Termin Egzaminu (Język C):

>> ŚRODA, 4 LUTEGO 2026 <<

GODZ. 11.00

[*POPRAWKA:* PONIEDZIAŁEK, 23 LUTEGO, 11.00]

!!!

Forma zaliczenia kursu: Egzamin pisemny – test wyboru *) **)

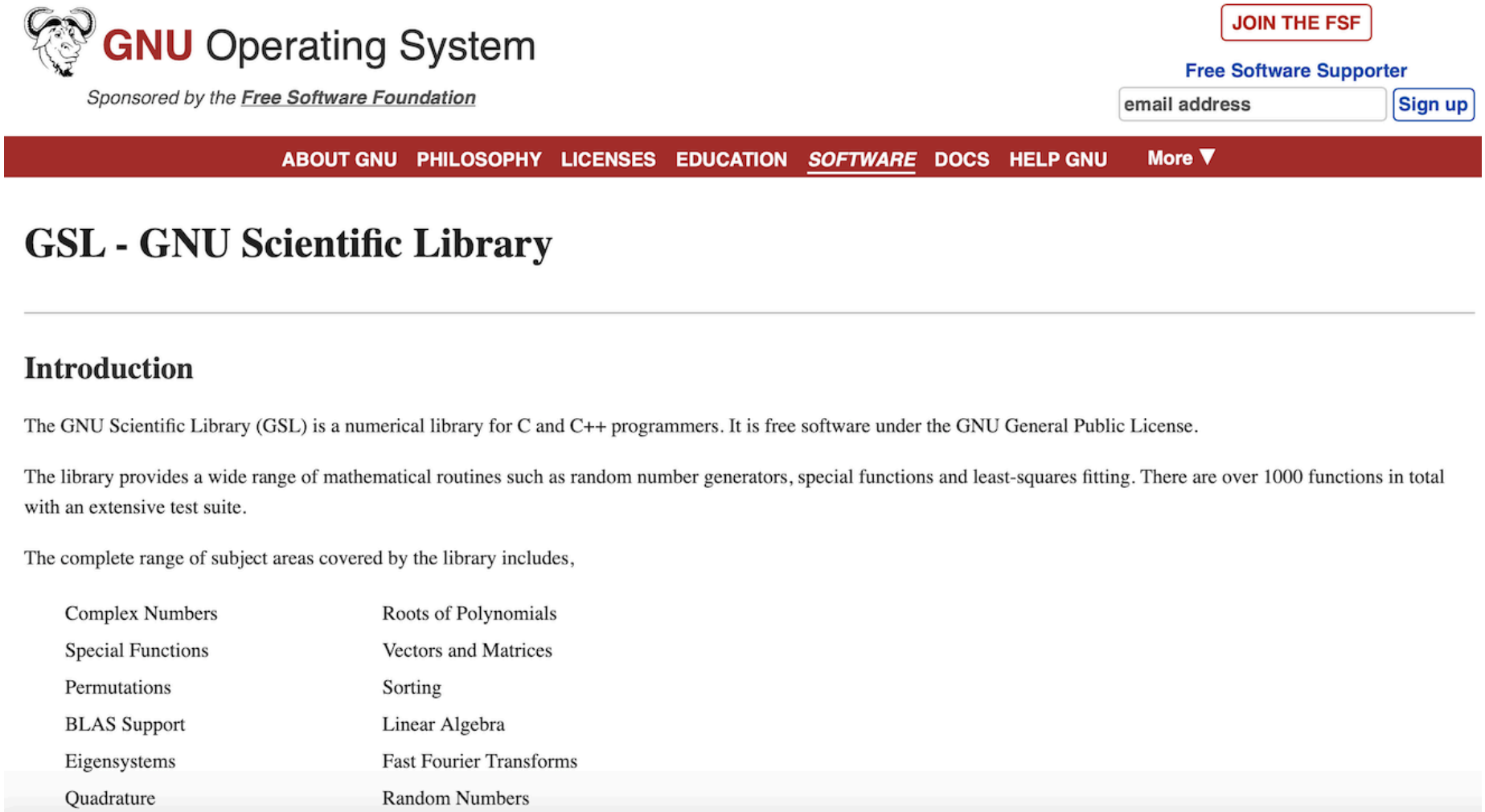
*) Warunkiem przystąpienia do egzaminu jest ***zaliczenie ćwiczeń***
(w uzasadnionych przypadkach: *zgoda prowadzącego ćwiczenia*)

) **Ocena 5.0 (bdb) z ćwiczeń *zwalnia z egzaminu*

[OCENA KOŃCOWA: *0.5*ocena z ćwiczeń + 0.5*wynik egzaminu* **]**

Kiedy potrzebujemy obliczeń numerycznych ...

— **GSL (*G*NU *S*cientific *L*ibrary):** www.gnu.org/software/gsl/



GNU Operating System

Sponsored by the Free Software Foundation

JOIN THE FSF

Free Software Supporter

email address Sign up

ABOUT GNU PHILOSOPHY LICENSES EDUCATION SOFTWARE DOCS HELP GNU More ▼

GSL - GNU Scientific Library

Introduction

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers. It is free software under the GNU General Public License.

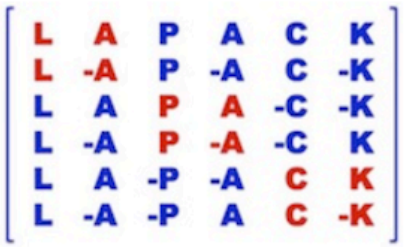
The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite.

The complete range of subject areas covered by the library includes,

Complex Numbers	Roots of Polynomials
Special Functions	Vectors and Matrices
Permutations	Sorting
BLAS Support	Linear Algebra
Eigensystems	Fast Fourier Transforms
Quadrature	Random Numbers

Wydajna numeryczna algebra liniowa …

— LAPACK (*Linear Algebra PACKage*): www.netlib.org/lapack/



Version 3.9.0
LAPACK on GitHub
Browse the LAPACK User Forum
Browse the LAPACK User Forum
Contact the LAPACK team

Get the latest LAPACK News

access

LAPACK is a software package provided by Univ. of Tennessee; Univ. of California, Berkeley; Univ. of Colorado Denver; and NAG Ltd..

Presentation

LAPACK is written in Fortran 90 and provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.

Porównanie wydajności GSL i LAPACK-a:

Thursday, June 25, 2009

gsl vs lapack performance

I had some doubts about the LU routines in the gsl library (GNU Scientific Library). See http://yetanothermathprogrammingconsultant.blogspot.com/2009/06/gsl-gnu-scientific-library.html. Here I try a quick experiment by inverting a square nxn matrix. As test matrix I used the Pei matrix (http://portal.acm.org/citation.cfm?id=368975). Here are the results:

library	**gsl**	**lapack**
routine	gsl_linalg_LU_decomp + gsl_linalg_LU_invert	dgesv
compiler	cygwin gcc	Lahey lf95
n=100	0.047 seconds	0.024 seconds
n=1000	6.735 seconds	2.017 seconds
n=2000	1:01 minutes	17.257 seconds

[*http://yetanothermathprogrammingconsultant.blogspot.com/2009/06/gsl-vs-lapack-performance.html*]

Dlaczego pisanie dobrych programów numerycznych nie jest łatwe?

[*DALEJ — omówimy przykład/pułapkę Jamesa H. Wilkinsona: https://en.wikipedia.org/wiki/Wilkinson's_polynomial*]

Wprowadzenie: *Jak szukać miejsc zerowych funkcji?*

— Metoda bisekcji (*równego podziału*) [=> *wolna,* ***b. stabilna***]

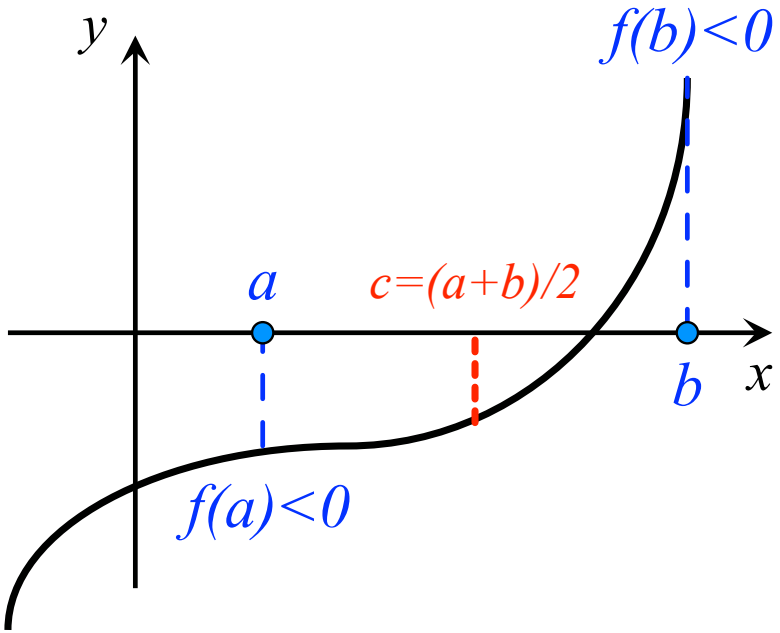
— Metoda stycznych (*Newtona*) [=> *szybka, ale sprawia kłopoty…*]

— Metoda siecznych (ang. *secant metod*) [=> ***kompromis***]

— …

[*Patrz np.*: https://mathworld.wolfram.com/HalleysMethod.html]

Metoda bisekcji



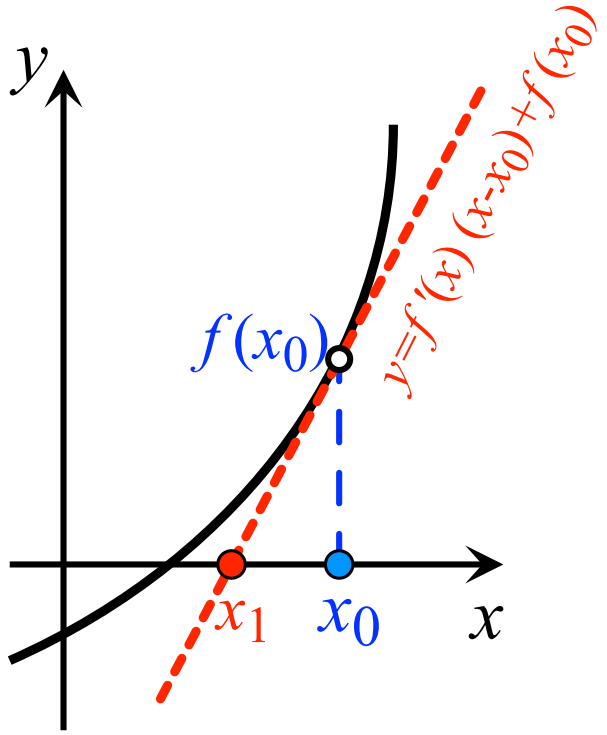
1. Wybieram *a* i *b* t., że *f(a)f(b)<0*
2. Obliczam: *c=(a+b)/2* oraz *f(c)*
3. Jeśli *f(a)f(c)<0*, podstawiam: *b=c*
4. W przeciwnym przypadku: *a=c*
5. Wracam do kroku 2.

W każdym kroku, przedział (a,b) ulega zmniejszeniu o czynnik 2; po n krokach znamy zatem rozwiązanie z dokładnością do $|a-b|\cdot 2^{-n}$.

[=> *~3 nowe cyfry dziesiętne po każdych 10 podziałach!*]

Metoda stycznych (m. Newtona)

Jeśli potrafimy obliczać *pochodną* $f'(x)$ funkcji $f(x)$, możemy postąpić inaczej niż w omówionej wcześniej metodzie bisekcji.



1. Ustalam n=0, wybieram punkt x_0
2. Obliczam: $f(x_n)$ oraz $f'(x_n)$
3. Obliczam: $x_{n+1} = x_n - f(x_n)/f'(x_n)$
4. Zwiększam n o 1; wracam do kroku 2.

Jeśli metoda Newtona jest zbieżna (a zależy to od własności funkcji f…), wynik w arytmetyce podwójnej precyzji

najczęściej zostaje ustalony [$x_n == x_n + (x_n - x_{n-1})$] *już po kilkunastu krokach. Łatwo jednak wskazać przypadki, kiedy metoda* ***w ogóle nie działa****, zob. https://en.wikipedia.org/wiki/Newton's_method [i przypisy!]*

Wielomian Wilkinsona

Niemal na początku ery komputerów (*w 1963 roku*), J. H. Wilkinson rozważał problem szukania miejsc zerowych wielomianu:

$$p(x) = (x-1)(x-2)(x-3)\cdot\ldots\cdot(x-20) =$$

$$x^{20} - 210\,x^{19} + 20615\,x^{18} - \ldots + 20!$$

[*Znając rozwiązania dokładne, możemy testować algorytmy numeryczne!*]

Wilkinson szukał zer *metodą Newtona*, na komputerze *Pilot ACE (zaprojektowanym przez A.Turinga)*, z 30 bitową reprezentacją liczb zmiennopozycyjnych. *Mantysa* liczyła 22 bity, a zatem $\varepsilon = 2^{-23}$ było już liczbą, dla której: $1.0 + \varepsilon == 1.0$.

Okazuje się (*czego Wilkinson początkowo nie był świadomy…*), że dla wielomianu p(x) zmiana współczynnika przy x^{19}, z wartości -210 na $-210 - 2^{-23}$ sprawia, że część rozwiązań równania $p(x) = 0$ w ogóle znika (*zamienia się w rozwiązania zespolone*), zaś np. $x_{20}=20$ przesuwa się do $x_{20} \approx 20.8$ (**!**)

Ku zaskoczeniu Wilkinsona, algorytm Newtona (*ani żaden inny!*) poszukiwania zer nie mógł zatem na takiej maszynie poprawnie działać …

Więcej: https://en.wikipedia.org/wiki/Wilkinson's_polynomial

Czy w C można “liczyć lepiej”?

Przykładem, któremu teraz się przyjrzymy będzie metoda *fast inverse square root,* służąca do obliczania funkcji $1/\sqrt{a}$ (*potrzebnej do normalizacji wektorów,* obliczania kątów w grafice 3D itp.).

```
# include <stdint.h>  /* zawiera: uint32_t ==> POSIX(!) */

float q_rsqrt(float number)
{
  union {
    float    f;
    uint32_t i;
  } conv = { .f = number };
  conv.i  = 0x5f3759df - (conv.i >> 1);
  conv.f *= 1.5F - (number * 0.5F * conv.f * conv.f);
  return conv.f;
}
```

Metoda pojawiła się w grze *Quake III Arena*, wydanej w 1999 roku [*patrz*: https://en.wikipedia.org/wiki/Fast_inverse_square_root].

W dużym skrócie, pomysł zasadza się na spostrzeżeniu, że — z uwagi na reprezentację l.zmiennopozycyjnych w maszynie — *odczytanie(*)* zmiennej typu `float` jako `usigned int` w istocie jest równoważne obliczeniu $\log_2$ dla tej zmiennej, przesuniętego o PEWNĄ_STAŁĄ (*dużą!*) i obciętego do części całkowitej.

Z kolei operacja odwrotna — odpowiadać będzie obliczaniu

$$2^{-\text{ PEWNA\_STAŁA }} \times 2^{(-\textit{ wynik powyższego }-)}.$$

Obliczając przybliżony $1/\sqrt{a}$ korzystamy z tożsamości

$$\log_2 \frac{1}{\sqrt{a}} = -\frac{1}{2}\log_2 a,$$

przy czym — zastępując logarytmowanie wspomnianą konwersją — musimy dodatkowo uwzględnić czynnik $2^{\text{PEWNA\_STAŁA}/2}$, co daje $\sqrt{2^{127}}$, a po konwersji do l.całkowitej (`unsigned`) i w zapisie szesnastkowym: `0x5F3759DF`. Linia kodu:

```
conv.i  = 0x5f3759df - (conv.i >> 1);
```

odpowiada zatem przybliżonej wersji podanej wyżej tożsamości dla logarytmów (o podstawie 2).

—

(*) Trzeba zaznaczyć, że zwykłe rzutowanie typów (ang. *cast*) oczywiście nie nadaje się do przybliżonego obliczania $\log_2 a$; w oryginalnej wersji kodu *Quake*'a użyto operacji rzutowania wskaźników: `*(long*) &a`, co niestety nie daje jednoznacznego wyniku dla każdej implementacji C. Problem — w ramach ANSI C i POSIX — rozwiązuje użycie *unii*.

Dalej, ***poprawiamy przybliżenie*** wykonując pojedynczą iterację *metody Newtona* dla funkcji: $f(x) = \frac{1}{x^2} - a,$ której miejsca zerowego szukamy $\left[f(x) = 0 \Leftrightarrow x = 1/\sqrt{a} \right]$. Każda iteracja, czyli

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n \left(\frac{3}{2} - \frac{1}{2} a x_n^2 \right),$$

odpowiada przypisaniu:

```
x *= 1.5F - (number * 0.5F * x * x);
```

Dokładność opisanej metody jest na poziomie *~3 cyfr znaczących*, z pewnością za mało dla obliczeń naukowych, jednak do zastosowanie w grafice była wystarczająca.

Poprzedni wykład [*13. 01. 2026*] :

— ***Struktury rekurencyjne*** (drzewa binarne, jednokierunkowe łańcuchy odsyłaczy)

— Tablice mieszające (*„hashmap-y”*)

— Synonimy typów: **`typedef`**

— Unie (*union*); pola bitowe

Kodowanie Huffmana

Metoda kodowania Huffmana to przykład ***bezstratnej kompresji danych***, w której wzorce (np. *znaki* lub *słowa*) często powtarzające się zastępujemy krótkimi ciągami bitów, zaś wzorce rzadko występujące — długimi.

Metoda została opracowana w 1952 roku przez **Davida Huffmana**, który był wówczas studentem w MIT (zob.: http://tinyurl.com/u4pc97ew). Inne, zasługujące na uwagę algorytmy kompresji bezstratnej to **LZ77** (**Lempel-Ziv-1977**), w którym kompresja jest dokonywana na poziomie merytorycznym informacji (poszukiwanie "zdań", tj. długich powtarzających się wzorców), czy też tzw. *"asymetryczne systemy liczbowe"* (*zob.* J.Duda, arXiv:1311.2540).

W teorii, jeśli do zapisu informacji używamy N różnych symboli (najczęściej będą to *znaki,* jednak nic nie stoi na przeszkodzie, aby były nimi np. pary znaków, słowa, itp.), które mogą pojawiać się z prawdopodobieństwami p_1, p_2, …, p_N (*znanymi!*), wówczas tzw. **entropia źródła**, która jest *miarą ilości informacji*, wyniesie:

$$H(S) = -\left(p_1 \log_2 p_1 + p_2 \log_2 p_2 + \ldots + p_N \log_2 p_N\right).$$

W praktyce, **współczynnik kompresji** będzie nieco niższy niż wynika z entropii źródła, $r = H(s)/\log_2 N_{\max}$, gdzie $N_{\max}$ oznacza maksymalną l.symboli w zapisie wejściowym (zwykle $N_{\max} = 256$ a zatem $\log_2 N_{\max} = 8 \equiv N_{\text{bits}}$). Jest tak, ponieważ potrzebujemy całkowitej liczby bitów do zakodowania każdego znaku [*szczegóły dalej…*], jak również *gdzieś* musimy zakodować słownik umożliwiający dekodowanie (tzw. *drzewo Huffmana*).

Statyczne kodowanie Huffmana

Algorytm Huffmana w wersji statycznej (*drzewo* budujemy tylko raz, nie uaktualniamy prawdopodobieństw) wygląda tak:

1. Określamy prawdopodobieństwo (lub ***liczbę wystąpień***) każdego symbolu w zbiorze danych [np. *pliku przed kompresją*]
2. Tworzymy listę drzew binarnych (na samym początku: *jednoelementowych!*), zawierających pary: (*symbol, liczba wystąpień*). Porządkujemy elementy wg liczby wystąpień.
3. ***Dopóki*** lista zawiera *więcej niż jedno* drzewo:
   a. Usuwamy 2 drzewa z najmniejszą liczbą wystąpień
   b. Dodajemy nowe drzewo, którego korzeń zawiera *sumę* l.wystąpień z usuniętych drzew; drzewa usunięte stają się lewym i prawym poddrzewem (=> *symbole przechowujemy wyłącznie w liściach!*)

Po zakończeniu algorytmu, początkowa lista zredukuje się do ***pojedynczego drzewa***.

Symbole w nim zapisane identyfikowane są za pomocą ciągów bitów określających kolejne ruchy lewo/prawo (=> `0` / `1`), które należy wykonać, aby dojść do odpowiedniego *liścia* drzewa.

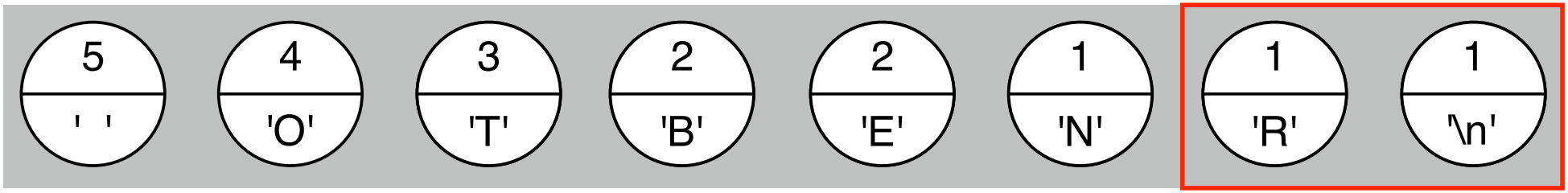
Kodowanie Huffmana jest przykładem ***kodowania prefiksowego***; tzn. takiego, w którym żaden krótszy ciąg bitów nie zawiera się w innym (dłuższym). Jeżeli — podczas dekodowania informacji — napotkamy liść drzewa oznacza do, że musimy wyprowadzić znak i powrócić do korzenia.

[⇒ *Podobna zasada obowiązuje np. przy kodowaniu znaków w standardzie Unicode*]

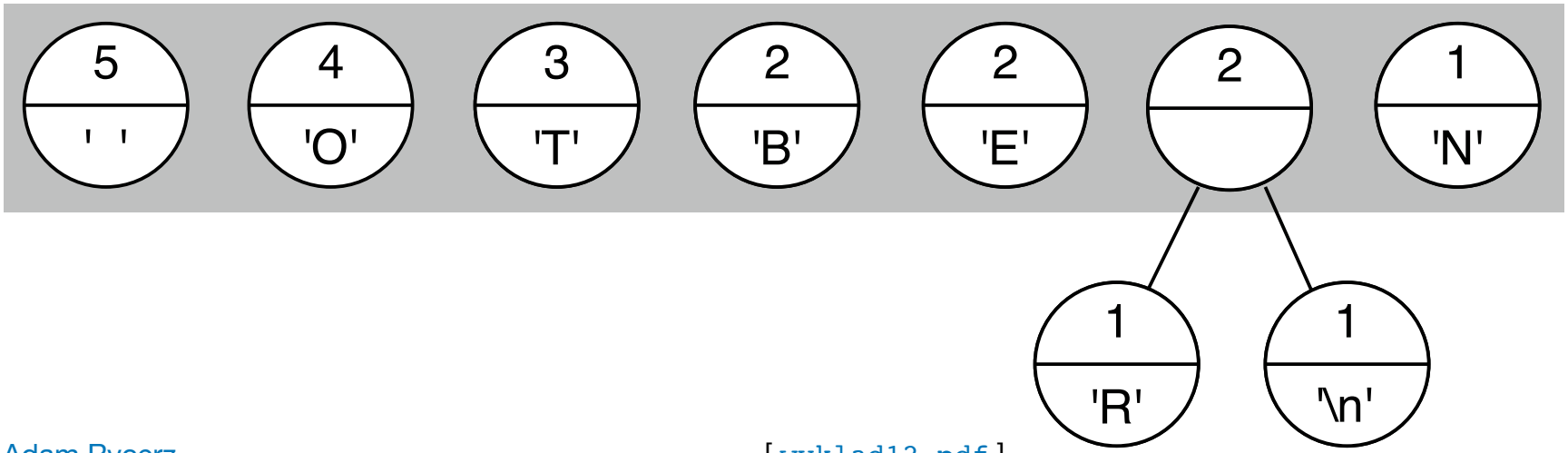
Przykład — *budujemy drzewo Huffmana dla tekstu:*

`“TO BE OR NOT TO BE\n”`

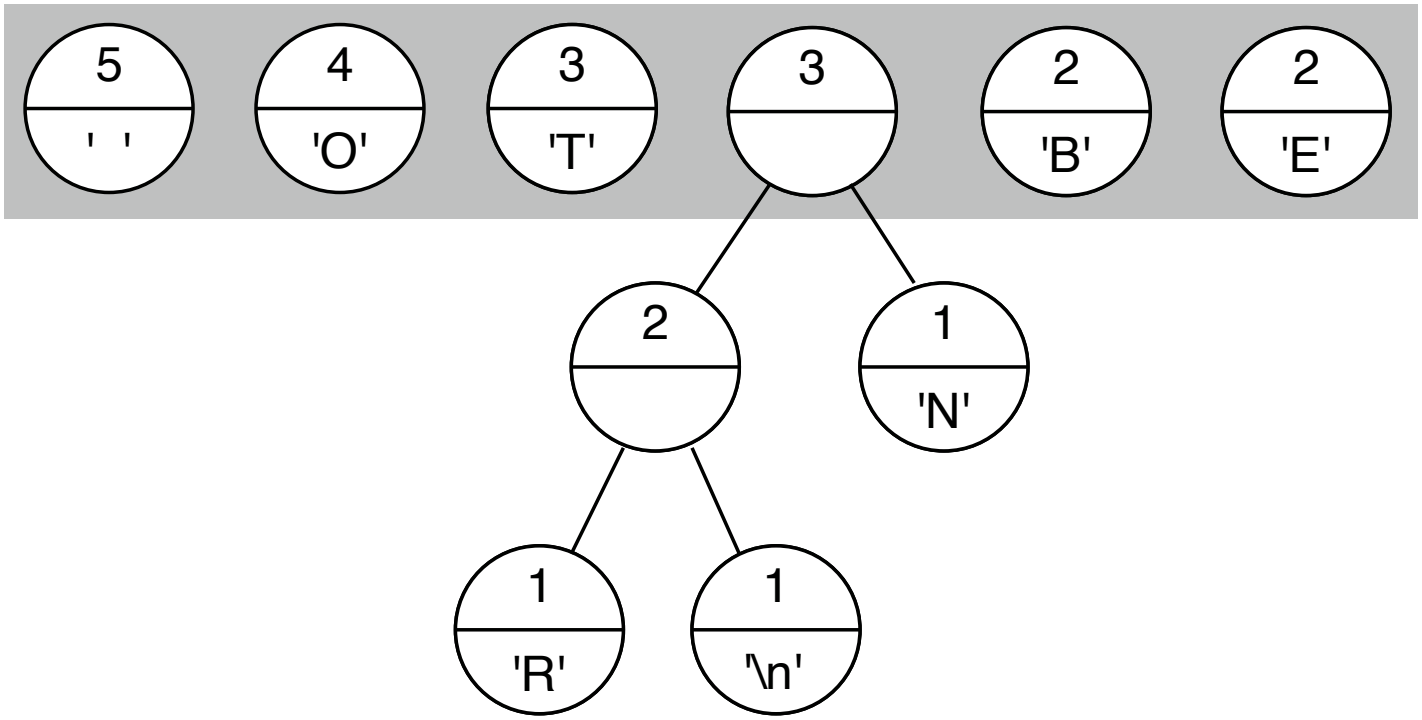
1.–2. Liczymy wystąpienia znaków i tworzymy listę drzew:



3a–b. Łączymy drzewa o najmniejszej l.powtórzeń:

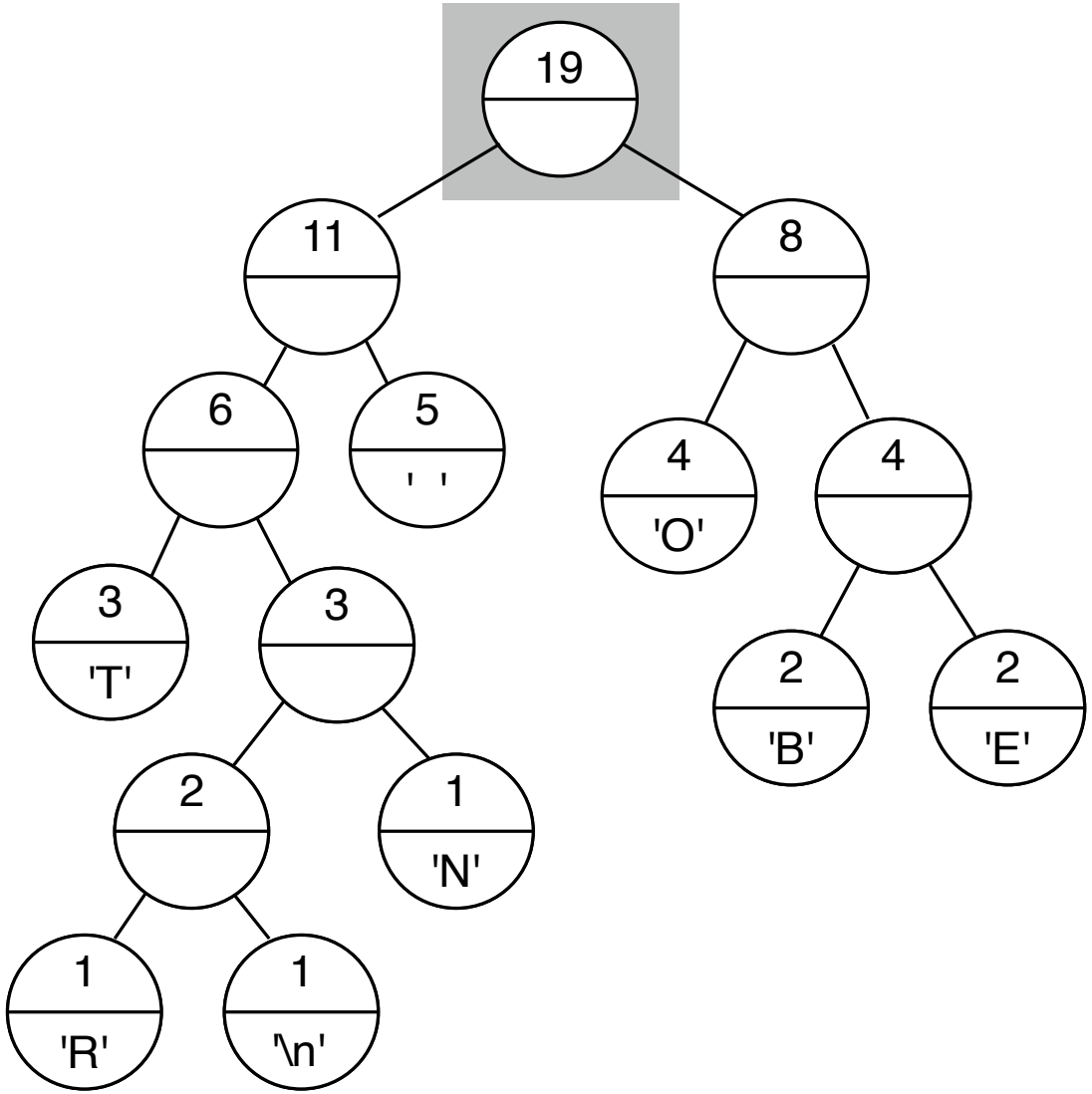


W kolejnym kroku otrzymujemy:

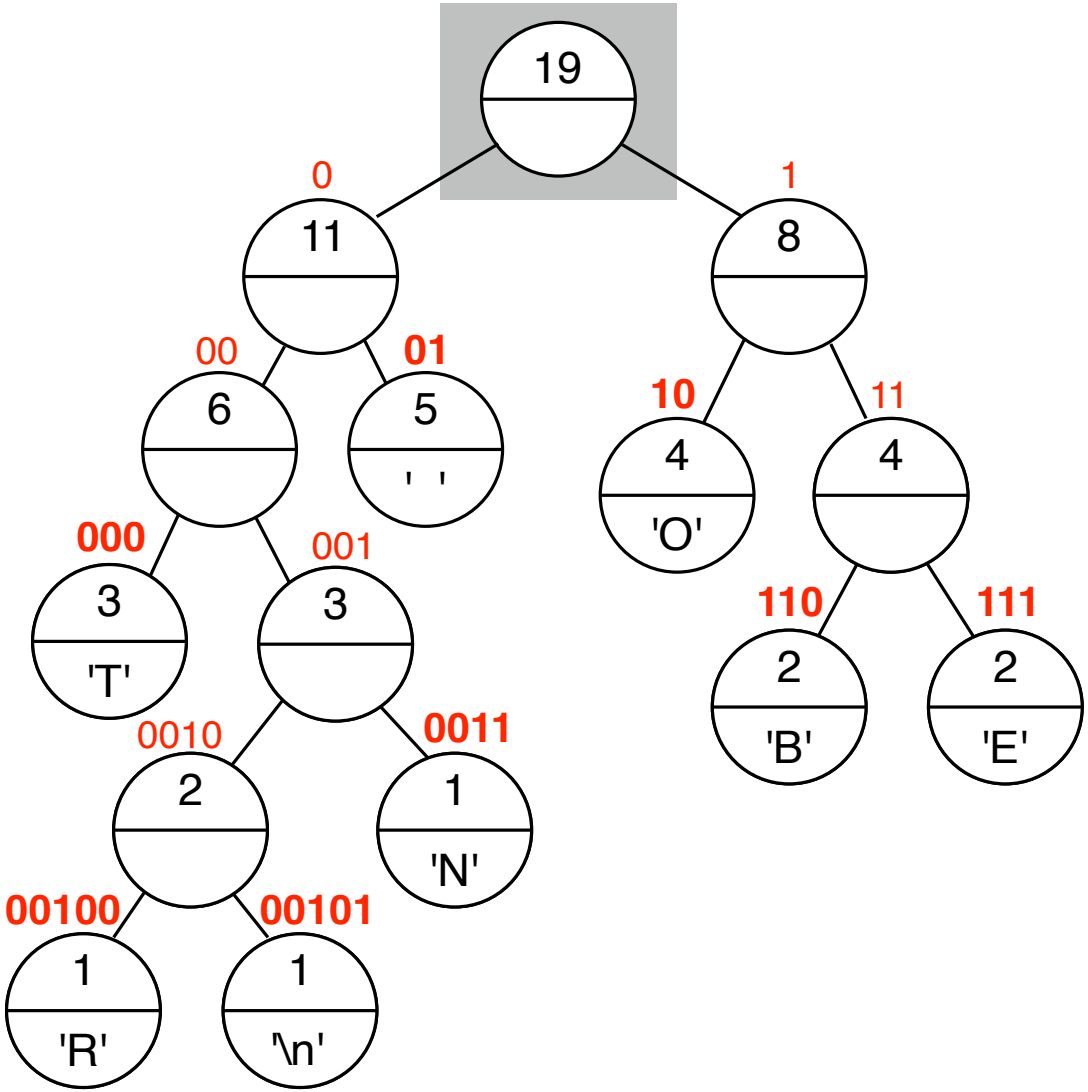


… *(tutaj wykonujemy 5 analogicznych operacji)* …

Ostatecznie, po *siedmiu* iteracjach kroków 3a–b (*na każdym etapie porządkujemy listę!*) mamy gotowe drzewo:



Dalej, generujemy ***kody Huffmana*** opisujące położenia znaków w drzewie:



Nasz ***słowniczek*** (tzn. lista kodów Huffmana) dla tekstu `"TO BE OR NOT TO BE\n"` wygląda zatem tak [*l.powtórzeń w nawiasach*]:

```
01      --  ' '   [5]
10      --  'O'   [4]
000     --  'T'   [3]
110     --  'B'   [2]
111     --  'E'   [2]
0011    --  'N'   [1]
00100   --  'R'   [1]
00101   --  '\n'  [1]
```

Widzimy, że znaki występujące ***częściej*** w tekście źródłowym zostały zapisane bliżej korzenia drzewa i będą zastąpione ***krótszymi*** kodami.

Łącznie, w napisie `"TO BE OR NOT TO BE\n"` o długości *19 bajtów* występuje ***8 różnych znaków***; które — zastąpione kodami o długości od dwóch (`01`, `10`) do pięciu bitów (`00100`, `00101`), według *słowniczka* powyżej — utworzą zakodowaną (*skompresowaną*) informację o długości [*l.wystąpień * dł.kodu + ...*]:

5*2 + 4*2 + 3*3 + ... 1*5 = 53 bity = 6,625 bajta,

co możemy *fizycznie* zapisać za pomocą *7 bajtów* [*zamiast początkowych 19!*]. Informacja zapisana w nadliczbowych bitach ostatniego bajtu nie ma znaczenia — nie będzie odczytywana przy dekompresji pliku.

Widzimy, że nasz stopień kompresji wynosi $7/19 \approx 0.368$.

Dla porównania, obliczając prawdopodobieństwa wystąpień poszczególnych znaków :

$$p(\text{`` ''}) = 5/19,\ \ p(\text{`O'}) = 4/19,\ \ldots\ ,\ p(\text{`\textbackslash n'}) = 1/19,$$

a następnie entropię źródła *H(S)* dostajemy, że teoretyczny stopień kompresji może wynosić:

$$\frac{H(S)}{N_{\text{bits}}} = \frac{2.755}{8} \approx 0.344$$

A zatem, wartość 0.368 jest całkiem niezłym wynikiem!

[*Gdyby, jakimś sposobem, udało nam się zapisać 0.625 bajta na końcu, kompresja wynosiłaby 6.625/19≈0.349, byłaby zatem b.bliska ideału.*]

[***Niestety, w pliku skompresowanym będziemy musieli jeszcze zakodować drzewo…***]

UWAGI TECHNICZNE:

Naturalną strukturą danych, przydatną na etapie *zliczania znaków* w tekście wejściowym, jak również kodowania tekstu (po stworzeniu "słowniczka" kodów) jest ***tablica mieszająca***, zawierająca wskaźniki do liści drzewa, której indeksem będzie po prostu *znak* (zrzutowany do typu `unsigned char`).

Plik wynikowy powinien rozpoczynać się od nagłówka, zawierającego **trzy liczby typu `unsigned long`**, będące (kolejno) rozmiarami w bajtach: *pliku wejściowego*, *drzewa*, oraz *wiadomości zakodowanej* [zob: http://tinyurl.com/4mm8r436]

Dalej, kodujemy drzewo w postaci ciągu:

[*kod Huffmana*] SEPARATOR [*znak kodowany*] ...

gdzie SEPARATOR jest dowolnym znakiem różnym od `0` i `1`.
Przykładowo, dla naszego drzewa ciąg może wyglądać tak:

```
00101|
01| 110|B111|E0011|N10|O000100|R000|T
```

Zapis drzewa (wraz z separatorami '|') liczy zatem 43 znaki, zaś cały plik po kompresji będzie miał *62 bajty* [= 43 + 7 + 3*4, gdzie ostatni składnik jest rozmiarem nagłówka zawierającego 3 liczby typu `unsigned long`] . Plik "skompresowany" okazuje się zatem dłuższy od oryginalnego, co dla b.krótkich zbiorów danych nie jest niczym osobliwym; nasze *drzewo moglibyśmy jeszcze nieco "dopakować" stosując bardziej wyszukany zapis kodów Huffmana…*
[**Dla porównania**: `gzip` tworzy plik o rozmiarze *45 bajtów*.]

Proces dekodowania rozpoczyna się od wczytania drzewa z pliku i budowy odpowiedniej struktury dynamicznej w pamięci komputera. (*Dzięki nagłówkowi wiemy, gdzie kończy się zapis drzewa a zaczyna wiadomość do odkodowania!*)

Następnie, kolejne ciągi bitów (*kody Huffmana*) zamieniamy na znaki zwyczajnie wędrując po drzewie; przy czym po każdorazowym wyprowadzeniu znaku — następuje powrót do korzenia. (Ponownie, nagłówek poinformował nas, ile znaków mamy do wyprowadzenia, nie czytamy zatem ew. *nadliczbowych* bitów w ostatnim bajcie pliku skompresowanego.)

Dla ilustracji, podamy teraz ***przykładowe implementacje*** funkcji realizujących zapis i odczyt kodów Hufmanna (*“bit po bicie”*).

```
int put1bit(FILE *f, int bit)   /* Wypisz bit do f */
{
  static unsigned char b = 00u;
  static int n=0, out;

  if (bit == '1')
    b += 01u << n;
  n++;

  if (n == 8) {
    out = putc(b, f);
    b =  00u;
    n = 0;
    return out;
  }

  return b;
}
```

Pierwsze 8 bitów trafia do ***bufora*** (zmienna statyczna `b` typu `unsigned char`). Kiedy bufor jest już zapełniony (`n == 8`), wyprowadzamy bajt zapisany w buforze `b` do strumienia `f`, przy czym ***wartość zwróconą*** przez funkcję `putc(b, f)` przechowujemy w zmiennej pomocniczej `out` [a jest nią wyprowadzony znak lub kod błędu, np `EOF`].

Po wyprowadzeniu znaku, zerujemy bufor `b` (i licznik bitów `n`), po czym zwracamy wartość `out`.

Jeśli `n < 8`, a zatem bufor nie jest jeszcze pełny, **funkcja każdorazowo zwróci aktualną zawartość bufora**. Umożliwi to — w razie potrzeby — zapisanie końcówki ostatniego kodu do pliku wyjściowego (co realizujemy *na zewnątrz funkcji*), gdyż dane do wypisania mogą "skończyć się" zanim bufor zostanie zapełniony (*liczba bitów do wyprowadzenia na ogół nie dzieli się przez 8*).

```
int get1bit(FILE *f)   /* Pobierz bit z f */
{
  static unsigned char b;
  static int n=0, out;

  if (n == 0) {
    if (EOF == (out = getc(f)))
      return EOF;
    b = out;
  }
  if (b & 01u)
    out = '1';
  else
    out = '0';
  b >>= 1;
  if (++n == 8)
    n = 0;
  return out;
}
```

Jeśli **bufor jest pusty** (`n == 0`), oraz próba wczytania znaku ze strumienia `f` z pomocą funkcji `getc(f)` nie zwróciła `EOF`-a, zapisujemy wczytany znak w buforze `b`.

Dalej, **zapisujemy ostatni bit bufora** (`b & 01u`), zdekodowany jako znak (`'1'` lub `'0'`) do zmiennej pomocniczej `out` po czym przesuwamy bufor zamazując ten bit (b >>= 1). Jeśli był to ostatni (czyli ósmy) bit, wówczas zerujemy licznik bitów (`n`).

Po wszystkim, zwracamy zdekodowany bit odczytany wcześniej z bufora, przechowany w `out`.

[*Jeśli chcemy, aby program działał poprawnie dla* ***plików binarnych****, w których mogą wystąpić znaki o kodach ujemnych, należy zwracać szczególną uwagę na rzutowanie znaków do* **`unsigned char`** *.*]

Deklaracja ***struktury przechowującej element drzewa*** (wraz z *kodem Huffmana*) może wyglądać tak:

```
struct node {
   unsigned char c;  /* przechowywany znak */
   unsigned count;   /* l.wystąpień */
   char code[NCHARS+1];  /* kod Huffmana, np. “010” */
   int len;   /* długość kodu Huffmana */
   struct node *left;   /* wskaźnik do lewego … */
   struct node *right;  /*  … i prawego potomka */
};
```

stałą NCHARS (==2^8) zdefiniowano: `#define NCHARS (01<<8)`

Dalej, definiujemy ***2 tablice wskaźników***, z których pierwsza jest *tablicą mieszającą*, druga zaś *uporządkowaną listą drzew*:

```
struct node *char_map[NCHARS+1], *char_tree[NCHARS+1];
```

Funkcja realizująca **zliczanie znaków** (poniżej) określa rozmiar pliku wejściowego (`*isize`) oraz kod ostatniego odnalezionego znaku (`*last_char` — *największa wartość kodu znaku*); zwraca liczbę *różnych* znaków pojawiających się na wejściu (`n`).

```
int CountChars(FILE *fnam, struct node *tab[],
   unsigned *isize, int *last_char)
{
   int c, j, n=0;

   *isize = *last_char = 0;
   while ( EOF != (c = getc(fnam)) ) {
     if ((*isize)++ >= UINT_MAX) {
       fprintf(stderr,
         "ERROR: maximum file size exceeded.\n");
       exit(2);
     }
```

```
      if (tab[c] == NULL) {
        /* Kod znaku jest zarazem pozycją w tablicy!! */
        tab[c] = NewNode(c);
        n++;
        if (c > *last_char)
          *last_char = c;
      }
      else {
        tab[c]->count++;
      }
    }
    return n;
}
```

Użyta powyżej funkcja tworząca ***pusty element drzewa*** zeruje także wszystkie liczniki:

```
struct node *NewNode(int c)
{
   struct node *t;

   t = (struct node*)malloc(sizeof(struct node));
   if (NULL == t) return NULL;
   t->c = c;
   t->count = 1;
   t->code[0] = '\0';
   t->len = 0;
   t->left = NULL;   /* LIŚĆ rozpoznamy po tym, że … */
   t->right = NULL;  /*  t->left == t->right == NULL */

   return t;
}
```

Dalej, mając tablicę mieszającą ***tworzymy uporządkowaną listę drzew***. Odbywa się to w dwóch krokach:

1) Kopiujemy (last+1) elementów z `char_map` do `char_tree`:

```
memcpy(char_tree, char_map,
       (last+1)*sizeof(struct node*));
```

2) sortujemy char_tree wywołując funkcję

```
SortCounts(char_tree, last+1);
```

która została tak napisana, aby ***wskaźniki puste*** (jeśli występują) trafiały zawsze na koniec listy (w dalszych krokach będą ignorowane). **Dodatkowo**, `SortCounts(…)` ma tę właściwość, że może zostać użyta na dowolnym etapie budowy drzewa — brane są pod uwagę wyłącznie *l.wystąpień zapisane w korzeniach*:

Sprawę załatwia wywołanie f.bibliotecznej **qsort**:

```
void SortCounts(struct node *tab[], int nchars)
{
   qsort(tab, nchars, sizeof(struct node *), comp);
}
```

przy czym ***porównywanie l.powtórzeń*** wykonuje funkcja:

```
int comp(const void *a, const void *b)
{
   struct node **px = (struct node **)a,
      **py = (struct node **)b;
   int x = (*px)?(*px)->count:0, /* Jeśli (*px==NULL)*/
      y = (*py)?(*py)->count:0;  /* wówczas x=0 (!) */
   return (y-x);
}
```

Łączenie 2 elementów listy o najmniejszych l.powtórzeń wykonujemy w pętli:

```
while (n-- > 1) {
   tab[n-1] = Join2Nodes(tab[n-1], tab[n]);
   tab[n] = NULL;
   SortCounts(tab, n);
}
```

przy czym operacja łączenia

```
   tab[n-1] = Join2Nodes(tab[n-1], tab[n]);
```

przebiega w taki sposób, że struktury wskazywane przez wskaźniki zapisane w tablicy `char_map` nie będą modyfikowane.

[*Tablica* `char_map` *zostanie później wykorzystana w procesie tłumaczenia znaków z pliku źródłowego na kody Huffmana.*]

Implementacja ***funkcji łączącej 2 elementy***:

```
struct node *Join2Nodes(struct node *left, struct node
*right)
{
  struct node *newnode = NewNode('\0');

  if (NULL == newnode) return NULL;
  newnode->count = left->count + right->count;
  newnode->left = left;
  newnode->right = right;
  return newnode;
}
```

Uwagi końcowe

Efektywność kompresji w kodowaniu Huffmana można ulepszyć modyfikując etap budowy drzewa np. tak, aby przechowywać *pary znaków*, lub inne (dłuższe) sekwencje.

Sytuacja znacząco się komplikuje, gdy chcemy ***kompresować dane wpływające ze strumienia wejściowego na bieżąco***, nie czekając z budową drzewa na zakończenie transmisji. Pojawia się wówczas problem aktualizacji (*przebudowy*) drzewa tak, aby długość kodów dopasowywać do *aktualnej* częstości występowania poszczególnych znaków.

[Przykładem algorytmu adaptacyjnego, pomyślanego tak, aby "nadążać za sytuacją" jest wspominany wcześniej **LZ77** (*Lempel-Ziv-1977*), który korzysta z bufora z podglądem oraz z okna przesuwnego.]

Literatura dla zainteresowanych

Więcej o zastosowaniach (`unicode` itp.):

https://cmps010-spring17-01.courses.soe.ucsc.edu/system/files/attachments/huffman_0.pdf

Więcej teorii — *K. Loudon, Mastering Algorithms with C,* rozdz.14.:

https://th.if.uj.edu.pl/~adamr/zadania/C/zbigniewrudy/MasterC.huffman1.pdf

Technikalia — nagłówki, kodowanie drzewa, itp.:

https://engineering.purdue.edu/ece264/17au/hw/HW13?alt=huffman