Rozdział 7.

Algorytmy przeszukiwania

Pojęcie „przeszukiwania" pojawiało się w tej książce już kilka razy w charakterze przykładów i zadań. Niemniej jest ono na tyle ważne, iż wymaga ujęcia w klamry osobnego rozdziału. Aby unikać powtórzeń, tematy już omówione będą zawierały raczej odnośniki do innych części książki niż pełne omówienia. Szczegółowej dyskusji zostanie poddana metoda *transformacji kluczowej*. Z uwagi na znaczenie i pewną odrębność tematu przeszukiwanie tekstów zostało zgrupowane w rozdziale kolejnym.

Przeszukiwanie liniowe

Temat przeszukiwania liniowego pojawił się już jako ilustracja pojęcia rekurencji. Iteracyjna wersja zaproponowanego tam programu jest oczywista — do jej wymyślenia nie jest nawet potrzebna znajomość treści rozdziału 6. Poniżej przedstawiony jest przykład przeszukiwania tablicy liczb całkowitych. Oczywiście metoda ta działa również w nieco bardziej złożonych przypadkach — modyfikacji wymaga jedynie funkcja porównująca x z aktualnie analizowanym elementem. Jeśli elementami tablicy są rekordy o dość skomplikowanej strukturze, to warto użyć jednej funkcji `szukaj`, która otrzymuje jako parametr wskaźnik do funkcji porównawczej[1].

Listing

linear.cpp

```
int szukaj(int tab[n], int x)
{
int i;
for( i=0; (i<n) && (tab[i]!=x ); i++);
return i;
}
```

Odnalezienie liczby `x` w tablicy `tab` jest sygnalizowane poprzez wartość funkcji; jeśli jest to liczba z przedziału *0..., n–1*, wówczas jest po prostu indeksem komórki, w której znajduje się `x`. W przypadku zwrotu liczby `n` jesteśmy informowani, iż element `x` nie został znaleziony. Zasada obliczania wyrażeń logicznych w C++ gwarantuje nam, że podczas analizy wyrażenia `(i<n) && (tab[i]!=x)` w momencie stwierdzenia fałszu pierwszego czynnika iloczynu logicznego reszta wyrażenia — jako niemająca znaczenia — nie będzie już sprawdzana. W konsekwencji nie będzie badana wartość spoza zakresu dozwolonych indeksów tablicy, co jest tym cenniejsze, iż kompilator C++ w żaden sposób o tego typu przeoczeniu by nas nie poinformował.

[1] Użycie wskaźników do funkcji zostało omówione na przykładzie w rozdziale 5.

Typ przeszukiwania, polegający na zwykłym sprawdzaniu elementu po elemencie, jest metodą bardzo wolno działającą, mamy tu do czynienia z klasą *O(n)*. Przeszukiwanie liniowe może być stosowane wówczas, gdy nie posiadamy żadnej informacji na temat struktury przeszukiwanych danych, ewentualnie sposobu ich składowania w pamięci.

Przeszukiwanie binarne

Jak już zostało zauważone w paragrafie poprzednim, ewentualna informacja na temat sposobu składowania danych może być niesłychanie użyteczna podczas przeszukiwania. W istocie często mamy do czynienia z uporządkowanymi już w pamięci komputera danymi: np. rekordami posortowanymi alfabetycznie, według niemalejących wartości pewnego pola rekordu, etc. Zakładamy zatem, że tablica jest posortowana, ale jest to dość częsty przypadek w praktyce, bowiem człowiek lubi mieć do czynienia z informacją uporządkowaną. W takim przypadku można skorzystać z naszej „meta-wiedzy" w celu usprawnienia przeszukiwania danych. Możemy bowiem łatwo wyeliminować z przeszukiwania te obszary tablicy, gdzie element `x` na pewno nie może się znaleźć. Dokładnie omówiony przykład *przeszukiwania binarnego* znalazł się już w rozdziale 2. — patrz zadanie 2. i jego rozwiązanie. W tym miejscu możemy dla odmiany podać iteracyjną wersję algorytmu:

binary-i.cpp

```
int szukaj(int tab[], int x)
{ //zwraca indeks poszukiwanej wartości lub -1
enum {TAK,NIE} Znalazlem=NIE;
int left=0, right=n-1, mid;
while( (left<=right) && (Znalazlem!=TAK) )
{
mid=(left+right)/2;
if(tab[mid]==x)
   Znalazlem=TAK;
else
   if(tab[mid]<x)
      left=mid+1;
   else
      right=mid-1;
}
if(Znalazlem==TAK)
   return mid;
else
   return -1;
}
```

Nazwy i znaczenie zmiennych są dokładnie takie same, jak we wspomnianym zadaniu, dlatego warto tam zerknąć choć raz dla porównania. Pewnej dyskusji wymaga problem wyboru elementu środkowego (`mid`). W naszych przykładach jest to dosłownie środek aktualnie rozpatrywanego obszaru poszukiwań. W rzeczywistości jednak może nim być oczywiście dowolny indeks pomiędzy `left` i `right`! Nietrudno jednak zauważyć, że dzielenie tablicy na pół zapewnia nam eliminację największego możliwego obszaru poszukiwań. Ich niepowodzenie jest sygnalizowane przez zwrot wartości `-1`. W przypadku sukcesu zwracany jest tradycyjnie indeks elementu w tablicy.

Przeszukiwanie binarne jest algorytmem klasy $O(\log_2 N)$ (patrz podpunkt „Rozkład logarytmiczny", rozdział 3.). Dla dokładnego uświadomienia sobie jego zalet weźmy pod uwagę konkretny przykład numeryczny:

Przeszukiwanie liniowe pozwala w czasie proporcjonalnym do rozmiaru tablicy (listy) odpowiedzieć na pytanie, czy element x się w niej znajduje. Zatem dla tablicy o rozmiarze 20 000 należałoby w najgorszym przypadku wykonać 20 000 porównań, aby odpowiedzieć na postawione pytanie. Analogiczny wynik dla przeszukiwania binarnego wynosi $log_2 20\,000$ (ok. *14* porównań).

Nic tak dobrze nie przemawia do wyobraźni, jak dobrze dobrany przykład liczbowy, a powyższy na pewno do takich należy.

Transformacja kluczowa (hashing)

Zanim powiemy choćby słowo na temat transformacji kluczowej[2], musimy sprecyzować dokładnie dziedzinę zastosowań tej metody. Otóż jest ona używana, gdy maksymalna liczba elementów należących do pewnej dziedziny[3] $\mathfrak{R}$ jest z góry znana (E_{max}), natomiast wszystkich możliwych (różnych) elementów tej dziedziny mogłoby być potencjalnie bardzo dużo (C). Tak dużo, że o ile przydział pamięci na tablicę o rozmiarze E_{max} jest w praktyce możliwy, o tyle przydział tablicy dla wszystkich potencjalnych C elementów dziedziny $\mathfrak{R}$ byłby fizycznie niewykonalny.

Dlaczego tak ważna jest uwaga o fizycznej niemożności przydziału takiej tablicy? Otóż gdyby taki przydział był realny, stworzylibyśmy bardzo dużą tablicę, w której elementy odnajdywane byłyby poprzez zwykłe adresowanie bezpośrednie i wszystkie funkcje szukania działałyby w czasie *O(1)*. Jak pamiętamy z rozdziału 3., notacja *O(1)* umownie oznacza, że liczba operacji wykonywanych przez algorytm jest niezależna od rozmiarów problemu. Teoretycznie wielkość zbioru do przeszukania nie będzie miała znaczenia, jeśli będziemy dysponowali pewną funkcją H, która pozwoli nam dotrzeć do poszukiwanego rekordu w mniej więcej tym samym, skończonym czasie.

Oto przykład ilustrujący sytuację:

- Chcemy zapamiętać $R_{max} = 250$ słów o rozmiarze *10* (tablica o rozmiarze $250 \cdot 11 = 2\,750$ bajtów jest w pełni akceptowalna[4].
- Wszystkich możliwych słów jest $C = 26^{10}$ (nie licząc w ogóle polskich znaków!). Praktycznie niemożliwe jest przydzielenie pamięci na tablicę, która mogłaby je wszystkie pomieścić.

Idea transformacji kluczowej polega na próbie odnalezienia takiej funkcji `H`[5], która — otrzymując w parametrze pewien zbiór danych — podałaby nam indeks w tablicy `T`, gdzie owe dane znajdowałyby się... gdyby je tam wcześniej zapamiętano!

Inaczej rzecz ujmując: transformacja kluczowa polega na odwzorowaniu:

$$dane \mapsto adres\ komórki\ w\ pamięci.$$

Zakładając taką organizację danych, położenie nowego rekordu w pamięci *teoretycznie* nie powinno zależeć od położenia rekordów już wcześniej zapamiętanych. Jak zapewne pamiętamy z rozdziału 5., nie był to przypadek list posortowanych, drzew binarnych, sterty itp. Naturalną konsekwencją nowego sposobu zapamiętywania danych jest maksymalne uproszczenie procesu po-

[2] Inne spotykane nazwy: mieszanie, rozpraszanie.

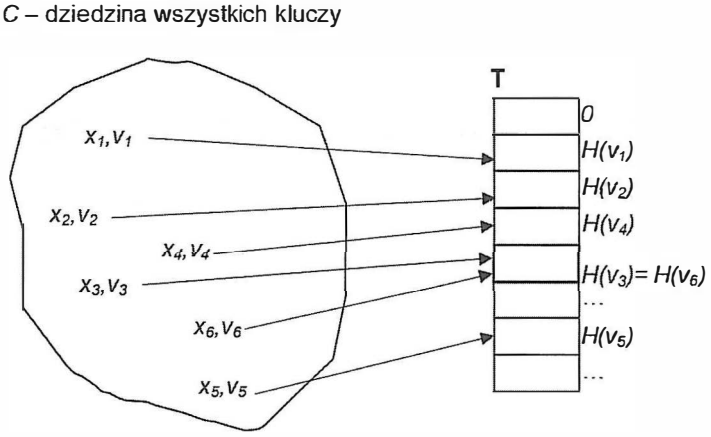
[3] „Dziedziny" w sensie matematycznym.

[4] Jeden dodatkowy bajt na znak '\0' kończący ciąg tekstowy w C++.

[5] Inna spotykana nazwa to „funkcja mieszająca".

szukiwania. Poszukując bowiem elementu x charakteryzowanego przez pewien klucz[6] v, możemy się posłużyć pewną znaną nam funkcją H. Obliczenie H(v) powinno zwrócić nam pewien adres pamięci, pod którym należy sprawdzić istnienie x, i do tego w sumie sprowadza się cały proces poszukiwania!

Idea transformacji kluczowej jest przedstawiona na rysunku 7.1.



Rysunek 7.1. *Idea transformacji kluczowej*

Załóżmy, że z całej dziedziny kluczy C interesuje nas wyłącznie sześć wartości kluczy: v_1, v_2, ..., v_6 (odpowiadają one elementom x_1, x_2, ..., x_6) i dysponujemy funkcją odwzorowującą wartość elementu na indeks tablicy T.

Element o kluczu v zostaje zapisany w tablicy T pod indeksem, który wyliczamy, aplikując funkcję H(v). Wartość NULL w tej tablicy oznacza brak elementu.

Ponieważ odwzorowujemy bardzo dużą dziedzinę C na mały zbiór elementów (wielkość tablicy T jest ograniczona), to w efekcie nieuniknione są kolizje odwzorowania wartości H(v) na indeks tablicy T. Na rysunku widać to na przykładzie elementów x_3 i x_6, o kluczach v_3 i v_6, dla których funkcja H wyliczyła taki sam indeks w tablicy! (Na szczęście nie jest to wada eliminująca metodę, co postaram się udowodnić w dalszej części rozdziału).

W metodzie transformacji kluczowej mamy do czynienia z zupełnym porzuceniem jakiegokolwiek procesu przeszukiwania zbioru danych: znając funkcję H, automatycznie możemy określić położenie dowolnego elementu w pamięci — wiemy również od razu, gdzie prowadzić ewentualne poszukiwania.

Czytelnik ma prawo w tym miejscu zadać sobie pytanie: to po co w takim razie męczyć się z listami, drzewami czy też innymi strukturami danych, jeśli można używać transformacji kluczowej? Jest to bardzo dobre pytanie, niestety odpowiedź na nie jest możliwa w jednym zdaniu. Wstępnie możemy tu podać dwa istotne powody ograniczające użycie tej metody:

- ograniczenia pamięci (trzeba z góry zarezerwować tablicę T na E_{max} elementów);
- trudności w odnalezieniu dobrej funkcji H.

O ile pierwszy powód jest oczywisty i może mniej istotny, to odnalezienie dobrej funkcji nie jest trywialne. A czym charakteryzuje się dobra funkcja H, dowiemy się w kolejnym podrozdziale.

---

[6] Pojęcie „klucza" pochodzi z teorii baz danych i jest dość powszechnie używane w informatyce; kluczem określa się zbiór atrybutów, które jednoznacznie identyfikują rekord (nie ma dwóch *różnych* rekordów posiadających *taką samą* wartość atrybutów kluczowych).

W poszukiwaniu funkcji H

Funkcja H na podstawie klucza v dostarcza indeksu w tablicy T, służącej do przechowywania danych. Potencjalnych funkcji, które na podstawie wartości danego klucza v zwrócą pewien adres adr, jest — jak się zapewne domyślamy — mnóstwo. Parametry, które mają główny wpływ na stopień skomplikowania funkcji H, to: długość tablicy, w której zamierzamy składować rekordy danych, oraz — bez wątpienia — wartość klucza v. Przed zamierzonym przystąpieniem do klawiatury, aby wklepać naprędce wymyśloną funkcję H, warto się dobrze zastanowić, które atrybuty rekordu danych zostaną wybrane do reprezentowania klucza. Logicznym wymogiem jest posiadanie przez tę funkcję następujących własności:

- Powinna być łatwo obliczalna, tak aby ciężaru przeszukiwania zbioru danych nie przenosić na czasochłonne wyliczanie H(v).
- Różnym wartościom klucza v powinny odpowiadać odmienne indeksy w tablicy T, tak aby nie powodować kolizji dostępu (np. elementy powinny być rozkładane w tablicy T równomiernie).
- Zastosowanie funkcji H powinno w efekcie spowodować równomierne i losowe rozmieszczenie elementów w tablicy T.

Pierwszy punkt nie wymaga komentarza, do kolejnych jeszcze powrócimy, gdyż stanowią one istotę metody transformacji kluczowej. W następnym paragrafie poznamy typowe metody konstruowania funkcji H. W rzeczywistych aplikacjach stosuje się przeróżne kombinacje cytowanych tam funkcji i w zasadzie nie można tu podać reguł postępowania. Często także wymagane jest eksperymentowanie i przeprowadzanie symulacji mających na celu odnalezienie dla zbioru danych o podanej charakterystyce funkcji H spełniającej podane wyżej warunki.

Najbardziej znane funkcje H

Najwyższa już pora zaprezentować kilka typowych funkcji matematycznych używanych do konstruowania funkcji stosowanych w transformacji kluczowej. Są to metody w miarę proste, jednak samodzielnie niewystarczające — w praktyce stosuje się raczej ich kombinacje niż każdą z nich osobno. Czytelnik, który z pojęciem transformacji kluczowej spotyka się po raz pierwszy, ma prawo być nieco zbulwersowany poniższymi propozycjami (modulo, mnożenie, etc.). Brakuje tu bowiem pewnej naukowej metody: nic nie jest do końca zdeterminowane, programista może w zasadzie wybrać, co mu się żywnie podoba, a algorytmy poszukiwania lub wstawiania danych będą i tak działały. W dalszych przykładach będziemy zakładać, że klucze są ciągami znaków, które można łączyć ze sobą i dość dowolnie interpretować jako liczby całkowite. Każdy znak alfabetu będziemy dla uproszczenia obliczeń w naszych przykładach kodować za pomocą 5 bitów (patrz tabela 7.1) — wybór kodu nie jest niczym zdeterminowany.

Tabela 7.1. *Przykład kodowania liter za pomocą 5 bitów*

A = 00001	B = 00010	C = 00011	D = 00100	E = 00101	F = 00110	G = 00111
H = 01000	I = 01001	J = 01010	K = 01011	L = 01100	M = 01101	N = 01111
O = 01110	P = 10000	Q = 10001	R = 10010	S = 10011	T = 10100	U = 10101
V = 10110	W = 10111	X = 11000	Y = 11001	Z = 11010		

Wspomniany wyżej brak metody jest na szczęście pozorny. Wiele podręczników algorytmiki błędnie prezentuje transformację kluczową, koncentrując się na tym JAK, a nie omawiając szczegółowo, PO CO chcemy w ogóle wykonywać operacje arytmetyczne na zakodowanych kluczach. Tymczasem sprawa jest względnie prosta:

- *Kodowanie* jest wykonywane w celu zamiany wartości klucza (niekoniecznie numerycznej!) na liczbę; sam kod jest nieistotny, ważne jest tylko, aby jako wynik otrzymać pewną liczbę, którą można później stosować w obliczeniach.

- Naszym celem jest możliwie jak najbardziej losowe rozmieszczenie rekordów w tablicy wielkości `M`: funkcja `H` ma nam dostarczyć w zależności od argumentu `v` adresy od `0` do `M-1`. Cały problem polega na tym, że nie jest możliwe uzyskanie losowego rozrzutu elementów, dysponując danymi wejściowymi, które z założenia nie są losowe. Musimy zatem uczynić coś, aby ową „losowość" w jakiś sposób *dobrze* zasymulować.

Badanie praktyczne dokonywane na dużych zestawach danych wejściowych wykazało, że istnieje grupa prostych funkcji arytmetycznych (`modulo`, `mnożenie`, `dzielenie`), które dość dobrze się do tego celu nadają[7]. Omówimy je kolejno w kilku paragrafach.

Suma modulo 2

Formuła: $H(v_1v_2... v_n) = v_1 \oplus v_2 \oplus ... \oplus v_n$

Przykład:

Dla

$R_{max} = 37$, H(„KOT") = $(\mathbf{01011}0111\mathbf{0100})_2$

daje

$(01011)_2 \oplus (01110)_2 \oplus (10000)_2 = (10001) = (17)_{10}$

Zalety:

- Funkcja `H` łatwa do obliczenia; `suma modulo 2`, w przeciwieństwie do iloczynu i sumy logicznej, nie powiększa (jak to czyni suma logiczna) ani nie pomniejsza (jak iloczyn) swoich argumentów.
- Używanie operatorów `&` i `|` powoduje akumulację danych odpowiednio na *początku* i na *końcu* tablicy `T`, czyli jej potencjalna pojemność nie jest efektywnie wykorzystywana.

Wady:

- Permutacje tych samych liter dają w efekcie identyczny wynik — można jednak temu zaradzić poprzez systematyczne przesuwanie cykliczne reprezentacji bitowej: pierwszy znak o jeden bit w prawo, drugi znak o dwa bity w prawo, etc.

Przykład:

- bez przesuwania H(„KTO") = $(01011)_2 \oplus (10100)_2 \oplus (01110)_2 = (17)_{10}$, jednocześnie H(„TOK") = $(17)_{10}$;
- z przesuwaniem H(„KTO") = $(10101)_2 \oplus (00101)_2 \oplus (11001)_2 = (9)_{10}$, natomiast H(„TOK") = $(01010)_2 \oplus (10011)_2 \oplus (01101)_2 = (10100)_2 = (20)_{10}$.

Suma modulo R_{max}

Formuła: $H(v) = v \% R_{max}$

Przykład:

Dla

$R_{max} = 37$: H(„KOT") = $(01011\mathbf{0111}\mathbf{0}10100)_2 \% (37)_{10} = (11732)_{10} = 3$.

Zalety:

- funkcja `H` łatwa do obliczenia.

[7] Jeśli zapomniałeś operacji arytmetycznych na wartościach logicznych, to polecam lekturę dodatku B.

Wady:

- Otrzymana wartość zależy — dość paradoksalnie — bardziej od R_{max} niż od klucza!

  Przykładowo: gdy R_{max} jest parzyste, *na pewno* wszystkie otrzymane indeksy danych o kluczach parzystych będą również parzyste, ponadto dla pewnych dzielników wiele danych otrzyma ten sam indeks. Można temu częściowo zaradzić poprzez wybór R_{max} jako liczby pierwszej, ale tu znowu będziemy mieli do czynienia z akumulacją elementów w pewnym obszarze tablicy
  — a wcześniej wyraźnie zażyczyliśmy sobie, aby funkcja `H` rozdzielała indeksy sprawiedliwie po całej tablicy!
- W przypadku dużych liczb binarnych niemieszczących się w reprezentacji wewnętrznej komputera obliczenie modulo już nie jest możliwe za pomocą zwykłego dzielenia arytmetycznego.

Co się tyczy ostatniej wady, to prostym rozwiązaniem dla ciągów tekstowych w C++ (wewnętrznie są to przecież zwykłe ciągi bajtów!) jest następująca funkcja, bazująca na interpretacji tekstu jako szeregu cyfr 8-bitowych:

```
int H(char *s, int Rmax)
{
for(int tmp=0; *s != '\0'; s++)
   tmp = (64*tmp+(*s)) % Rmax;
return tmp;
}
```

Mnożenie

Formuła: $H(v) = [(((v \cdot \theta) \% 1) \cdot E_{max})]$, gdzie $0 < \theta < 1$

Powyższą formułę należy odczytywać następująco: klucz v jest mnożony przez pewną liczbę θ z przedziału otwartego *(0, 1)*. Z wyniku bierzemy część ułamkową, mnożymy przez E_{max} i ze wszystkiego liczymy część całkowitą.

Istnieją dwie wartości parametru θ, które rozrzucają klucze w miarę równomiernie po tablicy:

$$\theta_1 = \frac{\sqrt{5}-1}{2} = 0{,}6180339887 \text{ oraz } \theta_2 = 1 - \theta_1 = 0{,}3819660113 .$$

Powyższa informacja jest prezentem od matematyków, a ponieważ *darowanemu koniowi nie patrzy się w zęby*, to nie będziemy zbytnio wnikać w kwestię, JAK oni to wynaleźli!

Przykład:

Dla

$\theta = 0{,}6180339887$, $E_{max} = 30$ i klucza $v =$ „*KOT*" $= 11\,732$ otrzymamy[8] $H($„*KOT*"$) = 23$.

Obsługa konfliktów dostępu

Kilka prostych eksperymentów przeprowadzonych z funkcjami zaprezentowanymi w poprzednim paragrafie prowadzi do szybkiego rozczarowania. Spostrzegamy, iż nie spełniają one założonych własności, co może łatwo skłonić do zwątpienia w sens całej prezentacji. Cóż, właściwie rzecz jest nieco bardziej złożona. Z jednej strony widzimy już, że idealne funkcje `H` nie istnieją[9], z drugiej

[8] Programowo można otrzymać tę wartość za pomocą instrukcji `int (fmod (11732 * 0.61803398887,1) * 30);` ponadto należy na początku programu dopisać `#include <math.h>`.

[9] Da je się to nawet uzasadnić teoretycznie (patrz np. dobrze znany w statystyce tzw. *paradoks urodzin*).

zaś strony dziwnym byłoby zaczynać dyskusję o transformacji kluczowej i doprowadzić ją do stwierdzenia, że... jej realizacja nie jest możliwa praktycznie! Oczywiście nie jest aż tak źle. Istnieje kilka metod, które pozwalają poradzić sobie w zadowalający sposób z zauważonymi niedoskonałościami, i one właśnie będą stanowić przedmiot naszych dalszych rozważań.

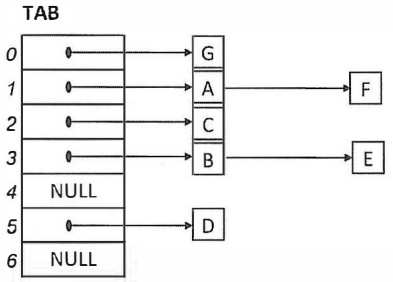
Powrót do źródeł

Co robić w przypadku stwierdzenia kolizji dwóch odmiennych rekordów, którym funkcja `H` przydzieliła ten sam indeks w tablicy `T`? Okazuje się, że można sobie poradzić poprzez pewną zmianę w samej filozofii transformacji kluczowej. Otóż jeśli umówimy się, że w tablicy `T` zamiast rekordów będziemy zapamiętywać *głowy* list do elementów charakteryzujących się tym samym kluczem, wówczas problem mamy... z głowy! Istotnie, jeśli wstawiając element `x` do tablicy pod indeks `m`, stwierdzimy, że już wcześniej ktoś się tam „zameldował", wystarczy doczepić `x` na koniec listy, której głowa jest zapamiętana w `T[m]`.

Analogicznie działa poszukiwanie: szukamy elementu `x` i `H(x)` zwraca nam pewien indeks `m`. W przypadku gdy `T[m]` zawiera `NULL`, możemy być pewni, że szukanego elementu nie odnaleźliśmy — w odwrotnej sytuacji, aby się ostatecznie upewnić, wystarczy przeszukać listę `T[m]`. (Warto przy okazji zauważyć, że listy będą na ogół bardzo krótkie).

Opisany powyżej sposób jest zilustrowany na rysunku 7.2.

Obrazuje on sytuację powstałą po sukcesywnym wstawianiu do tablicy `T` rekordów `A`, `B`, `C`, `D`, `E`, `F` i `G`, którym funkcja `H` przydzieliła adresy (indeksy): `1`, `3`, `2`, `5`, `3`, `1` i `0`. Indeksy tablicy, pod którymi nie ukrywają się żadne rekordy danych, są zainicjowane wartością `NULL` — patrz np. komórki `4` i `6`. Na pozycji `1` mamy do czynienia z konfliktem dostępu: rekordy `A` i `F` otrzymały ten sam adres! Odpowiednia funkcja `wstaw` (którą musimy przewidująco napisać!) wykrywa tę sytuację i wstawia element `F` na koniec listy `T[1]`.



Rysunek 7.2. *Użycie list do obsługi konfliktów dostępu*

Podobna sytuacja dotyczy rekordów `B` i `E`. Proces poszukiwania elementów jest zbliżony do ich wstawiania — Czytelnik nie powinien mieć trudności z dokładnym odtworzeniem sposobu przeszukiwania tablicy `T` w celu odpowiedzi na pytanie, czy został w niej zapamiętany dany rekord, np. `E`.

Co jest niepokojące w zaproponowanej powyżej metodzie? Zaprezentowana wcześniej idea transformacji kluczowej zawiera zachęcającą obietnicę porzucenia wszelkich list, drzew i innych skomplikowanych w obsłudze struktur danych na rzecz zwykłego odwzorowania:

$$dane \mapsto adres\ komórki\ w\ pamięci$$

Podczas dokładniejszej analizy napotkaliśmy jednak mały problem i... powróciliśmy do starych, dobrych list. Z tych właśnie przyczyn rozwiązanie to można ze spokojnym sumieniem uznać za nieco sztuczne[10] — równie dobrze można było trzymać się list i innych dynamicznych

[10] Choć parametry czasowe tej metody są bardzo korzystne.

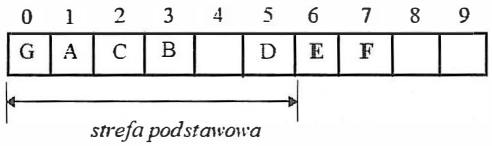
struktur danych, bez wprowadzania do nich dodatkowo elementów transformacji kluczowej! Czy możemy w tej sytuacji mieć nadzieję na rozwiązanie problemów dotyczących kolizji dostępu? Zainteresowanych odpowiedzią na to pytanie zachęcam do lektury następnych paragrafów.

Jeszcze raz tablice!

Metoda transformacji kluczowej została z założenia przypisana aplikacjom, które — pozwalając przewidzieć maksymalną liczbę rekordów do zapamiętania — umożliwiają zarezerwowanie pamięci na statyczną tablicę stwarzającą łatwy, indeksowany dostęp do nich. Jeśli możemy zarezerwować tablicę na wszystkie elementy, które chcemy zapamiętać, może by jej część przeznaczyć na obsługę konfliktów dostępu?

Idea polegałaby na podziale tablicy `T` na dwie części: strefę *podstawową* i strefę *przepełnienia*. Do tej drugiej elementy trafiałyby w momencie stwierdzenia braku miejsca w części podstawowej. Strefa przepełnienia byłaby wypełniana liniowo wraz z napływem nowych elementów „kolizyjnych". W celu zilustrowania nowego pomysłu spróbujmy wykorzystać dane z rysunku 7.1, zakładając rozmiar stref: *podstawowej* i *przepełnienia* na odpowiednio: `6` i `4`.

Efekt wypełnienia tablicy jest przedstawiony na rysunku 7.3.



Rysunek 7.3. *Podział tablicy do obsługi konfliktów dostępu*

Rekordy `E` i `F` zostały zapamiętane w momencie stwierdzenia przepełnienia na kolejnych pozycjach `6` i `7`. Sugeruje to, że gdzieś „w tle" musi istnieć zmienna zapamiętująca ostatnią wolną pozycję strefy przepełnienia.

Również w jakiś sposób należy się umówić co do oznaczania pozycji wolnych w strefie podstawowej — to już leży w gestii programisty i zależy w dużym stopniu od struktury rekordów, które będą zapamiętywane.

Rozwiązanie uwzględniające podział tablic nie należy do skomplikowanych, co jest jego niewątpliwą zaletą. Stworzenie funkcji `wstaw` i `szukaj` jest kwestią kilku minut i zostaje powierzone Czytelnikowi jako proste ćwiczenie.

Dla ścisłości należy jednak wskazać pewien słaby punkt. Otóż nie jest zbyt oczywiste, co należy zrobić w przypadku zapełnienia strefy... przepełnienia! (Wypisanie „ładnego" komunikatu o błędzie nie likwiduje problemu). Użycie tej metody powinno być poprzedzone szczególnie starannym obliczeniem rozmiarów tablic, tak aby nie załamać aplikacji w najbardziej niekorzystnym momencie — na przykład przed zapisem danych na dysk.

Próbkowanie liniowe

W opisanej poprzednio metodzie w sposób nieco sztuczny rozwiązaliśmy problem konfliktów dostępu w tablicy `T`. Podzieliliśmy ją mianowicie na dwie części służące do zapamiętywania rekordów, ale w różnych sytuacjach. O ile jednak dobór ogólnego rozmiaru tablicy R_{max} jest w wielu aplikacjach łatwy do przewidzenia, to dobranie właściwego rozmiaru strefy przepełnienia jest w praktyce bardzo trudne. Ważną rolę grają tu bowiem zarówno dane, jak i funkcja `H` i w zasadzie należałoby je analizować jednocześnie, aby w przybliżony sposób oszacować właściwe rozmiary obu części tablic. Problem oczywiście znika samoczynnie, gdy dysponujemy bardzo dużą ilością wolnej pamięci, jednak przewidywanie a priori takiego przypadku mogłoby być dość niebezpieczne.

Jak zauważyliśmy wcześniej, konflikty dostępu są w metodzie transformacji kluczowej nieuchronne. Powód jest prosty: nie istnieje idealna funkcja H, która rozmieściłaby równomiernie wszystkie R_{max} elementów po całej tablicy T. Jeśli taka jest rzeczywistość, to może zamiast walczyć z nią — jak to usiłowały czynić poprzednie metody — spróbować się do niej dopasować?

Idea jest następująca: w momencie zapisu nowego rekordu[11] do tablicy, w przypadku stwierdzenia konfliktu, możemy spróbować zapisać element w pierwszym wolnym miejscu. Algorytm funkcji wstaw byłby wówczas następujący (zakładamy próbę zapisu do tablicy T rekordu x charakteryzowanego kluczem v):

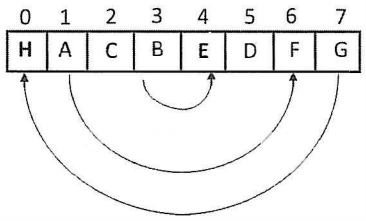
```
int pos=H(x.v);
while (T[pos] != WOLNE)
   pos = (pos+1) % Rmax;
T[pos]=x;
```

Załóżmy teraz, że poszukujemy elementu charakteryzującego się kluczem k. W takim przypadku funkcja szukaj mogłaby wyglądać następująco:

```
int pos=H(k);
while ((T[pos] != WOLNE) && (T[pos].v != k))
   pos = (pos+1) % Rmax;
return T[pos];    // zwraca znaleziony element
```

Różnica pomiędzy poszukiwaniem i wstawianiem jest w przypadku transformacji kluczowej doprawdy nieznaczna. Algorytmy są celowo zapisane w pseudokodzie, bowiem sensowny przykład korzystający z tej metody musiałby zawierać dokładne deklaracje typu danych, tablicy, funkcji H, wartości specjalnej WOLNE — analiza tego byłaby bardzo nużąca. Instrukcja pos = (pos+1) % Rmax; zapewnia nam powrót do początku tablicy w momencie dotarcia do jej końca podczas (kolejnych) iteracji pętli while.

Dla ilustracji spójrzmy, jak poradzi sobie nowa metoda przy próbie sukcesywnego wstawienia do tablicy T rekordów A, B, C, D, E, F, G i H, którym funkcja H przydzieliła adresy (indeksy): 1, 3, 2, 5, 3, 1, 7 i 7. Ustalmy ponadto rozmiar tablicy T na 8 — wyłącznie w ramach przykładu, bowiem w praktyce taka wartość nie miałaby zbytniego sensu. Efekt jest przedstawiony na rysunku 7.4:



Rysunek 7.4. *Obsługa konfliktów dostępu przez próbkowanie liniowe*

Dość ciekawymi jawią się teoretyczne wyliczenia *średniej liczby prób* potrzebnej do odnalezienia danej x. W przypadku poszukiwania zakończonego sukcesem średnia liczba prób wynosi około:

$$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right),$$

gdzie α jest współczynnikiem zapełnienia tablicy T. Analogiczny wynik dla poszukiwania zakończonego niepowodzeniem wynosi około:

$$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right).$$

[11] Oczywiście może to być również dowolna zmienna prosta!

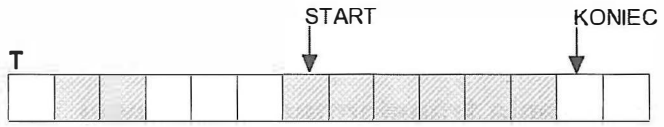
Przykładowo: dla tablicy zapełnionej w dwóch trzecich swojej pojemności ($\alpha = \frac{2}{3}$) liczby te wyniosą odpowiednio: *2* i *5*.

W praktyce należy unikać szczelnego zapełniania tablicy T, gdyż zacytowane powyżej liczby stają się bardzo duże (α nie powinno przybierać wartości bliskich *1*). Powyższe wzory zostały wyprowadzone przy założeniu funkcji H, która rozsiewa równomiernie elementy po dużej tablicy T. Te zastrzeżenia są tu bardzo istotne, bowiem podane wyżej rezultaty mają charakter statystyczny.

Podwójne kluczowanie

Stosowanie próbkowania liniowego prowadzi do niekorzystnego liniowego zapełniania tablicy T, co kłóci się z wymogiem narzuconym wcześniej funkcji H (patrz strona 169). Intuicyjnie rozwiązanie tego problemu nie wydaje się trudne: trzeba coś zrobić, aby nieco bardziej losowo porozrzucać elementy. Próbkowanie liniowe nie było z tego względu dobrym pomysłem, gdyż — napotkawszy pewien zapełniony obszar tablicy T — proponowało wstawienie nowego elementu tuż za nim — jeszcze go powiększając! Czytelnik mógłby zadać pytanie: a dlaczego jest to aż takie groźne? Oczywiście względy estetyczne nie grają tu żadnej roli: zauważmy, że liniowo zapełniony obszar przeszkadza w szybkim znalezieniu wolnego miejsca na wstawienie nowego elementu! Fakt ten utrudnia również sprawne poszukiwanie danych.

Rozpatrzmy prosty przykład przedstawiony na rysunku 7.5.



Rysunek 7.5. *Utrudnione poszukiwanie danych przy próbkowaniu liniowym*

Na rysunku tym zacieniowane komórki tablicy oznaczają miejsca już zajęte. Funkcja H(k) dostarczyła pewien indeks, od którego zaczyna się przeszukiwana strefa tablicy (poszukujemy pewnego elementu charakteryzującego się kluczem k) — powiedzmy, że zaczynamy poszukiwanie od indeksu oznaczonego symbolicznie jako START. Proces poszukiwania zakończy się sukcesem w przypadku trafienia w poszukiwany rekord — aby to stwierdzić, czynimy dość kosztowne[12] porównanie T[pos].v ≠ k (patrz algorytm procedury szukaj omówiony w poprzednim punkcie).

Co więcej, wykonujemy je za każdym razem podczas przesuwania się po liniowo wypełnionej strefie! Informację o ewentualnej porażce poszukiwań dostajemy dopiero po jej całkowitym sprawdzeniu i natrafieniu na pierwsze wolne miejsce. W naszym rysunkowym przykładzie dopiero po siedmiu porównaniach algorytm natrafi na pustą komórkę (oznaczoną etykietą KONIEC), która poinformuje go o daremności podjętego uprzednio wysiłku. Gdyby zaś tablica była zapełniona w mniej liniowy sposób, statystycznie o wiele szybciej natrafilibyśmy na WOLNE miejsce, co automatycznie zakończyłoby proces poszukiwania zakończonego porażką.

Na szczęście istnieje łatwy sposób uniknięcia liniowego grupowania elementów: tzw. *podwójne kluczowanie*. W chwili napotkania kolizji następuje próba rozrzucenia elementów za pomocą drugiej, pomocniczej funkcji H.

Procedura wstaw pozostaje niemal niezmieniona:

```
int pos =H1(x.v);
int krok=H2(x.v);
while (T[pos] != WOLNE)
```

[12] Koszt operacji porównania zależy od stopnia złożoności klucza, tzn. od liczby i typów pól rekordu, które go tworzą.

```
    pos = (pos+krok) % Rmax;
T[pos]=x;
```

Procedura poszukiwania jest bardzo podobna i Czytelnik z pewnością będzie w stanie napisać ją samodzielnie, wzorując się na przykładzie poprzednim.

Przedyskutujmy teraz problem doboru funkcji H2. Nie trudno się domyślić, iż ma ona duży wpływ na jakość procesu wstawiania (i oczywiście poszukiwania!). Przede wszystkim funkcja H2 powinna być różna od H1! W przeciwnym przypadku doprowadzilibyśmy tylko do bardziej skomplikowanego tworzenia stref „ciągłych" — a właśnie od tego chcemy uciec. Kolejny wymóg jest oczywisty: musi być to funkcja prosta, która nie spowolni nam procesu poszukiwania i wstawiania. Przykładem takiej prostej i jednocześnie skutecznej w praktyce funkcji może być H2(k) = 8 - (k % 8): zakres skoku jest dość szeroki, a prostota niezaprzeczalna!

Metoda *podwójnego kluczowania* jest interesująca z uwagi na widoczny zysk w szybkości poszukiwania danych. Popatrzmy na teoretyczne rezultaty wyliczeń średniej liczby prób przy poszukiwaniu zakończonym sukcesem i porażką. W przypadku poszukiwania zakończonego sukcesem średnia liczba prób wynosi około:

$$\frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)$$

(gdzie α jest, tak jak poprzednio, współczynnikiem zapełnienia tablicy T).

Analogiczny wynik dla poszukiwania zakończonego niepowodzeniem wynosi około:

$$\frac{1}{1-\alpha}.$$

Zastosowania transformacji kluczowej

Dotychczas obracaliśmy się wyłącznie w kręgu elementarnych przykładów: tablice o małych rozmiarach, proste klucze znakowe lub liczbowe itp. Rzeczywiste aplikacje mogą być oczywiście znacznie bardziej skomplikowane i dopiero wówczas Czytelnik będzie mógł w pełni docenić wartość posiadanej wiedzy. Zastosowania transformacji kluczowej mogą być dość nieoczekiwane: dane wcale nie muszą znajdować się w pamięci głównej; w przypadku programu bazy danych można w dość łatwy sposób użyć H-kodu do sprawnego odszukiwania danych. Konstruując duży kompilator lub konsolidator, możemy wykorzystać metody transformacji kluczowej do odszukiwania skompilowanych modułów w dużych plikach bibliotecznych.

Podsumowanie metod transformacji kluczowej

Transformacja kluczowa poddaje się dobrze badaniom porównawczym — otrzymywane wyniki są wiarygodne i intuicyjnie zgodne z rzeczywistością. Niestety sposób ich wyprowadzenia jest skomplikowany i ze względów czysto humanitarnych zostanie tu opuszczony. Mimo to ogólne wnioski o charakterze praktycznym są warte zacytowania:

- Przy słabym wypełnieniu[13] tablicy T wszystkie metody są w przybliżeniu tak samo efektywne.
- Metoda *próbkowania liniowego* doskonale sprawdza się przy dużych, słabo wykorzystanych tablicach T (czyli wówczas, gdy dysponujemy dużą ilością wolnej pamięci). Za jej stosowaniem przemawia również niewątpliwa prostota.

[13] Tzn. do ok. 30 – 40% całkowitej objętości tablicy.

Na koniec warto podkreślić coś, o czym w ferworze prezentacji rozmaitych metod i dyskusji mogliśmy łatwo zapomnieć: transformacja kluczowa jest narzędziem wprost idealnym... ale tylko w przypadku obsługi danych, których liczba jest z dużym prawdopodobieństwem przewidywalna. Nie możemy sobie bowiem pozwolić na załamanie się aplikacji z powodu naszych zbyt nieostrożnych oszacowań rozmiarów tablic!

Przykładowo: wiedząc, że będziemy mieli do czynienia ze zbiorem rekordów w liczbie ustalonej na przykład na `700`, deklarujemy tablicę `T` o rozmiarze `1000`, co zagwarantuje nam szybkie poszukiwanie i wstawianie danych nawet przy zapisie wszystkich `700` rekordów. Wypełnienie tablicy w 70 – 80% okazuje się tą magiczną granicą, po przekroczeniu której sens stosowania transformacji kluczowej staje się coraz mniej widoczny — dlatego po prostu nie warto zbytnio się do niej zbliżać. Niemniej metoda jest ciekawa i warta stosowania — oczywiście gdy uwzględnimy kontekst praktyczny aplikacji końcowej.


Uwaga

Kilka przykładowych funkcji `H` zostało zebranych w pliku *hash.cpp* — zachęcam do eksperymentów i symulowania ich funkcjonowania dla różnych zbiorów danych.