Marek Gągolewski
INSTYTUT BADAŃ SYSTEMOWYCH PAN
WYDZIAŁ MATEMATYKI I NAUK INFORMACYJNYCH POLITECHNIKI WARSZAWSKIEJ

Algorytmy
i podstawy programowania

5. Rekurencja. Abstrakcyjne typy danych

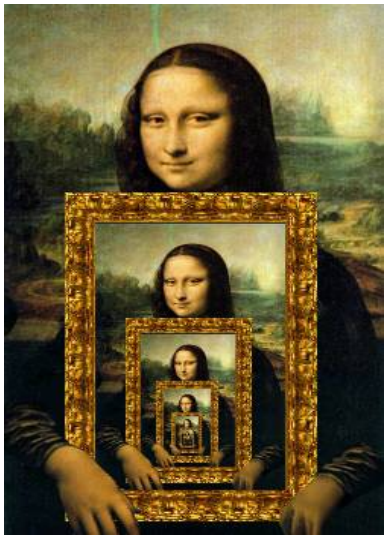
Materiały dydaktyczne dla studentów matematyki
na Wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
Ostatnia aktualizacja: 1 października 2016 r.

Spis treści

5.1. Rekurencja

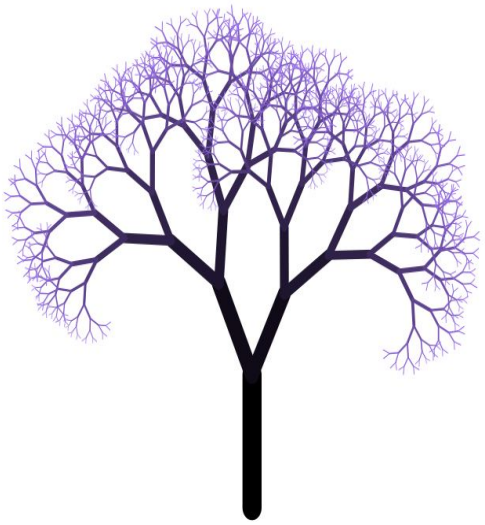
Z *rekurencją* (bądź rekursją, ang. *recursion*) mamy do czynienia wtedy, gdy w definicji pewnego obiektu pojawia się odniesienie do niego samego.

Rozważmy najpierw rys. 5.1. Przedstawia on pewną znaną, zacną damę, trzymającą obraz, który przedstawia ją samą trzymającą obraz, na którym znajduje się ona sama trzymająca obraz... Podobny efekt moglibyśmy uzyskać filmując kamerą telewizor pokazujący to, co właśnie nagrywa kamera.



Rys. 5.1. Rekurencyjna Mona Lisa

A teraz popatrzmy na rys. 5.2 przedstawiający prosty fraktal. Przypatrując się dłużej tej strukturze widzimy, że ma ona bardzo prostą konstrukcję. Wydaje się, że narysowanie kształtu takiego drzewa odbywa się następująco. Rysujemy odcinek o pewnej długości. Następnie obróciwszy się nieco w lewo/prawo, rysujemy nieco krótszy odcinek. W punktach, w których zakończyliśmy rysowanie, czynimy podobnie itd.



Rys. 5.2. Fraktalne drzewo

W przypadku języków programowania mówimy o rekurencji, *gdy funkcja wywołuje samą siebie*. Czyniąc tak jednak we „frywolny” sposób spowodowalibyśmy zawieszenie kom-

putera. Istotną cechą rekurencji jest więc dodatkowo *warunek stopu*, czyli zdefiniowanie przypadku, w którym rekurencyjne wywołanie zostaje przerwane.

Zapamiętaj

Każde wywołanie funkcji (także rekurencyjne) powoduje utworzenie nowego zestawu zmiennych lokalnych!

Rekurencja, jak widzimy, jest bardzo prostą techniką, której opanowanie pozwoli nam tworzyć bardzo ciekawe i często łatwe do zaprogramowania algorytmy. Wiele bowiem obiektów (np. matematycznych) jest właśnie ze swej natury zdefiniowana rekurencyjnie.

W poniższych paragrafach rozważymy kilka ciekawych przykładów takich zagadnień.

5.1.1. Przykład: silnia

W poniższej definicji silni pojawia się odniesienie do... silni. Zauważmy jednak, że obiekt ten jest dobrze określony, gdyż podany został warunek stopu.

$$\begin{cases} 0! &= 1, \\ n! &= n\,(n-1)! \quad \text{dla } n > 0. \end{cases}$$

Funkcję służącą do obliczenia silni można napisać opierając się wprost na powyższym wzorze.

```
1 int silnia(int n)
2 {
3     assert(n >= 0); // upewnijmy się...
4     if (n == 0) return 1;
5     else return n*silnia(n-1);
6 }
```

Dla porównania przyjrzyjmy się, jakby mogła wyglądać analogiczna funkcja niekorzystająca z dobrodziejstw rekurencji.

```
1 int silnia2(int n)
2 {
3     assert(n >= 0);
4     int w = 1;
5     for (int i=1; i<=n; ++i)
6         w *= i;
7     return w;
8 }
```

To, którą wersję uważamy za czytelniejszą, zależy oczywiście od nas samych.

5.1.2. Przykład: NWD

Poznaliśmy już algorytm wyznaczania największego wspólnego dzielnika dwóch liczb naturalnych. Okazuje się, że NWD można także określić poniższym równaniem rekurencyjnym. Niech $1 \leqslant n \leqslant m$. Wówczas:

$$\mathsf{NWD}(n,m) = \begin{cases} m & \text{dla } n = 0, \\ \mathsf{NWD}(m \bmod n, n) & \text{dla } n > 0. \end{cases}$$

Definicja ta przekłada się bezpośrednio na następujący kod w języku C++.

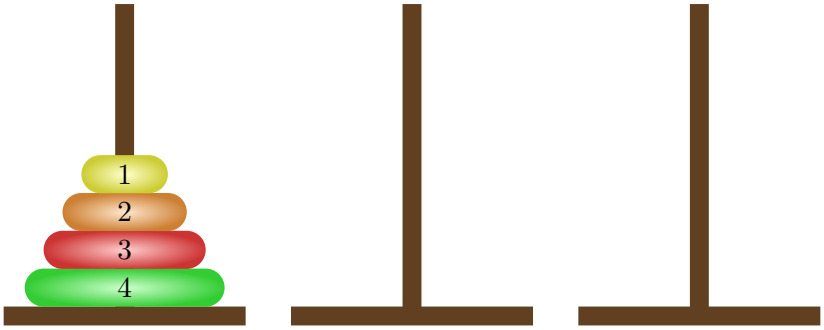
```
int nwd(int n, int m)
{
    assert(0 <= n && n <= m);
    if (n == 0) return m;
    else return nwd(m % n, n);
}
```

I w tym przypadku łatwo jest napisać równoważną wersję nierekurencyjną:

```
int nwd(int n, int m)
{
    assert(0 <= n && n <= m);
    while (n != 0) {
        int c = n;
        n = m % n;
        m = c;
    }
    return m;
}
```

5.1.3. Przykład: wieże z Hanoi

A teraz zabawna łamigłówka. Załóżmy, że danych jest n *krążków* o różnych średnicach oraz trzy *słupki*. Początkowo wszystkie krążki znajdują się na słupku nr 1, w kolejności od największego (dół) do najmniejszego (góra). Sytuację wyjściową dla $n = 4$ przedstawia rys. 5.3.



Rys. 5.3. Sytuacja wyjściowa dla 4 krążków w problemie wież z Hanoi.

Cel: przeniesienie wszystkich krążków na słupek nr 3.

Zasada #1: krążki przenosimy pojedynczo.

Zasada #2: krążek o większej średnicy nie może znaleźć się na krążku o mniejszej średnicy.

Zastanówmy się, jak by mógł wyglądać algorytm służący do rozwiązywania tego problemu. Niech będzie to funkcja `Hanoi(k, A, B, C)`, która przekłada `k` krążków ze słupka `A` na słupek `C` z wykorzystaniem słupka pomocniczego `B`, gdzie `A,B,C` $\in \{1, 2, 3\}$. Wykonywany ruch będzie wypisywany na ekranie.

Aby przenieść n klocków ze słupka `A` na `C`, należy przestawić $n-1$ mniejszych elementów na słupek pomocniczy `B`, przenieść n-ty krążek na `C` i potem pozostałe $n-1$ krążków przestawić z `B` na `C`. Jest to, rzecz jasna, sformułowanie zawierające elementy rekurencji.

Wywołanie początkowe: `Hanoi(n, 1, 2, 3)`.

Rozwiązanie w języku C++:

```
void Hanoi(int k, int A, int B, int C)
{
    if (k>0) // warunek stopu
    {
        Hanoi(k-1, A, C, B);
        cout << A << "->" << C << endl;
        Hanoi(k-1, B, A, C);
    }
}
```

Zauważmy, że stworzenie wersji nierekurencyjnej powyższej funkcji byłoby dla nas bardzo trudne – w przeciwieństwie do przykładów z silnią i NWD, gdzie rekurencja łatwo się „rozplątywała". Podczas bardziej zaawansowanego kursu algorytmiki dowiemy się, że każdą funkcję stosującą rekurencję możemy zapisać w postaci iteracyjnej przy użyciu pomocniczej struktury danych typu stos. Często jest to jednak zadanie nietrywialne i wynik pozostawia wiele do życzenia pod względem czytelności.

Ciekawostka

Interesującym zagadnieniem jest wyznaczenie liczby przestawień potrzebnych do rozwiązania łamigłówki. W zaproponowanym algorytmie jest to:

$$\begin{cases} L(1) &= 1, \\ L(n) &= L(n-1)+1+L(n-1) \text{ dla } n>0. \end{cases}$$

Można pokazać, że jest to optymalna liczba przestawień.

Rozwiążmy powyższe równanie rekurencyjne, aby uzyskać postać jawną rozwiązania:

$$\begin{aligned} L(n) &= 2L(n-1)+1, \\ L(n)+1 &= 2(L(n-1)+1). \end{aligned}$$

Zauważmy, że $L(n)+1$ tworzy ciąg geometryczny o ilorazie 2. Zatem

$$\begin{aligned} L(1)+1 &= 2, \\ L(2)+1 &= 4, \\ L(3)+1 &= 8, \\ &\vdots \\ L(n)+1 &= 2^n. \end{aligned}$$

Więc

$$L(n) = 2^n - 1.$$

Zagadka Wież z Hanoi stała się znana w XIX wieku dzięki matematykowi E. Lucasowi. Jak głosi tybetańska legenda, mnisi w świątyni Brahmy rozwiązują tę łamigłówkę przesuwając 64 złote krążki. Podobno, gdy skończą oni swe zmagania, nastąpi koniec świata. Zakładając jednak, że nawet gdyby wykonanie jednego ruchu zajmowało tylko 1 sekundę, to na pełne rozwiązanie i tak potrzeba $2^{64}-1 = 18446744073709551615$ sekund, czyli około 584542 miliardów lat. To ponad 400 razy dłużej niż szacowany wiek Wszechświata!

5.1.4. Przykład (niemądry): liczby Fibonacciego

Jako ostatni problem rozpatrzmy równanie, do którego doszedł Leonard z Pizy (1202), rozważając rozmnażanie się królików.

Sformułował on następujący uproszczony model szacowania liczby par w populacji. Na początku mamy daną jedną parę królików. Każda para osiąga płodność po upłynięciu jednej jednostki czasu od narodzenia. Z każdej płodnej pary rodzi się w kolejnej chwili jedna para potomstwa.

Liczbę par królików w populacji w chwili n można opisać za pomocą tzw. ciągu Fibonacciego.

$$\begin{cases} F_0 &= 1, \\ F_1 &= 1, \\ F_n &= F_{n-1} + F_{n-2} \text{ dla } n > 1. \end{cases}$$

W wyniku otrzymujemy zatem ciąg $(1, 1, 2, 3, 5, 8, 13, 21, 34, \dots)$.

Bezpośrednie przełożenie powyższego równania na kod w C++ może wyglądać jak niżej.

```
1 int fibrek(int n)
2 {
3     if (n > 1) return fibrek(n-1)+fibrek(n-2);
4     else return 1;
5 }
```

Powyższy algorytm rekurencyjny jest jednak bardzo nieefektywny! Przypatrzmy się, jak przebiegają obliczenia dla wywołania `fibrek(5)`:

```
fibrek(5);
   fibrek(4);
      fibrek(3);
         fibrek(2);
            fibrek(1); #
            fibrek(0); #
         fibrek(1); #
      fibrek(2);
         fibrek(1); #
         fibrek(0); #
   fibrek(3);
      fibrek(2);
         fibrek(1); #
         fibrek(0); #
      fibrek(1); #
```

Większość wartości jest niepotrzebnie liczona wielokrotnie.

Rozważmy dla przykładu, ile potrzebnych jest operacji arytmetycznych (dodawanie) do znalezienia wartości F_n za pomocą powyższej funkcji. Liczbę tę można opisać równaniem:

$$\begin{cases} L(0) &= 0, \\ L(1) &= 0, \\ L(n) &= L(n-1) + L(n-2) + 1 \text{ dla } n > 1. \end{cases}$$

Rozpatrując kilka kolejnych wyrazów, zauważamy, że $L(n) = F_{n-1} - 1$ dla $n > 0$.

Okazuje się, że dla dużych n, $L(n)$ jest proporcjonalne do c^n dla pewnej stałej c. Zatem liczba kroków potrzebnych do uzyskania n-tej liczby Fibonacciego algorytmem rekurencyjnym rośnie wykładniczo, tj. bardzo (naprawdę bardzo) szybko! Dla przykładu

czas potrzebny do policzenia F_{40} na komputerze autora tych refleksji to 1,4 s, dla F_{45} to już 15,1 s, a dla F_{50} to aż 2 minuty i 47 s.

Tym razem okazuje się, że implementacja stosująca rekurencję wprost jest wysoce nieefektywne. Przykład ten jest jednak bardzo pouczający dla matematyków – nie zawsze „przepisanie" definicji interesującego nas obiektu będzie prowadziło do dobrej w praktyce metody obliczeniowej[1]

Zatem w tym przypadku powinniśmy użyć raczej „zwykłego" rozwiązania iteracyjnego, które – co istotne – wykorzystuje tylko $O(n)$ operacji arytmetycznych.

Przykładowy pełny program wyznaczający n-tą liczbę Fibonacciego (dla danego z klawiatury n) mógłby wyglądać następująco.

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

int pobierz_n()
{
    int n;
    do
    {
        cout << "Podaj n: ";
        cin >> n;
        if (n < 0) {
            cout << "Niepoprawne dane! Spróbuj ponownie." << endl;
        }
    while (n < 0);

    return n; // tutaj na pewno n >= 0
}

int fibiter(int n)
{
    if (n <= 1) return 1; // tu wynik jest znany

    int k = 1;
    int Fk_poprz = 1;   // F_{k-1}
    int Fk_teraz = 1;   // F_k

    do {
        int nast = Fk_teraz+Fk_poprz; // policz kolejny wyraz
        k++;

        Fk_poprz = Fk_teraz;   // F_{k-1} (już dla nowego k)
        Fk_teraz = nast;       // F_k
    }
    while (k <= n);

    return Fk_teraz;
}


int main()
{
```

[1]Inny przykład: mimo że rozwiązanie układu równań postaci $Ax = b$ możemy poprawnie wyrazić za pomocą $x = A^{-1}b$, nigdy implementacja komputerowa nie powinna odwracać macierzy wprost. Podyktowane jest to tzw. stabilnością numeryczną metody – innymi słowy, z powodu niedoskonałości arytmetyki zmiennoprzecinkowej komputera, taki sposób obliczeń generuje często zbyt niedokładny wynik.

```
43      int n = pobierz_n();
44      cout << "F(" << n << ") = " << fibiter(n) << endl;
45      return 0;
46  }
```

Zwróćmy uwagę na podział programu na czytelne moduły (tj. funkcje), z których każdy służy do przeprowadzenia ściśle określonej czynności.

5.2. Podstawowe abstrakcyjne typy danych

W tym podrozdziale omówimy cztery następujące abstrakcyjne typy danych:

1. stos (rozdz. 5.2.1),
2. kolejkę (rozdz. 5.2.2),
3. kolejkę priorytetową (rozdz. 5.2.3),
4. słownik (rozdz. 5.2.4).

Zapamiętaj

Abstrakcyjny typ danych (ATD) umożliwia przechowywanie i organizowanie w specyficzny dla siebie sposób informacji danego, ustalonego typu.

Każdy abstrakcyjny typ danych określony jest za pomocą:

1. dziedziny, tj. typu zmiennych, które można w nim przechowywać,
2. właściwego sobie dopuszczalnego zbioru operacji, które można na nim wykonać.

Co ważne, definicje wszystkich abstrakcyjnych typów danych *abstrahują od sposobu ich implementacji*. Każdy zaprezentowany przez nas ATD może zostać zaprogramowany na wiele — z punktu widzenia realizowanej funkcjonalności — równoważnych sposobów. Jednakże owe implementacje będą różnić się m.in. efektywnością obliczeniową i pamięciową, które — jak wiemy — mają kluczowe znaczenie w praktyce. W trakcie kolejnych zajęć będziemy rzecz jasna starać się poszukiwać rozwiązań optymalnych. Najprostsze, tablicowe implementacje, pozostawiamy jako ćwiczenie.

Informacja

Dla uproszczenia umawiamy się, że każdy prezentowany abstrakcyjny typ danych będzie służył do przechowywania pojedynczych wartości całkowitych, tj. danych typu `int`.

Jako jeden z najbardziej przyjaznych matematykowi przykładów ATD można wymienić znany z teorii mnogości *zbiór* (u nas zwany *słownikiem*, zob. rozdz. 5.2.4). Otóż zbiór może przechowywać różne (odmienne) wartości (zgodnie z umową — całkowite). Dopuszczalne operacje wykonywane na nim to m.in. sprawdzenie, czy dany element znajduje się w zbiorze, dodanie elementu do zbioru i usunięcie elementu ze zbioru. Zauważmy, że wszystkie

inne operacje teoriomnogościowe (np. suma, iloczyn, różnica zbiorów, czy dopełnienie) mogą zostać zrealizowane za pomocą trzech wymienionych (aczkolwiek w sposób daleki od optymalnego pod względem szybkości działania[2]).

Zadanie

Zastanów się, w jaki sposób można najprościej i najefektywniej zaimplementować ATD typu zbiór (słownik) o elementach z $\{0, 1, \ldots, 100\}$, a w jaki o dowolnych elementach z $\mathbb{Z}$ z użyciem tablicy. Jaką pesymistyczną złożoność obliczeniową mają w każdym przypadku Twoje implementacje trzech wyżej wymienionych operacji? Czy da się je zrealizować w lepszy sposób?

5.2.1. Stos

Stos (ang. *stack*) jest abstrakcyjnym typem danych typu LIFO (ang. *last-in-first-out*), który udostępnia przechowywane weń elementy w kolejności odwrotnej niż ta, w której były one zamieszczane.

Stos udostępnia trzy operacje:

- *push* (umieść) — wstawia element na stos,
- *pop* (zdejmij) — usuwa i zwraca element ze stosu (LIFO!),
- *empty* (czy pusty) — sprawdza, czy na stosie nie znajdują się żadne elementy.

Przykład użycia stosu.

```
int main()
{
    Stos s = tworzStos();

    push(s, 3);
    push(s, 7);
    push(s, 5);

    while (!empty(s))
       cout << pop(s);
    // wypisze na ekran 573

    kasujStos(s);

    return 0;
}
```

[2] Więcej na ten temat dowiemy się z wykładu z *Algorytmów i struktur danych*. Zob. także książkę N. Wirtha „Algorytmy + struktury danych = programy", Wyd. WNT, Warszawa, 2001.

Zadanie

Zaimplementuj operacje wykorzystywane w powyższej funkcji `main()`. Stos reprezentuj w postaci struktury zawierającej dynamicznie alokowaną tablicę, na przykład:

```
struct Stos {
   int* wartosci;
   int n; // aktualny rozmiar tablicy
   int g; // indeks ,,wierzchołka'' stosu
};
```

Zaproponuj różne implementacje, m.in. przypadek, n zmienia się za każdym razem, gdy wykonujemy operacje *push* i *pop* oraz gdy n podwaja się, gdy dopiero zachodzi taka potrzeba. Jaką złożoność obliczeniową mają ww. operacje w pesymistycznym, a jaką w optymistycznym przypadku?

5.2.2. Kolejka

Kolejka (ang. *queue*) to ATD typu FIFO (ang. *first-in-first-out*). Podstawową własnością kolejki jest to, że pobieramy z niej elementy w takiej samej kolejności (a nie w odwrotnej jak w przypadku stosu), w jakiej były one umieszczane.

Kolejka udostępnia następujące trzy operacje:

- *enqueue* (umieść) — wstawia element do kolejki,
- *dequeue* (wyjmij) — usuwa i zwraca element z kolejki (FIFO!),
- *empty* (czy pusty).

Przykład użycia kolejki.

```
int main()
{
    Kolejka k = tworzKolejke();

    enqueue(k, 3);
    enqueue(k, 7);
    enqueue(k, 5);

    while (!empty(k))
       cout << dequeue(k);
    // wypisze na ekran tym razem 375

    kasujKolejke(k);

    return 0;
}
```

Zadanie

Zaimplementuj operacje wykorzystywane w powyższej funkcji `main()`. Kolejkę reprezentuj w postaci struktury zawierającej dynamicznie alokowaną tablicę. Jaką złożoność obliczeniową mają ww. operacje w pesymistycznym, a jaką w optymistycznym przypadku?

5.2.3. Kolejka priorytetowa

Kolejka priorytetowa (ang. *priority queue*) to abstrakcyjny typ danych typu LPF (ang. *lowest-priority-first*). Może ona służyć do przechowywania elementów, na których została określona pewna relacja porządku liniowego <=[3]. Wydobywa się z niej elementy poczynając od tych, które mają najmniejszy priorytet (są minimalne względem relacji <=). Jeśli w kolejce priorytetowej znajdują się elementy o takich samych priorytetach, obowiązuje dla nich kolejność FIFO, tak jak w przypadku zwykłej kolejki.

Omawiany ATD udostępnia trzy operacje:

- *insert* (włóż) — wstawia element na odpowiednie miejsce kolejki priorytetowej.
- *pull* (pobierz) — usuwa i zwraca element minimalny,
- *empty* (czy pusty).

Przykład użycia kolejki priorytetowej.

```
1  int main()
2  {
3      KolejkaPriorytetowa k = tworzKolejkePriorytetowa();
4
5      insert(k, 3);
6      insert(k, 7);
7      insert(k, 5);
8
9      while (!empty(k))
10         cout << pull(k);
11     // wypisze na ekran tym razem 357
12
13     kasujKolejkePriorytetowa(k);
14
15     return 0;
16 }
```

Zadanie

Zaimplementuj operacje wykorzystywane w powyższej funkcji `main()`. Kolejkę priorytetową reprezentuj w postaci struktury zawierającej dynamicznie alokowaną tablicę. Jaką złożoność obliczeniową mają ww. operacje w pesymistycznym, a jaką w optymistycznym przypadku?

5.2.4. Słownik

Słownik (ang. *dictionary*) to abstrakcyjny typ danych, który udostępnia trzy operacje:

- *search* (znajdź) — sprawdza, czy dany element znajduje się w słowniku,
- *insert* (dodaj) — dodaje element do słownika, o ile taki już się w nim nie znajduje,
- *remove* (usuń) — usuwa element ze słownika.

Dla celów pomocniczych rozważać możemy także następujące działania na słowniku:

- *print* (wypisz),
- *removeAll* (usuń wszystko).

[3] W naszym przypadku, jako że umówiliśmy się, iż omawiane ATD przechowują dane typu `int`, najczęściej będzie to po prostu zwykły porządek $\leqslant$.

Informacja

Czasem zakłada się, że na przechowywanych elementach została określona relacja porządku liniowego `<=`. Wtedy możemy tworzyć bardziej efektywne implementacje słownika, np. z użyciem drzew binarnych (zob. rozdz. 7) albo algorytmu wyszukiwania binarnego (połówkowego) w przypadku tablic.

Pesymistyczna złożoność obliczeniowa algorytmu wyszukiwania binarnego wynosi zaledwie $O(\log n)$ dla danej uporządkowanej tablicy `t` rozmiaru n. Metoda ta bazuje na założeniu, że poszukiwany element `x`, jeśli oczywiście znajduje się w tablicy, musi być ulokowany na pozycji (indeksie) ze zbioru `{l,l+1,...,p}`. Na początku ustalamy oczywiście `l=0` i `p=n-1`. W każdej kolejnej iteracji znajdujemy „środek" owego obszaru poszukiwań, tzn. `m=(p+l)/2` oraz sprawdzamy, jak ma się `t[m]` do `x`. Jeśli okaże się, że jeszcze nie trafiliśmy na właściwe miejsce, to wtedy zawężamy (o co najmniej połowę) nasz obszar modyfikując odpowiednio wartość `l` bądź `p`.

```
bool search(int x, int* t, int n)
{
   if (n <= 0) return false;

   // wyszukiwanie binarne (połówkowe)
   int l = 0;      // początkowy obszar poszukiwań
   int p = n-1;    //          - cała tablica
   while (l <= p)
   {
      int m = (p+l)/2; // "środek" obszaru poszukiwań

      if (x == t[m]) return true; // znaleziony - koniec
      else if (x > t[m]) l = m+1; // zawężamy obszar poszukiwań
      else               p = m-1; // jw.
   }

   return false; // nieznaleziony
}
```

Powyższą funkcję łatwo zmodyfikować, by zwracała indeks, pod którym znajduje się znaleziony element bądź jakąś wyróżnioną wartość (np. -1), gdy elementu nie ma w tablicy (potrafisz?). Spróbuj także samodzielnie zaimplementować rekurencyjną wersję tego algorytmu.

Przykład użycia słownika.

```
int main()
{
   Slownik s = tworzSlownik();

   int i, j;
   cout << "Podaj wartości, które maja się znaleźć w słowniku. ";
   cout << "Wpisz wartość ujemną, by zakończyć wstawianie. ";

   cin >> i;
   while (i >= 0) {
      insert(s, i);
      cin >> i;
   }

```

```
15      cout << "Podaj wartości, które chcesz znaleźć w słowniku. ";
16      cout << "Wpisz wartość ujemną, by zakończyć wyszukiwanie. ";
17
18      cin >> i;
19      while (i >= 0) {
20         if (search(s, i)) {
21            cout << "Jest!" << endl;
22            cout << "Usunąć? 0 == NIE." << endl;
23            cin >> j;
24            if (j != 0)
25               remove(s, i);
26         }
27         else
28            cout << "Nie ma!" << endl;
29         cin >> i;
30      }
31
32      kasujSlownik(s);
33
34      return 0;
35  }
```

5.3. Ćwiczenia

Zadanie 5.1. Zdefiniuj strukturę `Wektor2d` o dwóch składowych typu `double`. Utwórz funkcję `dlugosc()`, która dla danego obiektu typu `Wektor2d` zwraca jego normę euklidesową. Dalej, stwórz funkcję `inssort()`, która implementuje algorytm sortowania przez wstawianie danej tablicy o elementach typu `Wektor2d*` (zamianę elementów implementujemy wprost na wskaźnikach). Napisz też funkcję `main()`, w której wszystko zostanie dokładnie przetestowane.

Zadanie 5.2. Napisz funkcję `horners()`, która wyznacza wartości wszystkich k podanych wielomianów w podanym punkcie. Wielomian reprezentujemy przy użyciu struktury `Wielomian` (składającej się z tablicy `w` o elementach typu `double` oraz stopnia wielomianu `n`). Deklaracja funkcji do napisania: `double* horners(Wielomian* W, int k, double x)`.

Zadanie 5.3 (MD)**.** Zdefiniuj strukturę `Zbior`, która reprezentuje podzbiór zbioru liczb naturalnych (pola: tablica `e` o elementach typu `int` oraz liczba elementów `n`).

Napisz funkcję `void dodaj_element(Zbior& zb, int e)`, która doda do wskazanego zbioru element `e` (jeżeli element już znajduje się w zbiorze, funkcja nie powinna robić nic).

Ponadto napisz funkcję `Zbior przeciecie(Zbior* zbiory, int k)`, która wyznaczy przecięcie wszystkich zbiorów zadanych w postaci `k`-elementowej tablicy `zbiory`, a także `Zbior suma(Zbior* zbiory, int k)`, która oblicza sumę podanych zbiorów.

Zadanie 5.4 (MB)**.** Zdefiniuj strukturę `Ulamek` o dwóch składowych `licznik`, `mianownik` typu `int` – reprezentuje ona liczbę wymierną. Napisz funkcje służące do dodawania, odejmowania, mnożenia i dzielenia ułamków, każdą postaci `Ulamek nazwa_funkcji(Ulamek x, Ulamek y)`. Po wykonaniu każdej z operacji ułamki powinny być odpowiednio znormalizowane; patrz algorytm NWD (zalecana metoda: przez dodatkową funkcję `normalizuj()`).

Następnie napisz funkcje, które mając dane tablice ułamków wyznaczą ich skumulowane minimum, maksimum, sumę i iloczyn. Napisz też funkcje, które wyznaczają sumę, różnicę, iloczyn i iloraz ułamków z dwóch zadanych tablic (odpowiednio zwektoryzowane). Dodatkowo, możesz stworzyć funkcję, która sortuje ułamki względem ich wartości oraz wypisuje ułamki na ekranie.

Zadanie 5.5 (MB)**.** Dana jest struktura zdefiniowana jako `struct Naukowiec {int *cytowania; int n; };`. Zakładamy, że `cytowania` są zawsze posortowane nierosnąco.

Napisz funkcję, która dodaje nową publikację z podaną liczbą cytowań do tablicy `cytowania`, np. dla tablicy $(5, 4, 4, 1)$ po dodaniu 3 otrzymujemy $(5, 4, 4, 3, 1)$ (zwracamy uwagę na poprawną alokację nowej tablicy oraz dealokację starej).

Utwórz także funkcję `h()`, która wyznacza indeks Hirscha danego naukowca.

Dodatkowo, zaimplementuj następujące funkcje, które na wejściu otrzymują tablicę naukowców: (a) wyznaczanie naukowca o największym indeksie Hirscha, (b) sortowanie naukowców pod względem indeksu h (od najlepszego do najgorszego).