukcja sterty na przykładzie*

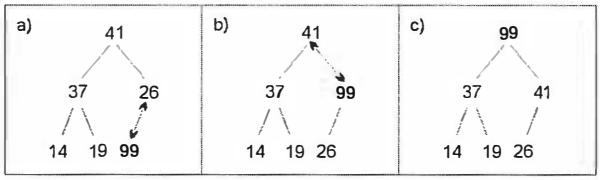


Na rysunku widzimy, gdzie wędruje nowy — zaznaczony wytłuszczoną czcionką — element. Poprzez porównanie z etapem poprzednim łatwo zauważamy modyfikacje struktury drzewa. Załóżmy, że dokładamy na koniec drzewa liczbę 99 (patrz etap 5.). Drzewo ma już 5 elementów, zatem nowy powędruje na miejsce nr 6 w tablicy — pod 26. W tym momencie jednak zostaje złamana zasada konstrukcji sterty: potomek węzła ma większą wartość niż sam węzeł, do którego jest on przywieszony! Co możemy zrobić, aby przywrócić porządek? W tym miejscu wystarczy zwyczajnie zamienić 26 i 99 miejscami, aby wszystko się *lokalnie* uspokoiło. Zauważmy, że taka lokalna zamiana przywraca porządek jedynie na aktualnie analizowanym poziomie — burząc go być może na następnym! Zatem, aby w całej stercie zapanował porządek, należy proces zamieniania kontynuować[19] w górę aż do osiągnięcia korzenia. (W naszym przykładzie konieczna będzie jeszcze zamiana liczb 99 i 41). Programową realizację opisanej powyżej czynności wykona procedura o nazwie `DoGory`. Opisaną sytuację ilustruje rysunek 5.20.

Teraz, gdy już wiemy, CZYM jest sterta i JAK się ją tworzy, pora wyjaśnić wreszcie, dlaczego sterta umożliwia łatwe tworzenie kolejek priorytetowych. Wiemy już, że istotną cechę wyróżniającą kolejki priorytetowe od innych podobnych struktur danych stanowi to, że pierwszym obsługiwanym „klientem" jest ten, który ma największą wartość (lub też, w przypadku rekordów, największą wartość pewnego wybranego pola).

[18] Spotyka się również implementacje, w których jest to wartość nie większa.

[19] Jeśli oczywiście zachodzi potrzeba.



Rysunek 5.20. *Poprawne wstawianie nowego elementu do sterty*

Jeśli trzymać się ciągle analogii kolejki do kasy sklepowej, to można by powiedzieć, że wszyscy ustawiają się elegancko na końcu „ogonka", ale to kasjerka patrzy klientom w oczy i wybiera do obsługi tych najbardziej uprzywilejowanych (ewentualnie najprzystojniejszych...).

W przypadku list i zwykłych tablic problemem byłoby znalezienie właśnie tego największego elementu — należałoby w tym celu dokonać przeszukania, które zajmuje czas proporcjonalny do N (wielkości tablicy lub listy). A jak to wygląda w naszym przypadku? Spójrzmy raz jeszcze na tablicę z rysunku 5.18 dla upewnienia się: TAK, my w ogóle nie musimy szukać największego elementu, bowiem z założenia znajduje się on w komórce tablicy o indeksie `1`!

Po euforii powinna jednak przyjść chwila zastanowienia: a co z wstawianiem? Elementy są co prawda zawsze dokładane na koniec, ale potem zawsze trzeba wywołać procedurę `DoGory`, która przywróci stercie zachwiany (ewentualnie) porządek. Czy czasem owa procedura nie jest na tyle kosztowna, że ewentualny zysk z użycia sterty nie jest już tak oczywisty? Na szczęście okazuje się, że nie. Wszelkie algorytmy operujące na stercie wykonują się wprost proporcjonalnie do długości drogi odwiedzanej podczas przechodzenia przez drzewo binarne reprezentujące stertę. Co można powiedzieć o tej długości, wiedząc, że drzewo binarne jest kompletne? Na przykład to, iż dowolny wierzchołek jest odległy od wierzchołka (korzenia) o co najwyżej $log_2 N$ węzłów! Z tego właśnie powodu algorytmy stertowe wykonują się na ogół w czasie „logarytmicznym". Jest to dobry wynik, decydujący często o użyciu tej, a nie innej struktury danych.

Po tak długim wstępie warto wreszcie zaprezentować kilka linii kodu w C++, które przemówią lepiej niż rozwlekłe wyjaśnienia. Definicja klasy `Sterta`[20] jest następująca:

Listing

sterta.h

```
#include <iostream>
using namespace std;

class Sterta
{
public:
  Sterta(int nMax)
   {
   t=new int[nMax+1];
   L=0;
   }
  void wstaw(int x);
  int  obsluz();
  void DoGory();
  void NaDol();
  void pisz();
private:
  int *t;
  int L; //liczba elementów
}; // koniec definicji klasy Sterta
```

[20] Dla uproszczenia podany zostanie przykład dla sterty liczb całkowitych.

Konstruktor klasy tworzy tablicę, w której będą zapamiętywane elementy — t[0] jest nieużywane, stąd deklaracja tablicy o rozmiarze nMax+1, a nie nMax (jest to szczegół implementacyjny ukryty przed użytkownikiem).

Na początek zajmijmy się wstawieniem nowego elementu do sterty:

```
void Sterta::wstaw(int x)
{
t[++L]=x;
DoGory();
}
```

Procedura DoGory była już wcześniej wzmiankowana: zajmuje się ona przywróceniem porządku w stercie po dołączeniu na koniec tablicy t nowego elementu.

Treść procedury DoGory nie powinna stanowić niespodzianki. Jedyną różnicą pomiędzy wskazaną na rysunku 5.20 zamianą elementów jest... jej brak! W praktyce szybsze okazuje się przesunięcie elementów w drzewie, tak aby zrobić miejsce na unoszony do góry ostatni element tablicy:

```
void Sterta::DoGory()
{
int temp=t[L];
int n=L;
while( (n!=1) && (t[n/2]<=temp) )
   {
   t[n]=t[n/2];
   n=n/2;
   }
t[n]=temp;
}
```

Jest to być może zbędna sztuczka, biorąc pod uwagę oryginalny algorytm polegający na systematycznym zamienianiu elementów ze sobą (w miarę potrzeby) podczas przechodzenia przez węzły drzewa, jednak pozwala ona nieco przyspieszyć procedurę[21].

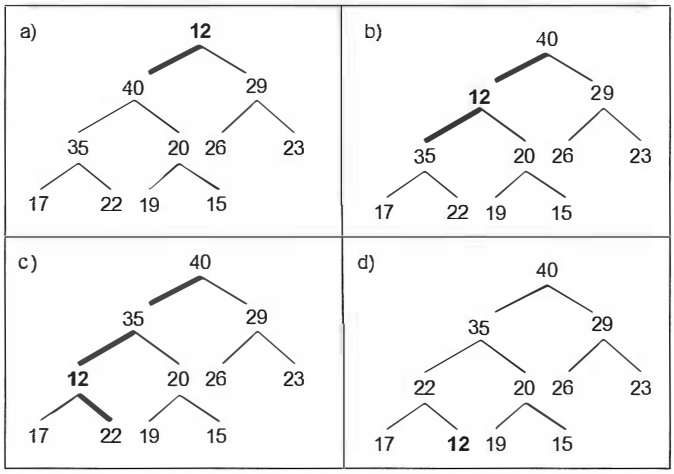
Nawiązując do kolejek priorytetowych, wspomnieliśmy, że są one łatwo implementowalne za pomocą sterty. Wstawianie „klienta" do kolejki priorytetowej (czyli sterty) na sam jej koniec zostało zrealizowane powyżej. Jak pamiętamy, pierwszym obsługiwanym „klientem" w kolejce priorytetowej był ten, który miał największą wartość — t[1]. Ponieważ po usunięciu tego elementu w tablicy robi się dziura, ostatni element tablicy wstawiamy na miejsce korzenia, dekrementujemy L i wywołujemy procedurę NaDol, która skoryguje w odpowiedni sposób stertę, której porządek mógł zostać zaburzony:

```
int Sterta::obsluz()
{
int x=t[1];
t[1]=t[L--];    // brak kontroli błędów!!!
NaDol();
return x;
}
```

(Czytelnik powinien samodzielnie rozbudować powyższą metodę, wzbogacając ją o elementarną kontrolę błędów).

Jak powinna działać procedura NaDol? Zmiana wartości w korzeniu mogła zaburzyć spokój zarówno w lewym, jak i prawym jego potomku. Nowy korzeń należy za pomocą zamiany z większym z jego potomków przesiać w dół drzewa aż do momentu znalezienia właściwego dlań miejsca. Popatrzmy na efekt zadziałania procedury NaDol wykonanej na pewnej stercie (patrz rysunek 5.21).

[21] Która oczywiście pozostanie w dalszym ciągu logarytmiczna — cudów bowiem w informatyce nie ma!



Rysunek 5.21. *Ilustracja procedury NaDol*

Element 12 został zaznaczony wytłuszczoną czcionką. Za pomocą pogrubionej kreski zaprezentowano drogę, po której zstępował element 12 w stronę swojego... miejsca ostatecznego spoczynku!

Oto jak można sposób zrealizować procedurę `NaDol`:

```
void Sterta::NaDol()
{
int i=1;
while(1)
   {
   int p=2*i;       // lewy potomek węzła 'i' to (p), prawy to (p+1)
   if(p>L)
      break;
   if(p+1<=L)       //prawy potomek niekoniecznie musi istnieć!
      if(t[p]<t[p+1]) p++; //przesuwamy się do następnego
   if(t[i]>=t[p]) break;
   int temp=t[p]; //zamiana
   t[p]=t[i];
   t[i]=temp;
   i=p;
   }
}
```

Sposób korzystania ze sterty jest zbliżony do poprzednio opisanych struktur danych i nie powinien sprawić Czytelnikowi żadnych problemów. Nieco bardziej interesujące jest ukazanie efektownego zastosowania sterty do... sortowania danych.

Wystarczy bowiem dowolną tablicę do posortowania wpierw zapamiętać w stercie, używając metody `wstaw`, a następnie zapisać ją od tyłu w miarę obsługiwania za pomocą metody `obsluz`:

```
#include "sterta.h"
int main()
{
int i, tab[14]={12,3,-12,9,34,23,1,81,45,17,9,23,11,4};
Sterta s(14);
for (i=0;i<14;i++)
s.wstaw(tab[i]);
for (i=14;i>0;i--)
tab[i-1]=s.obsluz();
cout<<"Tablica posortowana:\n";
for (i=0;i<14;i++)
cout << "  " << tab[i];
}
```

Jest to oczywiście jedno z możliwych zastosowań sterty — prosta i efektywna metoda sortowania danych, średnio zaledwie dwa razy wolniejsza od sortowania szybkiego poznanego w poprzednim rozdziale (algorytmu Quicksort).

Powyższa procedura może być jeszcze bardziej przyspieszona poprzez włączenie kodu metod `wstaw` i `obsluz` wprost do funkcji sortującej, tak aby uniknąć zbędnych i kosztownych wywołań proceduralnych. W tym przypadku zachodzi jednak potrzeba zaglądania do prywatnych informacji klasy — tablicy `t` (patrz plik *sterta.h*), zatem procedura sortująca musiałaby być funkcją zaprzyjaźnioną. Łamiemy jednak w tym momencie koncepcję programowania obiektowego (separacja prywatnego wnętrza klasy od jej zewnętrznego interfejsu)!

Jest to cena, którą płacimy za efektywność — funkcje zaprzyjaźnione zostały wprowadzone do C++ zapewne również z uwagi na użycie tego języka do programowania aplikacji wyjściowych, a nie tylko do prezentacji algorytmów (jak to jest w przypadku Pascala, który zawiera celowe mechanizmy zabezpieczające przed używaniem dziwnych sztuczek, bez których programy działałyby zbyt wolno na rzeczywistych komputerach).


Patrz także

Patrz także rozdział 4., gdzie zamieściłem bardziej zwięzłą wersję samego algorytmu sortowania przez kopcowanie.

Drzewa i ich reprezentacje

Dyskusją na temat tzw. *drzew* można by z łatwością wypełnić kilka rozdziałów. Temat jest bardzo rozległy i różnorodność aspektów związanych z drzewami znacznie utrudnia decyzję dotyczącą tego, co wybrać, a co pominąć. W ostatecznym rozrachunku zwyciężyły względy praktyczne: zostaną szczegółowo omówione te zagadnienia, które Czytelnik będzie mógł z dużym prawdopodobieństwem wykorzystać w codziennej praktyce programowania. Bardziej szczegółowe rozważania dotyczące drzew można znaleźć w zasadzie w większości książek poświęconych ogólnie strukturom danych. Ponieważ jednak te ostatnie nie są celem samym w sobie (o czym bardzo często autorzy książek o algorytmice zapominają), to wierzę, że bardziej praktyczne podejście do tematu zostanie przez większość Czytelników zaakceptowane.

Rozważania na temat drzew będą głównie ilustrowane przy pomocy najpopularniejszych i najczęściej używanych *drzew binarnych*, których użyteczność w rozwiązywaniu przeróżnych zagadnień algorytmicznych jest niezaprzeczalna.

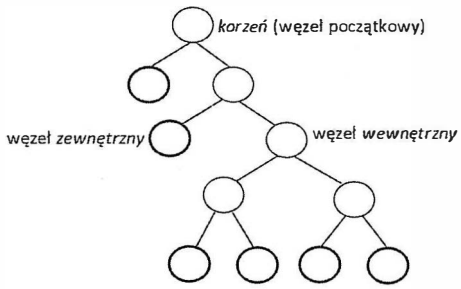
Drzewo jako reprezentacja np. struktur organizacyjnych, systemu folderów w systemie operacyjnym, genealogii rodzinnych itp. jest doskonale znane każdemu z nas. Już tak szerokie rozpowszechnienie drzew w świecie rzeczywistym sugeruje dużą użyteczność tych struktur.

Drzewo jest reprezentowane przez *zbiór węzłów*, połączonych ze sobą relacją, nazwijmy to, „rodzicielską": umawiany się, że z jednego węzła może się wywodzić kolejny węzeł potomny (jeden lub więcej).

Wyróżniony węzeł drzewa, od którego umownie wywodzą się pozostałe węzły, nazywamy *korzeniem*. Węzeł jest przedstawiany jako kółeczko lub inny kształt, w zależności od naszych potrzeb. Prosty przykład znajduje się na rysunku 5.22.

Krawędzie łączące węzły drzewa mogą być wyposażone w strzałki, ale taka notacja jest w zasadzie nadmiarowa. Zazwyczaj strzałki na rysunkach drzew są pomijane, gdyż kierunek przechodzenia wynika z zasady „rodzicielskiej", ale z drugiej strony same w sobie strzałki nie szkodzą i czasami się je pokazuje.

Węzły nieposiadające potomków nazywane są *liśćmi* lub, już bez nawiązywania do analogii biologicznych, *węzłami końcowymi* (na rysunku 5.22 przykładowy węzeł oznaczony jako węzeł zewnętrzny).



Rysunek 5.22. *Drzewo i podstawowe pojęcia z nim związane*

Wysokością węzła nazywamy długość najdłuższej ze ścieżek prowadzących od tego węzła do liści.

Głębokością węzła nazywamy długość ścieżki łączącej go z korzeniem.

Drzewo, którym od węzła musi odchodzić określona liczba potomków, zwane jest *m*-drzewem. Najbardziej znanym przykładem takiego typu drzewa są drzewa binarne, w których od danego węzła odchodzą dwa węzły potomne — lewy i prawy. Drzewo, w którym kolejność wymieniania węzłów-potomków ma dla nas znaczenie, zwane jest *uporządkowanym*.

W *m*-drzewach czasami wyróżnia się dwa rodzaje węzłów: wewnętrzne i zewnętrzne — te ostatnie nie posiadają potomków. Zwróćmy uwagę na drobny niuans terminologiczny: w *m*-drzewie „liść" jest tak naprawdę węzłem wewnętrznym (!), którego wszyscy potomkowie są zewnętrzni.

W przypadku drzew binarnych warto wiedzieć o ich właściwościach matematycznych, które ułatwiają analizę złożoności algorytmów opartych na tych strukturach:

- Drzewo binarne zawierające n węzłów wewnętrznych ma $n+1$ węzłów zewnętrznych.
- Wysokość drzewa binarnego zawierającego n węzłów wewnętrznych wynosi co najmniej $lg\ n$, a co najwyżej $n-1$.

W rozdziale 10. poznamy uogólnioną strukturę drzewiastą, zwaną grafem. Ponieważ tematowi grafów poświęcony jest wyodrębniony rozdział, nie będę tu wnikał w relacje pomiędzy grafami a drzewami, przyjmijmy po prostu, że w najbardziej ogólnym przypadku zawsze mamy do czynienia z drzewami. Jeśli w drzewie nie wyróżnimy korzenia i nie wprowadzimy ograniczeń na liczbę węzłów przylegających, to nagle zaczynamy mówić o strukturze grafowej!

Czym w praktyce są drzewa binarne? Są to struktury bardzo podobne do list jednokierunkowych, ale wzbogacone o jeszcze jeden *wymiar* (lub *kierunek*, jak kto woli).

Podstawowa komórka służąca do konstrukcji drzewa binarnego ma postać:

```
struct wezel
{
int info; // lub dowolny inny typ danych
struct wezel *lewy,*prawy;
}
```

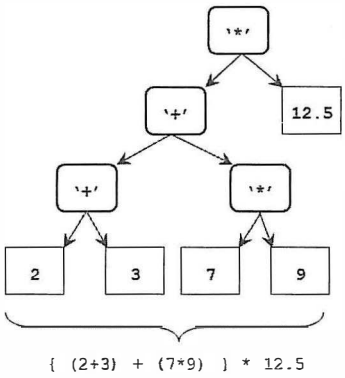
Jak łatwo jest zauważyć, w miejsce jednego wskaźnika `następny` (jak w liście jednokierunkowej) mamy do czynienia z dwoma wskaźnikami o nazwach `lewy` i `prawy`, będącymi wskaźnikami do lewej i prawej gałęzi drzewa binarnego. Jeśli nasz algorytm wymaga umiejętności „przechadzania się" w obu kierunkach (od korzenia w dół, do kolejnych węzłów potomnych i na odwrót), struktura danych powinna zawierać także wskaźnik do „przodka", od którego wywodzi się węzeł potomny, np.:

```
struct wezel
{
int info; // lub dowolny inny typ danych
struct wezel *lewy,*prawy; // węzły potomne
struct wezel *przodek;     // „ojciec"
}
```

Wskazanie na węzeł przodek jest zbędne, jeśli do przechadzania się po drzewach używamy rekurencji, która z natury rzeczy zapamiętuje, skąd nastąpiło przejście podczas przechadzania się po drzewie.

Aby dobrze zrozumieć sposób działania i użyteczność drzew binarnych, popatrzmy na rysunek 5.23.

Rysunek 5.23.
Drzewa binarne i wyrażenia arytmetyczne




```
struct węzeł
{
char operator;
float val;
węzeł *lewy, *prawy;
}
Konwencja:
operator=='0' => znaczenie ma
                 pole val(wartość)
Operator ∈ ('+', '-', '*', … etc.)
=> pole val jest ignorowane
```

Rysunek pokazuje jeden z możliwych przykładów zastosowania drzew binarnych do reprezentowania wyrażeń arytmetycznych. Do tego przykładu jeszcze powrócimy w dalszych paragrafach, na razie wystarczy ogólny opis sposobu korzystania z takiej reprezentacji. Otóż dowolne wyrażenie arytmetyczne może być zapisane w kilku odmiennych postaciach związanych z położeniem operatorów: *przed swoimi argumentami*, *po nich* oraz klasycznie *pomiędzy nimi* (jeśli oczywiście mamy do czynienia tylko z wyrażeniami dwuargumentowymi, co pozwolimy sobie tutaj dla uproszczenia przykładów założyć).

Struktura danych z tego rysunku jest zwykłym drzewem binarnym, posiadającym dwa pola przeznaczone do przechowywania danych (operator i val) oraz tradycyjne wskaźniki do lewego i prawego odgałęzienia naszego odwróconego do góry nogami drzewa. Umówimy się ponadto, że w przypadku gdy pole operator zostanie zainicjowane jakąś bezsensowną wartością (tutaj 0; nie jest to żaden znany operator), to wówczas pole val ma jakąś wartość, którą możemy uznać za sensowną. Taka dualna reprezentacja może posłużyć do łatwego rozróżnienia przy użyciu tylko jednego typu rekordów dwóch typów węzłów:

- *wartości* („listek" drzewa),
- *operatora arytmetycznego*, wiążącego w ogólnym przypadku trzy typy węzłów:
  - Lewy i prawy potomek stanowią wyrażenia.
  - Lewy potomek jest wyrażeniem, a prawy wartością.
  - Prawy potomek jest wyrażeniem, a lewy wartością.

Jeśli napiszemy odpowiednie funkcje obsługujące powyższą strukturę danych wedle przyjętych przez nas reguł postępowania, to za pomocą takiej prostej reprezentacji możemy wyrazić dowolnie skomplikowane wyrażenia arytmetyczne, wykonywać na nich operacje, różniczkować je, etc. Wszystko zależy wyłącznie od tego, co zamierzamy uzyskać — możliwych zastosowań jest dość sporo, a ponadto, jak się okaże już wkrótce, jeśli do pracy zaprzęgniemy rekurencję, to algorytmy obsługi drzew binarnych (i nie tylko) staną się bardzo proste i zrozumiałe na pierwszy rzut oka.

Czy reprezentacja za pomocą rekurencyjnych struktur danych jest optymalna? Na to pytanie można odpowiedzieć sensownie jedynie, mając przed oczami ostateczne zastosowanie implementowanego drzewa: jeśli nie dbamy zbytnio o zajętość pamięci, a zależy nam na łatwości implementacji, to reprezentacja tablicowa może okazać się nawet lepsza od tej *klasycznej*, zaprezentowanej powyżej.

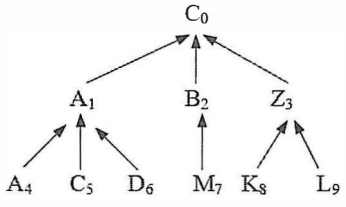
Jak zapamiętać drzewo w tablicy? Nie jest to bynajmniej dla nas problem nowy, poznaliśmy już wcześniej prostą metodę na zapamiętanie w tablicy innej drzewiastej struktury danych — sterty, podobnie jak metodę tzw. *tablic równoległych* do reprezentacji list z wieloma kryteriami sortowania. Jak widać, inteligentne użycie tablic może nam podsunąć możliwości z trudem uzyskiwane w przypadku optymalnych, listowych struktur danych.

Popatrzmy dla przykładu na implementację tablicową drzew, w których nie są zapamiętywane informacje dotyczące *potomków* danego węzła (tzn. nie interesuje nas zstępowanie w stronę liści), ale informacje o *rodzicach* danego potomka.

Terminologia używająca określeń: *ojciec*, *syn*, *potomek lewy*, *potomek prawy*, etc. jest ogólnie spotykana w książkach poświęconych strukturom drzewiastym — również anglojęzycznych. W tym miejscu warto być może przytoczyć anegdotę dotyczącą właśnie tego typu określeń, które mogą osoby nieprzyzwyczajone prowadzić do konfuzji. W 1993 roku uczestniczyłem w kursie języka angielskiego przeznaczonym dla Francuzów i prowadzonym przez przybyłą do Francji Amerykankę o dość ekstrawaganckim sposobie bycia. W trakcie kursu należało przygotować małe exposé na dowolny w zasadzie, ale techniczny, temat. Jeden z francuskich studentów omówił pewien algorytm dotyczący rozproszonych baz danych, w którym dość sporo miejsca zajmowało wyjaśnienie drzewiastej struktury danych, służącej do reprezentacji pewnych istotnych dla algorytmu danych. Terminologia, której używał do opisu drzewa, była identyczna z zaprezentowaną powyżej: ojciec, syn, potomek itp. Anglosasi są ogólnie dość uczuleni na punkcie jawnego rozróżniania form osobowych (on, ona) od bezosobowych, obejmujących w zasadzie wszystko oprócz osób (określane w sposób ogólny zaimkiem *it*). Student, o którym jest mowa, omawiał coś o charakterze bez wątpienia bezosobowym — strukturę danych, ale od czasu do czasu używał określeń zarezerwowanych normalnie dla istot ludzkich — ojciec, syn... Amerykanka słuchała jego przemowy przez dobrych kilka minut, otwierając coraz szerzej oczy, aż w końcu nie wytrzymała, wyskoczyła na środek klasy i przerwała Francuzowi: „What father? What child? Please show me where is the *zizi*[22] here!" — pokazując jednocześnie na narysowane na tablicy drzewo...

Ale wróćmy do tematu i pokażmy wreszcie obiecaną implementację drzew za pomocą tablic, tak aby uzyskać informację o węzłach ojcach. Rysunek 5.24 przedstawia drzewo służące do zapamiętywania liter (czyli pole `val` jest typu `char`).

Rysunek 5.24. *Tablicowa reprezentacja drzewa*



	syn	ojciec
0	C	0
1	A	0
2	B	0
3	Z	0
4	A	1
5	C	1
6	D	1
7	M	2
8	K	3
9	L	3

Numery znajdujące się przy węzłach mają charakter wyłącznie ilustracyjny — ich wybór jest raczej dowolny i nie podlega żadnym szczególnym regułom (chyba że sobie sami je wymyślimy na użytek konkretnej aplikacji). W ramach kolejnej konwencji umówmy się, że jeśli `ojciec[x]` jest równy `x`, to mamy do czynienia z pierwszym elementem drzewa.

Teraz, gdy już wiemy, jak reprezentować drzewa, wykorzystując dostępne w C++ (oraz w każdym nowoczesnym języku programowania) mechanizmy, popatrzmy na możliwe sposoby przechadzania się po gałęziach drzew.

[22] W ten sposób francuskie dzieci określają nieodłączny atrybut każdego mężczyzny.

Drzewa binarne i wyrażenia arytmetyczne

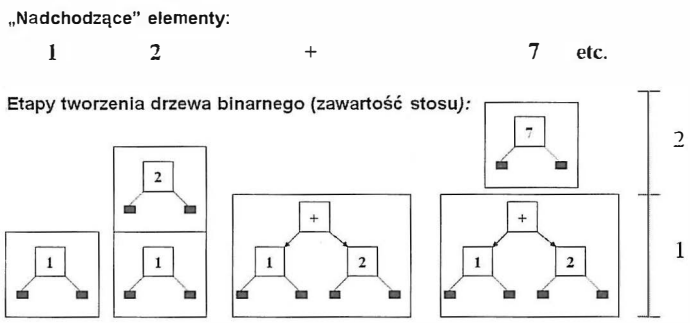
Nasze rozważania o drzewach będziemy prowadzić poprzez prezentację dość rozbudowanego przykładu, na podstawie którego zobrazowane zostaną fenomeny, z którymi programista może się zetknąć, oraz mechanizmy, z których będzie on musiał sprawnie korzystać w celu efektywnego wykorzystania nowo poznanej struktury danych.

Problematyka będzie dotyczyła kwestii zasygnalizowanej już na rysunku 5.23. Zobaczyliśmy tam, że drzewo doskonale się nadaje do reprezentacji informatycznej wyrażeń arytmetycznych, bardzo naturalnie zapamiętując nie tylko informacje zawarte w wyrażeniu (tzn. *operandy* i *operatory*), ale i ich logiczną strukturę, która daje się poglądowo przedstawić właśnie w postaci drzewa.

Przypomnijmy jeszcze raz typ komórki, który może służyć — zgodnie z ideą przedstawioną na rysunku 5.23 — do zapamiętywania zarówno operatorów (ograniczymy się tu do: +, -, * i do dzielenia wyrażonego za pomocą znaku dwukropka lub /), jak i operandów (liczb rzeczywistych).

```
struct wyrazenie
{
double val;
char op;
wyrazenie *lewy,*prawy;
};
```

Inicjacja takiej komórki determinuje późniejszą interpretację jej zawartości. Jeśli w polu `op` zapamiętamy wartość `0`, to będziemy uważali, że komórka nie jest operatorem i wartość zapamiętana w polu `val` ma sens. W odwrotnym zaś przypadku będziemy zajmowali się wyłącznie polem `op` bez zwracania uwagi na to, co znajduje się w `val`. Popatrzmy na rysunek 5.25, który ukazuje kilka pierwszych etapów tworzenia drzewa binarnego wyrażenia arytmetycznego.



Rysunek 5.25. *Tworzenie drzewa binarnego wyrażenia arytmetycznego*

Do tworzenia drzewa użyjemy dobrze nam znanego z poprzednich dyskusji stosu (patrz: strona 119). Tym razem będzie on służył do zapamiętywania wskaźników do rekordów typu `struct wyrazenie`, co implikuje jego deklarację przez `STOS<wyrazenie*> s` (Jak widać, warto było raz się pomęczyć i stworzyć stos w postaci klasy szablonowej).

Typowe wyrażenie arytmetyczne, zapisane w powszechnie używanej postaci (zwanej po polsku wrostkową), da się również przedstawić w tzw. *Odwrotnej Notacji Polskiej* (ONP, postfiksowej). Zamiast pisać `a op b`, używamy formy: `a b op`. Mówiąc krótko: operator występuje po swoich argumentach. Operacja arytmetyczna jest łatwa do odtworzenia w postaci klasycznej, jeśli wiemy, ile operandów wymaga dany operator.

Analiza wyrażenia beznawiasowego odbywa się w następujący sposób:

- Czytamy argumenty znak po znaku, odkładając je na stos.
- W momencie pojawienia się jakiegoś operatora ze stosu zdejmowana jest odpowiednia dlań liczba argumentów — wynik operacji kładziony jest na stos jako kolejny argument.

Na rysunku 5.25 możemy zaobserwować opisany wyżej proces w bardziej poglądowej formie niż powyższy suchy opis. Pierwsze dwa argumenty, 1 i 2, jako niebędące operatorami, są odkładane na stos (w programie odpowiadać to będzie stworzeniu dwóch komórek pamięci, których pola wskaźnikowe lewy i prawy są zainicjowane wartościami NULL). Trzecim elementem, który przybywa z zewnątrz, jest operator +. Tworzona jest nowa komórka pamięci, jednocześnie sam fakt nadejścia operatora prowokuje zdjęcie ze stosu dwóch argumentów, którymi są komórki zawierające liczby 1 i 2. Te komórki są doczepiane do pól wskaźnikowych komórki zawierającej operator +. Kolejnym nadchodzącym elementem jest znowu liczba 7 — jest ona odkładana na stos i proces może być kontynuowany.

W opisany powyżej sposób pracują kompilatory w momencie obliczania wyrażeń za pośrednictwem stosu. Jedyną różnicą jest to, że odkładane na stos są nie kolejne poddrzewa, ale już obliczone fragmenty dowolnie w zasadzie skomplikowanych wyrażeń arytmetycznych. Czytelnik zgodzi się chyba ze stwierdzeniem, że z punktu widzenia komputera ONP jest istotnie bardzo wygodna w użyciu[23].

Przypatrzmy się konkretnym instrukcjom w C++, które zajmują się inicjacją drzewa binarnego.


Listing

wyrazen.cpp

```
typedef struct
{
double val;
char op;
}VAL;

int main()
{
STOS<wyrazenie*> s;
// Przykład POPRAWNEJ sekwencji danych, w przypadku sekwencji
// błędnej, gdy np. zabraknie drugiego operanda, otrzymane drzewo
// będzie również bezsensowne (proszę wykonać odpowiednie próby)
VAL t[9]={{2.'0'}.{3.'0'}.{0.'+'}.{7.'0'}.{9.'0'}.{0.'*'}.{0.'+'}.
          {12.5.'0'}.{0.'*'}};
wyrazenie *x;
for(int i=0; i<9; i++)
   {
   x=new wyrazenie;
   if( (t[i].op=='*') || (t[i].op=='+')||(t[i].op=='-')
                      ||(t[i].op=='/')||(t[i].op==':') )
      x->op =t[i].op;
   else
      {
       x->val=t[i].val;
       x->op='0'; // Umowna konwencja oznaczająca wartość, a nie operator
 }
   x->lewy =NULL;
   x->prawy=NULL;
   if((t[i].op=='*') || (t[i].op=='+') || (t[i].op=='-')
                     ||(t[i].op=='/')||(t[i].op==':') )
      {
      wyrazenie *l1, *p1;
       s.pop(l1);
       s.pop(p1);
       x->lewy =l1; // „Podwiązanie" pod węzeł x
```

[23] Wbrew pozorom notacja ONP jest dość wszechstronnie stosowana w pewnych obszarach, patrz np. kalkulatory firmy *Hewlett Packard*, język Forth, język opisu stron drukarek laserowych Postscript. W pewnych kręgach jest to zatem dość znana notacja.

```
    x->prawy=pl; // „Podwiązanie" pod węzeł x
       }
    s.push(x);
    }
  pisz_infix(x);cout << "=" << oblicz(x) << endl; // Omówione dalej
  pisz_prefix(x);cout << "=" << oblicz(x) << endl; // Omówione dalej
  }
```

W powyższym listingu tablica t zawiera *poprawną* sekwencję danych, tzn. taką, która istotnie stworzy drzewo binarne mające sens. Warto odrobinę poeksperymentować z zawartością tablicy, aby zobaczyć, jak algorytm zareaguje na błędny ciąg danych. Można się spodziewać, że w przypadku np. braku drugiego operandu lub operatora otrzymane rezultaty będą również błędne — jest to prawda, ale najlepiej przekonać się o tym na własnej skórze.

Jak jednak obejrzeć zawartość drzewa, które tak pieczołowicie stworzyliśmy? Wbrew pozorom zadanie jest trywialne i sprowadza się do wykorzystania własności „topograficznych" drzewa binarnego. Sposób interpretacji formy wyrażenia (czy jest ona *infiksowa*, *prefiksowa* czy też *postfiksowa*) zależy bowiem tylko i wyłącznie od sposobu przechodzenia przez gałęzie drzewa!

Popatrzmy na realizację funkcji służącej do wypisywania drzewa w postaci klasycznej, tzn. wrostkowej. Jej działanie można wyrazić w postaci prostego algorytmu rekurencyjnego:

```
wypisz(w)
{
jeśli wyrażenie w jest liczbą. to wypisz ją;
jeśli wyrażenie w jest operatorem op. to wypisz je w kolejności:
                    (wypisz(w -> left) op wypisz(w -> right))
}
```

Realizacja programowa jest oczywiście dosłownym tłumaczeniem powyższego zapisu:

```
void pisz_infix(struct wyrazenie *w)
{ //funkcja wypisuje wyrażenie w postaci wrostkowej
if(w->op=='0') //wartość liczbowa...
   cout << w->val;
else
   {
   cout << "(";
   pisz_infix(w->lewy);
   cout << w->op;
   pisz_infix(w->prawy);
   cout <<")";
   }
}
```

W analogiczny sposób możemy zrealizować algorytm wypisujący wyrażenie w formie beznawiasowej, czyli ONP:

```
void pisz_prefix(struct wyrazenie *w)
{//funkcja wypisuje wyrażenie w postaci prefiksowej
if(w->op=='0') //wartość liczbowa...
   cout<<w->val<<" ";
else
   {
   cout << w->op << " ";
   pisz_prefix(w->lewy);
   pisz_prefix(w->prawy);
   }
}
```

Jak łatwo zauważyć, w zależności od sposobu przechadzania się po drzewie możemy w różny sposób przedstawić jego zawartość bez wykonywania jakiejkolwiek zmiany w strukturze samego drzewa!

Reprezentacja wyrażeń arytmetycznych byłaby z pewnością niekompletna, gdybyśmy nie uzupełnili jej funkcjami wykonującymi operacje na tychże wyrażeniach. Zanim jednak cokolwiek zechcemy obliczać, musimy dysponować funkcją, która sprawdzi, czy wyrażenie znajdujące się w drzewie jest prawidłowo skonstruowane, tzn. czy przykładowo nie zawiera nieznanego nam operatora arytmetycznego.

Zauważmy, że o poprawności drzewa decyduje już sam sposób jego konstruowania z użyciem stosu. Pomimo tego ułatwienia dysponowanie dodatkową funkcją sprawdzającą poprawność drzewa jest jednak mało kosztowne — dosłownie kilka linijek kodu — a użyteczność takiej dodatkowej funkcji jest oczywista.

```
int poprawne(struct wyrazenie *w)
{//czy wyrażenie jest poprawne składniowo?
if(w->op=='0')
   return 1;                  // OK, wg naszej konwencji jest to liczba
switch (w->op)
   {
   case '+':
   case '-':
   case '*':                  // to są znane operatory
   case ':':
   case '/': return ( poprawne(w->lewy) * poprawne(w->prawy) );
   default : return (0);      // błąd, operator nieznany!
   }
```

Nie będę nikogo zachęcał do zrealizowania powyższych funkcji w formie iteracyjnej — jest to oczywiście wykonalne, ale rezultat nie należy do specjalnie czytelnych i eleganckich.

Przejdźmy wreszcie do prezentacji funkcji, która zajmie się obliczeniem wartości wyrażenia arytmetycznego. Jej schemat jest bardzo zbliżony do tego zastosowanego w funkcji poprawne:

```
double oblicz(struct wyrazenie *w)
{
if(poprawne(w)) // wyrażenie poprawne?
   if (w->op=='0')
      return (w->val); //pojedyncza wartość
   else
      switch (w->op)
         {
         case '+':return oblicz(w->lewy)+oblicz(w->prawy);
         case '-':return oblicz(w->lewy)-oblicz(w->prawy);
         case '*':return oblicz(w->lewy)*oblicz(w->prawy);
         case ':':
         case '/':if(oblicz(w->prawy)!= 0)
            return (oblicz(w->lewy)/oblicz(w->prawy));
                  else
                     {
                     cerr << "\nDzielenie przez zero!\n";
                     return -1; //ułomna sygnalizacja błędów
                     }
         }
else cerr << "Błąd składni...!\n";
}
```

Dla dopełnienia prezentacji tego dość sporego kawałka kodu popatrzmy na rezultaty wykonania funkcji main:

```
 (12.5*((9*7)+(3+2)))=850
* 12.5 + * 9 7 + 3 2 =850
```

Zachęcam Czytelnika do kontynuowania eksperymentów z drzewiastymi strukturami danych, bowiem temat jest pasjonujący, a rezultaty potrafią zrobić wrażenie.

Uniwersalna struktura słownikowa

Nasze rozważania poświęcone strukturom drzewiastym zakończymy, prezentując szczegółową implementację tzw. *Uniwersalnej Struktury Słownikowej* (określanej dalej jako *USS*). Jest to dość złożony przykład wykorzystania możliwości, jakie oferują drzewa, i nawet jeśli Czytelnik nie będzie miał w praktyce okazji skorzystać z *USS*, to zawarte w tym paragrafie informacje i techniki będą mogły zostać wykorzystane przy rozwiązywaniu innych problemów, w których w grę wchodzą zbliżone kwestie.

Z uwagi na czytelność wyjaśnień wszelkie przykłady dotyczące *USS* będą tymczasowo obywały się bez poruszania zagadnienia polskich znaków diakrytycznych: *ą*, *ę*, *ć*, etc. Temat ten poruszę dopiero pod koniec tego paragrafu, gdzie zaproponuję prosty sposób rozwiązania niniejszego problemu — w istocie będą to niewielkie, wręcz kosmetyczne modyfikacje zaprezentowanych już za moment algorytmów.

Najwyższa już pora wyjaśnić właściwy temat naszych rozważań. Otóż wiele programów z różnych dziedzin, ale operujących tekstem wprowadzanym przez użytkownika, może posiadać funkcję sprawdzania poprawności ortograficznej wprowadzanych pieczołowicie informacji (patrz np. arkusze kalkulacyjne, edytory tekstu). Całkiem prawdopodobne jest, iż wielu Czytelników chciałoby móc zrealizować w swoich programach taki mały weryfikator, jednak z uwagi na znaczne skomplikowanie problemu nawet się do niego nie przymierzają. W istocie z problemem weryfikacji ortograficznej są ściśle związane następujące pytania, na które odpowiedź wcale nie jest jednoznaczna i prosta:

- Jakich struktur danych używać do reprezentacji słownika?
- Jak zapisać słownik na dysku?
- Jak wczytać słownik „bazowy" do pamięci?
- Jak uaktualniać zawartość słownika?

Konia z rzędem temu, kto bez wahania ma gotowe odpowiedzi na te pytania! Oczywiście na wszystkie naraz, bowiem nierozwiązanie na przykład problemu zapisu na dysk czyni resztę całkowicie bezużyteczną.

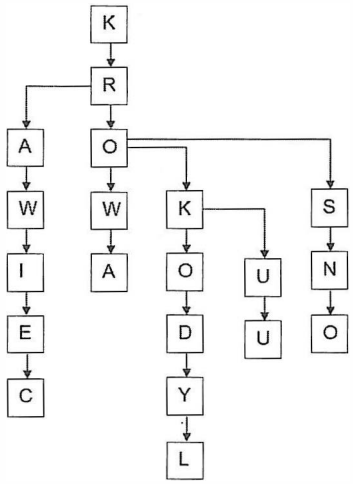
Ze wszelkiego rodzaju słownikami wiąże się również problem ich niebagatelnej objętości. O ile jeszcze możemy się łatwo pogodzić z zajętością miejsca na dysku, to w przypadku pamięci komputera decyzja już nie jest taka prosta — średniej wielkości słownik ortograficzny może z łatwością „zatkać" całą dostępną pamięć i nie pozostawić miejsca na właściwy program. No chyba że ma on wypisywać komunikat: „Out of memory"[24]... Sprawy komplikują się niepomiernie, jeśli w grę wchodzi tak bogaty język, jakim jest np. nasz ojczysty — z jego mnogimi formami deklinacyjnymi, wyjątkami od wyjątków, etc. Zapamiętanie tego wszystkiego bez odpowiedniej kompresji danych może okazać się po prostu niewykonalne.

Istnieją liczne metody kompresji danych, większość z nich ma jednak charakter archiwizacyjny — służący do przechowywania, a nie do dynamicznego operowania danymi. Marzeniem byłoby posiadanie struktury danych, która przez swoją naturę automatycznie zapewnia kompresję danych już w pamięci komputera, nie ograniczając dostępu do zapamiętanych informacji.

Prawdopodobnie wszyscy Czytelnicy domyślili się natychmiast, że *USS* należy do tego typu struktur danych.

Idea *USS* opiera się na następującej obserwacji: wiele słów posiada te same rdzenie (przedrostki), różniąc się jedynie końcówkami (przyrostkami). Przykładowo weźmy pod uwagę następującą grupę słów: KROKUS, KROSNO, KRAWIEC, KROKODYL, KRAJ. Gdyby można było zapisać je w pamięci w formie drzewa przedstawionego na rysunku 5.26, to problem kompresji mielibyśmy z głowy. Z *31* znaków do zapamiętania zrobiło nam się raptem *21*, co może nie

[24] Pol. *Brak pamięci.*



Rysunek 5.26. *Kompresja danych zaletą Uniwersalnej Struktury Słownikowej*

oszałamia, ale pozwala przypuszczać, że w przypadku rozbudowanych słowników zysk byłby jeszcze większy. Zakładamy oczywiście, że w słowniku będą zapamiętywane w dużej części serie słów zaczynających się od tych samych liter — czyli przykładowo pełne odmiany rzeczowników, etc.

Pora już na przedstawienie owej tajemniczej *USS* w szczegółach. Jej realizacja jest nieco przewrotna, bowiem zbędne staje się zapamiętywanie słów i ich fragmentów, a pomimo tego cel i tak zostaje osiągnięty!

Program zaprezentuję w szczegółowo skomentowanych fragmentach. Oto pierwszy z nich zawierający programową realizację *USS*:

Listing

uss.cpp

```
const int n=29;

typedef struct slownik
   {
   struct slownik *t[n];
   } USS, *USS_PTR;
```

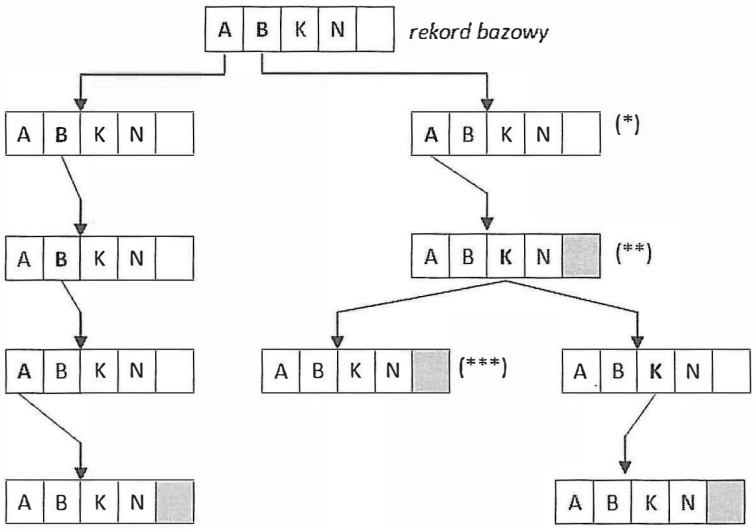
Mamy oto typową dla C++ deklarację typu rekurencyjnego, którego jedynym elementem jest tablica wskaźników do tegoż właśnie typu. (Tak, zdaję sobie sprawę, iż brzmi to okropnie). Literze *a* (lub *A*) odpowiada komórka `t[0]`, analogicznie literom *z* (lub *Z*) komórka `t[25]`. Dodatkowe komórki pamięci będą służyły znakom specjalnym, które nie należą do podstawowych liter alfabetu, ale dość często wchodzą w skład słów (np. myślnik, polskie znaki diakrytyczne itp.).

W celu oszczędności miejsca słowa będą zapamiętywane już w postaci przetransformowanej na duże litery. Słowo *odpowiada* jest tu bardzo charakterystyczne, bowiem słowa nie są w *USS* zapamiętywane bezpośrednio.

Uwaga

Zapętlenie wskaźnika `t[n-1]` *do swojej własnej tablicy oznacza znacznik końca słowa.*

Dokładną zasadę działania *USS* wyjaśnimy na przykładzie zamieszczonym na rysunku 5.27.



Rysunek 5.27. *Reprezentacja słów w USS*

Założeniem przyjętym podczas analizy niech będzie ograniczenie liczby liter alfabetu do *4*: A, B, K, N. *USS* zawiera tablicę t o rozmiarze 5: ostatnia komórka służy jako znacznik końca słowa. Jeśli wskaźnik w t[4] wskazuje na t, oznacza to, że w tym miejscu pewne słowo zawiera swój znacznik końca. Które dokładnie? Spójrzmy jeszcze raz na rysunek 5.26. Komórka nazwana pierwotną umożliwia dostęp do wszystkich słów naszego 4-literowego alfabetu. Wskaźnik znajdujący się w t[1] (czyli t['B']) zawiera adres komórki oznaczonej jako (*). Znajdujący się w niej wskaźnik t[0] (czyli t['A']) wskazuje na (**). Tu uwaga! W komórce (**) t[4] jest zapętlony, czyli znajduje się tu znacznik końca słowa, na którego litery składały się odwiedzane ostatnio indeksy: najpierw 'B', potem 'A', na koniec znacznik końca słowa — co daje razem słowo BA.

Proces przechadzania się po drzewie nie jest bynajmniej zakończony: od komórki (**) odchodzi strzałka do (***), w której także następuje zapętlenie. Jakie słowo teraz przeczytaliśmy? Oczywiście BAK! Rozumując podobnie, możemy przeczytać jeszcze słowa BANK i ABBA.

Idea USS, dość trudna do wyrażenia bez poparcia rysunkiem, jest zaskakująco prosta w realizacji końcowej, w postaci programu wynikowego. Oczywiście nie stworzymy tutaj kompletnego modułu obsługi słownika, ale ta reszta, której brakuje (obsługa zapisu danych na dysk, „ładne" procedury wyświetlania, etc.), to już tylko zwykła praca wykończeniowa.

Omówmy po kolei procedury tworzące zasadniczy szkielet modułu obsługi *USS*.

Funkcje do_indeksu i z_indeksu pełnią role translacyjne. Z indeksów liczbowych tablicy t (elementu składowego rekordu *USS*) możemy odtworzyć odpowiadające poszczególnym pozycjom litery i vice versa. To właśnie zwiększając wartość stałej n oraz nieco modyfikując te dwie funkcje, możemy do modułu obsługującego *USS* dołączyć znajomość polskich znaków!

```
int do_indeksu(char c) //znak ASCII -> indeks
{
if ( (c<='Z') && (c>='A') || (c<='z') && (c>='a') )
   return toupper(c)-'A';   // toupper = zamiana małej litery na dużą
else
   {
   if (c==' ')  return 26;
   if (c=='-')  return 27;
   }
}

char z_indeksu(int n)                    // indeks -> znak ASCII
{
```

```
if ( n>=0  &&  n<= ('Z'-'A') )
   return toupper((char) n+'A');
else
   {
   if (n==26)  return ' ';
   if (n==27)  return '-';
   }
}
```

Funkcja `zapisz` otrzymuje wskaźnik do pierwszej komórki słownika. Zanim zostanie stworzona nowa komórka pamięci, funkcja ta sprawdzi, czy aby jest to na pewno niezbędne. Przykładowo niech w drzewie *USS* istnieje już słowo *ALFABET*, a my chcemy doń dopisać imię sympatycznego kosmity ze znanego amerykańskiego serialu: ALF. Otóż wszystkie poziomy odpowiadające literom 'A', 'L' i 'F' już istnieją — w konsekwencji żadne nowe komórki pamięci nie zostaną stworzone. Jedynie na poziomie litery 'F' zostanie utworzona komórka, w której do `t[n-1]` zostanie wpisany wskaźnik „do siebie". Przypomnijmy, że ten ostatni służy jako znacznik końca słowa.

```
void zapisz(char *slowo,USS_PTR p)
{
USS_PTR q; //zmienna pomocnicza
int pos;
for (int i=1; i<=strlen(slowo); i++)
   {
   pos=do_indeksu(slowo[i-1]);
   if (p->t[pos] != NULL) p=p->t[pos];
   else
      {
      q=new USS;
      p->t[pos]=q;
      for (int k=0; k<n; q->t[k++]=NULL);
      p=q;
      }
   }
p->t[n-1]=p; //pętla jako koniec słowa
}
```

Funkcja `pisz_slownik` służy do wypisania zawartości słownika — być może nie w najczytelniejszej formie, ale można się dość łatwo zorientować, jakie słowa zostały zapamiętane w *USS*.

```
void pisz_slownik(USS_PTR p)
{
for (int i=0; i<26; i++)
   if (p->t[i] != NULL)
      {
         if ( (p->t[i])->t[n-1]==p->t[i])      //gdy koniec słowa to
            cout << z_indeksu(i) << endl <<" ";//pisz znak końca linii
         else
            cout << z_indeksu(i);
         cout << "---"; // aby ładniej wyglądało
         pisz_slownik(p->t[i]); // wypisz rekurencyjnie resztę
      }
}
```

Funkcja `szukaj` realizuje dość oczywisty algorytm szukania pewnego słowa w drzewie: jeśli przejdziemy wszelkie gałęzie (poziomy) odpowiadające literom poszukiwanego słowa i trafimy na znacznik końca tekstu, to wynik jest chyba oczywisty!

```
void szukaj(char *slowo,USS_PTR p)
{// szukaj słowa w słowniku
int test=1, i=0;
while ((test==1) && (i<strlen(slowo)) )
   {
   if (p->t[do_indeksu(slowo[i])]==NULL)
      test=0; // brak odgałęzienia, słowa nie ma!
```

```
    else
       p=p->t[do_indeksu(slowo[i++])]; // szukamy dalej
    }
if ( (i==strlen(slowo)) && (p->t[n-1]==p) && test)
    cout << "Słowo znalezione!\n";
else
    cout << "Słowo nie zostało znalezione w słowniku\n";;
}
```

Oto przykładowa funkcja `main`:

```
int main()
{
int i;
char tresc[100];
USS_PTR p=new USS; // tworzymy nowy słownik
for(i=0; i<n; p->t[i++]=NULL);
for(i=1; i<=7; i++) // wczytamy 7 słów
    {
    cout << "Podaj słowo, które mam umieścić w słowniku:";
    cin >> tresc;
    zapisz(tresc,p);
    }
pisz_slownik(p);     // wypisujemy słownik
for(i=1 ;i<=4;i++)   // szukamy 4 słów
    {
    cout << "Podaj słowo, które mam poszukać w słowniku:";
    cin >> tresc;
    szukaj(tresc,p);
    }
}
```

Przypuśćmy, że podczas sesji z programem wpisaliśmy następujące słowa: `alf`, `alfabet`, `alfabetycznie`, `anagram`, `anonim`, `ASTRonoMIa`, `Ankara` (wielkie i małe litery zostały celowo pomieszane ze sobą). Po wczytaniu tej serii program powinien wypisać zawartość słownika w dość dziwnej, co prawda, ale w miarę czytelnej formie, która ukazuje rzeczywistą konstrukcję drzewa *USS* dla tego przykładu:

```
A-L-F
 -A-B-E-T
 -Y-C-Z-N-I-E
 -N-A-G-R-A-M
 -K-A-R-A
 -O-N-I-M
 -S-T-R-O-N-O-M-I-A
```

Zbiory

Implementacja programowa zbiorów matematycznych napotyka na szereg ograniczeń związanych z używanym językiem programowania.

Miłośnicy Pascala znają zapewne definicje zbliżone do:

```
type Litery = 'A' ..'Z';
     ZbiorLiter = set of Litery;
var Alfabet:ZbiorLiter;
    c: char;
begin
Alfabet:=['A'..'Z'];
read(c);
if c in Alfabet then {itd.}
end.
```

Oczywiście to, co dla programisty pascalowego jest zbiorem, wcale nim nie jest dla matematyka z uwagi na wymóg *jednakowego typu* zapamiętywanych elementów.

Niemniej jednak dla podstawowych zastosowań konwencje istniejące w Pascalu nadają się znakomicie, gdyż możliwe jest np. wykonywanie operacji typu: dodawanie elementu do zbioru, mnożenie (iloczyn) zbiorów, odejmowanie zbiorów, sprawdzanie, czy obiekt należy do zbioru itd.

W tej książce do opisu algorytmów i prezentacji struktur danych używamy języka C++, który na ogół spełnia swoje zadanie dość dobrze. Niestety nie posiada on wbudowanej obsługi zbiorów i w związku z tym należy ją dołożyć w sposób jawny, używając przy okazji różnorodnych technik, zależnych od aktualnie realizowanych zadań.

Weźmy dla przykładu *implementację zbioru znaków*, która nie wymaga użycia struktur listowych i dynamicznego przydzielania pamięci. Załóżmy, że w komputerze występuje „tylko" 256 znaków (między innymi znaki alfabetu duże i małe, cyfry oraz tzw. znaki kontrolne niedrukowalne).

Do zasymulowania zbioru wystarczy wówczas najzwyklejsza tablica typu `unsigned char`, tak jak w przykładzie poniżej:


Listing

set.cpp

```
class Zbior
{
private:
   unsigned char zbior[256]; // całą tablica ASCII
public:
Zbior()
   {//zerowanie zbioru w konstruktorze
   for(int i=0;i<256;i++)
      zbior[i]=0;
   }
Zbior& operator +(unsigned char c)
   { // dodaj znak 'c' do zbioru i zwróć zmieniony obiekt
   zbior[c]=1;
   return *this;
   }
Zbior& operator -(unsigned char c)
   {// usuń znak 'c' ze zbioru i zwróć zmieniony obiekt
   zbior[c]=0;
   return *this;
   }
bool nalezy(unsigned char c) // czy 'c' należy do zbioru?
   {
   return zbior[c]==1;
   }

Zbior& dodaj(Zbior s2)        // dodaj zawartość zbioru 's2' do obiektu
   {
   for(int i=0; i<256;i++)
      if(s2.nalezy(i))        // jeśli element obecny w s2
         zbior[i]=1;          // dodaj go do zbioru
   return *this;          // zwraca zmodyfikowany obiekt
   }

int ile()            // zwraca liczbę elementów w zbiorze
   {
   int n;
   for(int i=0; i<256; i++)
     if(zbior[i]==1)          // element obecny
        n++;
   return n;
   }
```

```
void pisz()
{// wypisuje zawartość zbioru
   int i;
   cout << "{";
   for(i=0; i<256; i++)
      if(zbior[i]==1)        // wypisz obecny element
         cout << (char)i<< " ";
   if(i==0)
      cout << "Zbiór pusty!";
   cout << "}\n";;
}
}; // koniec definicji klasy Zbiór
```

Pomimo dużej prostoty powyższa implementacja umożliwia już manipulacje typowe dla zbiorów:

```
int main()
{
Zbior s1, s2;
s1=s1+'A'; s1=s1+'A'; s1=s1+'B'; s1=s1+'C';
s2=s2+'B'; s2=s2+'B'; s2=s2+'E'; s2=s2+'F';
cout << "Zbior S1 =";  s1.pisz();
s1=s1-'C';
cout << "Zbior S1 - 'C' =";  s1.pisz();
cout << "Zbior S2 =";  s2.pisz();
   s1.dodaj(s2);
cout << "Zbior S1 + S2 = ";
   s1.pisz();
}
```

Uruchomienie programu powinno spowodować wyświetlenie na ekranie następujących komunikatów:

```
Zbiór S1 = { A B C }
Zbiór S1 - 'C' = { A B }
Zbiór S2 = { B E F }
Zbiór S1 + S2 = { A B E F }
```

Czytelnik z łatwością uzupełni samodzielnie operacje dostępne w powyższej implementacji klasy `Zbior` o przecinanie (iloczyn) i odejmowanie zbiorów.

Możliwe jest oczywiście stworzenie bardziej ogólnej implementacji zbiorów, akceptującej zmienną liczbę danych (wymaga dynamicznego przydziału pamięci, np. za pomocą list) i zezwalającej na złożone elementy składowe, np. rekordy danych. Wydaje się jednak, że zaprojektowanie klasy `Zbior` z użyciem klas szablonowych i list byłoby nadużyciem siły, w przypadku gdy jedynymi potrzebnymi nam elementami zbiorów miałyby zostać jedynie... znaki alfabetu!

Zadania

Zadanie 1.

Zastanów się, jak można w prosty sposób zmodyfikować model *Uniwersalnej Struktury Słownikowej* (patrz strona 140), aby możliwe było jej użycie jako słownika 2-językowego, np. polsko-angielskiego. Oszacuj wzrost kosztu słownika (chodzi o ilość zużytej pamięci) dla następujących danych: 6 000 rekordów *USS* w pamięci zawierających 25 000 zapamiętanych słów.

Zadanie 2.

Zestaw dość podobnych zadań. Napisz funkcje, które usuwają:

- pierwszy element listy;
- ostatni element listy;

- pewien element listy, który odpowiada kryteriom poszukiwań podanym jako parametr funkcji (aby uczynić funkcję uniwersalną, wykorzystaj metodę przekazania wskaźnika funkcji jako parametru).

Zadanie 3.

Napisz funkcję, która:

- Zwraca liczbę elementów listy (a).
- Zwraca *k*-ty element listy (b).
- Usuwa *k*-ty element listy (c).


Ostrzeżenie

Podczas rozwiązywania zadań 2. i 3. proszę dokładnie przemyśleć efektywny sposób informowania o sytuacjach błędnych (np. próba usunięcia *k*-tego elementu, podczas gdy on nie istnieje, etc.).

Rozwiązania zadań

Zadanie 1.

Modyfikacja struktury *USS*:

```
typedef struct slownik
{
struct slownik  *t[n];
char            *tlumaczenie;
} USS,*USS_PTR;
```

Tłumaczenie jest „dopisywane" (alokowane) do funkcji `zapisz` podczas zaznaczania końca słowa — w ten sposób nie stracimy związku *słowo-tłumaczenie*.

Koszt:

- bez drugiego języka:
  - Koszt = (n = 29)·4 bajty („duży" model pamięci) = 696 000 bajtów = ok. 679 kB.
- z drugim językiem:
  - Założenie: średnia długość słowa angielskiego wynosi 9 bajtów + ogranicznik, czyli 10 bajtów.
- Koszt = przypadek poprzedni plus 25 000·10 plus pewna liczba niezużytych wskaźników na tłumaczenie — przyjmijmy zaokrąglenie na 1 000. Ostatecznie mamy: 25 000·10+1 000·4 = 254 000 bajtów, czyli ok. 248 kB.

W danym przypadku koszt wzrósł o ok. 36% pierwotnej zajętości pamięci.

Zadanie 3.

Oto propozycja rozwiązania zadania 3a:

```
int cpt(ELEMENT *q, int res=0)
{
if (glowa==NULL)
   return res;
else
   cpt(q->nastepny,res+1);
}
```

Przykładowe wywołanie: `int ilosc=cpt(inf -> glowa)`.