Treść

Artykuły

Struktury danych 1
Wstęp 2
Konwencje 3
Tablice 4
Listy 5
Stosy 8
Kolejki 12
Drzewa 12
Złożoność obliczeniowa 12
Implementacje w C++ 16
Implementacje w Pascalu 16
Bibliografia 17
Dla twórców podręcznika 17

Przypisy

Źródła i autorzy artykułu 21
Źródła, licencje i autorzy grafik 22

Licencje artykułu

Licencja 23

Struktury danych

Witaj w podręczniku poświęconym tworzeniu i używaniu struktur danych. Ich znajomość ma fundamentalne znaczenie, jeśli chodzi o programowanie oraz projektowanie algorytmów. Jeśli czujesz się na siłach, pomóż w jego rozwoju, uzupełniając brakujące rozdziały! **Wcześniej jednak zapoznaj się z rozdziałem Dla twórców podręcznika**, gdzie znajdziesz ustalenia edycyjne dla autorów.

Spis treści

Wstęp

1. Wstęp
2. Konwencje

Struktury danych

1. Tablice
2. Listy
3. Stosy
4. Kolejki
5. Drzewa
6. Zbiory
7. Kopce
8. Struktura Find-Union
9. Tablice haszujące
10. Grafy

Dodatki

1. Złożoność obliczeniowa
2. Implementacje w C++
3. Implementacje w Pascalu
4. Bibliografia
5. Dla twórców podręcznika

Wstęp

Spis treści	
Wstęp	Wstęp - Konwencje
Struktury danych	Tablice - Listy - Stosy - Kolejki - Drzewa - Zbiory - Tablice haszujące - Grafy
Dodatki	Złożoność obliczeniowa - Implementacje w C++ - Implementacje w Pascalu - Bibliografia - Dla twórców podręcznika

Wstęp

Niemal każdy algorytm, każdy program, operuje na różnorodnych danych. Mają one przeważnie określoną formę zapewniającą mu pożądane właściwości. Do ich przechowywania i dalszej obróbki potrzebne jest ich zapamiętanie oraz stworzenie dodatkowych algorytmów zapewniających dostęp do wszystkich elementów oraz umożliwiających modyfikację zawartości zbioru. Struktury danych, którym w całości poświęcony jest ten podręcznik, są właśnie takimi zaawansowanymi pojemnikami na dane, które gromadzą je i układają w odpowiedni sposób. Ich różnorodność jest ogromna, a dla każdej znaleziono wiele zastosowań oraz interesujących algorytmów. Powszechnie spotykane jest używanie jednych struktur danych do przetwarzania informacji zgromadzonych w innych. Można więc powiedzieć, że są one fundamentalnym narzędziem programisty i ich znajomość jest niemal niezbędna.

Struktury danych znajdziemy wszędzie. Znajdują się w ogromnych bazach danych z milionami rekordów, gdzie pomagają w ich segregacji oraz przyspieszają dostęp do potrzebnych informacji. Odtwarzacz audio zainstalowany na twoim komputerze wykorzystuje je do przechowywania listy ulubionych utworów. Obecne są nawet w komputerowych procesorach, gdzie asystują przy wykonywaniu kolejnych rozkazów oraz wywołaniach procedur. Każdy program bardziej skomplikowany, niż *Hello world* czy prosty kalkulator, jest zmuszony do korzystania z nich w jakimś stopniu, już chociażby do prostego zapamiętania analizowanych danych w pamięci komputerowej. Dlatego ich znajomość jest właśnie tak istotna. Jeżeli pragniesz zgłębiać algorytmikę, dogłębne poznanie struktur danych to jedna z pierwszych rzeczy, od których powinieneś zacząć.

Dzięki temu podręcznikowi poznasz najważniejsze z istniejących struktur, dowiesz się o ich wadach oraz zaletach oraz nauczysz się implementować je w twoich programach. Staraliśmy się tak dobrać opisy, aby prezentowały zagadnienie zarówno od strony praktycznej, jak i teoretycznej. Podręcznik adresowany jest do wszystkich osób, które zainteresowały się algorytmiką lub które pragną uporządkować swą wiedzę w tej dziedzinie. Jeśli dopiero zaczynasz przygodę z algorytmami, będziesz miał okazję zaznajomienia się z bardziej naukowym podejściem do problemów informatycznych. Zakładamy, że masz także pewne pojęcie o złożoności obliczeniowej, jednak jeżeli widzisz ten termin po raz pierwszy w życiu, przygotowaliśmy dla Ciebie specjalny poświęcony jej rozdział. Na zakończenie pozostaje nam jedynie życzyć miłej i owocnej lektury.

Konwencje

Spis treści	
Wstęp	Wstęp - Konwencje
Struktury danych	Tablice - Listy - Stosy - Kolejki - Drzewa - Zbiory - Tablice haszujące - Grafy
Dodatki	Złożoność obliczeniowa - Implementacje w C++ - Implementacje w Pascalu - Bibliografia - Dla twórców podręcznika

Konwencje

Aby jak najbardziej uczynić treść podręcznika przyjazną dla czytelnika, przyjęliśmy kilka konwencji, które omówimy w tym rozdziale. Opis każdej struktury danych trzyma się mniej więcej ustalonego schematu:

1. Ilustracja zagadnienia lub problemu, na jaki możemy natknąć się w codziennej praktyce
2. Wstępna prezentacja struktury danych, w zwięzły i jasny sposób opisująca ideę jej działania
3. Naukowy opis struktury, z wykazem właściwości oraz operacji, jakie można na niej wykonywać
4. Sposoby implementacji najważniejszych fragmentów struktury danych
5. Dodatkowe informacje
6. Ćwiczenia podzielone na zbiór podstawowy oraz zaawansowany

Do prezentacji algorytmów używamy bazującego na Pascalu pseudokodu, w którym dla czytelności niektóre partie kodu zastąpiliśmy słownym opisem. Właściwą implementację w językach Pascal oraz C++ czytelnik może znaleźć w dodatkach lub podjąć próbę napisania jej samodzielnie.

Tekst wzbogacony jest dodatkowymi ramkami informacyjnymi:

	Ramki informacyjne prezentują dodatkowe informacje warte uwagi.

	Porada Czytaj uważnie dodatkowe ramki informacyjne, ponieważ zawierają one użyteczne informacje.

	Ramka informacyjna to umieszczony w tekście obiekt z ramką, kolorowym tłem oraz ikonką, który prezentuje warte zapamiętania dodatkowe informacje. Ten rodzaj ramki zawiera różne definicje.

	Uwaga! Nieczytanie ramek informacyjnych może skutkować nieprzewidzianymi przez autorów podręcznika konsekwencjami!

Pamiętaj także, że podręcznik ten znajduje się dopiero w fazie rozwoju, stąd też niektóre jego fragmenty są niekompletne lub nawet nienapisane. Prosimy w takim wypadku o cierpliwość, a osoby z odpowiednimi chęciami i wiedzą, które nie widzą przeszkód w licencji GNU FDL, o pomoc w rozwoju tego podręcznika.

Tablice

Spis treści	
Wstęp	Wstęp - Konwencje
Struktury danych	Tablice - Listy - Stosy - Kolejki - Drzewa - Zbiory - Tablice haszujące - Grafy
Dodatki	Złożoność obliczeniowa - Implementacje w C++ - Implementacje w Pascalu - Bibliografia - Dla twórców podręcznika

Tablice

Załóżmy, że nasz program wczytał pewien zbiór danych tego samego typu, do którego elementów potrzebny jest szybki dostęp oraz o którym wiemy, że nie będzie modyfikowany. Do jego przechowywania najlepiej nadają się **tablice** - pierwsza, a zarazem najprostsza ze struktur danych. Jest więcej, niż prawdopodobne, że spotkałeś się już z nimi, ponieważ praktycznie każdy język programowania oferuje dla nich wbudowane wsparcie. Każdy element tablicy posiada swój unikalny **indeks**, który określa jego położenie w tablicy. Oto prosty przykład tablicy jako zbioru argumentów pewnej funkcji matematycznej:

Indeks	0	1	2	3	4	5	6	7
Wartość	6	2	5	4	3	3	8	1

Indeks może pełnić tutaj rolę argumentu funkcji.

Istnieją także tablice wielowymiarowe, w których do identyfikacji elementu używany jest więcej, niż jeden indeks. Przykładem tablicy dwuwymiarowej może być np. szkolna tabliczka mnożenia.

Cechy tablic

Charakterystyczne cechy tablic to:

- Stały rozmiar deklarowany przy tworzeniu tablicy, co znacznie utrudnia jej wykorzystanie w przypadku zmieniających się zbiorów, do których dodajemy bądź usuwamy elementy.
- Do każdego elementu gwarantowany dostęp w stałym czasie - dzięki konieczności wstępnego określenia rozmiaru tablica może zajmować ciągły obszar pamięci. W ten sposób odnalezienie elementu o podanym indeksie sprowadza się do prostego przemnożenia rozmiaru elementów przez indeks i odczytaniu danych spod otrzymanego adresu pamięci.

Na tablicach można wykonać następujące operacje:

- *CREATE(t: Array; s:Integer)* - tworzy tablicę *t* liczącą sobie *s* elementów
- *DESTROY(t: Array;)* - usuwa tablicę *t*
- *READ(t: Array; i: Integer)* - odczytuje wartość elementu o indeksie *i*
- *WRITE(t: Array; i: Integer; d: Data)* - zapisuje wartość *d* ("Data" możesz zastąpić przez dowolny typ, jaki potrzebujesz) do elementu tablicy o indeksie *i*

Zauważ, że nie mamy tutaj żadnej operacji "zerującej" zawartość tablicy. Wynika to z tego, iż w myśl definicji tablica posiada zawsze stałą liczbę elementów. Jeżeli masz do przechowania mniej informacji, możesz wykonać to na kilka sposobów:

- Dodatkowa zmienna przechowująca ilość faktycznie wykorzystanych elementów tablicy - rozwiązanie to wymaga, aby zajęte elementy były ułożone w tablicy w sposób ciągły, a niewykorzystane znajdowały się zawsze na końcu, jednak zawsze mamy szybki dostęp do informacji o wykorzystaniu tablicy.

- Wybranie jakiejś specyficznej wartości, np. *0* na reprezentowanie "pustych" elementów, które można wykorzystać. Dane mogą być przechowywane wtedy w sposób nieciągły.

Łącząc oba sposoby, łączymy jednocześnie ich zalety.

Implementacja

Tablice są wbudowane w niemal wszystkie języki programowania i w zasadzie nie trzeba ich samodzielnie implementować. Należy mieć na uwadze pewne różnice w implementacjach, np. Pascal dopuszcza możliwość ustalenia zakresu indeksów, podczas gdy w C/C++ elementy indeksowane są zawsze od 0. Język C++ dodatkowo posiada bardzo wygodną w użyciu i szybką implementację tablic przy pomocy pojemników (np. *vector*). Niektóre języki skryptowe posunęły się dalej i zlikwidowały ograniczenie stałego rozmiaru tablicy oraz typu indeks pozwalając na używanie tzw. tablic asocjacyjnych (np. PHP, który automatycznie potrafi dostosować rozmiar do ilości elementów). W rzeczywistości jednak ich interpretery implementują takie tablice jako bardziej złożone struktury danych, np. tablice haszujące.

Ćwiczenia

Ćwiczenie 1: Napisz funkcję do zmiany rozmiaru tablicy poprzez utworzenie nowej kopii oraz przeniesienie do niej danych. Zastanów się, jak będzie się ona zachowywać, jeżeli nowy rozmiar będzie mniejszy od ilości już przechowywanych w niej elementów? Jak będzie się zachowywać, jeżeli dopuścisz możliwość nieciągłego ułożenia elementów?

Listy

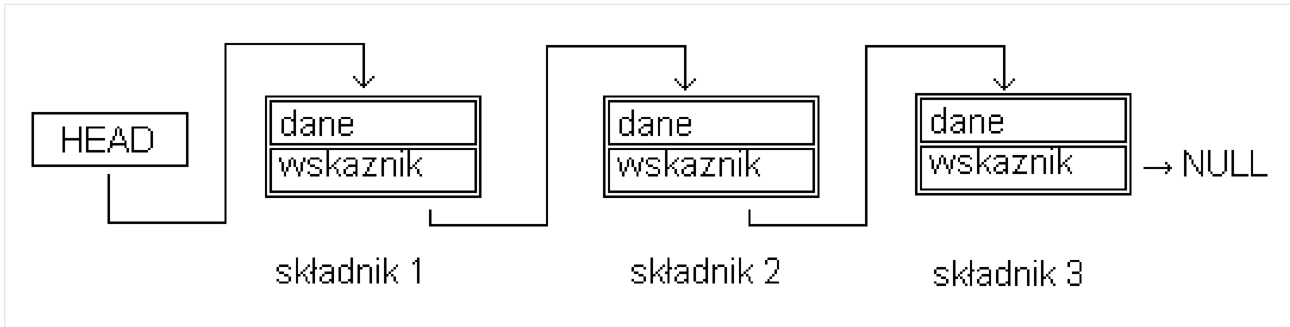
Spis treści	
Wstęp	Wstęp - Konwencje
Struktury danych	Tablice - Listy - Stosy - Kolejki - Drzewa - Zbiory - Tablice haszujące - Grafy
Dodatki	Złożoność obliczeniowa - Implementacje w C++ - Implementacje w Pascalu - Bibliografia - Dla twórców podręcznika

Listy

Poznane w poprzednim rozdziale tablice doskonale nadają się do przechowywania nieznacznie zmieniających się danych. Jednak weźmy na warsztat program do odtwarzania muzyki. Zazwyczaj wyposażony jest on w playlistę będącą zbiorem naszych ulubionych utworów, które chcemy odsłuchiwać. Playlista może mieć dowolną długość: można na niej trzymać zarówno 20, jak i 2000 piosenek, w dodatku w każdej chwili mamy możliwość dodania dowolnej ilości nowych. Ciężko sobie wyobrazić, by program przydzielał przy uruchomieniu kilka megabajtów "na wszelki wypadek", aby zabezpieczyć się przed sytuacją braku miejsca. Jednym ze sposobów rozwiązania tego problemu jest wykorzystanie nowej struktury danych - *list*. Ich zasadniczą cechą jest to, że zawsze zajmują w pamięci dokładnie tyle miejsca, ile potrzeba i zawsze można bezproblemowo dodać do nich nowe elementy.

Budowa listy

Każdy rekord w liście, oprócz przechowywanych przez siebie danych, zawiera również przynajmniej jedno dodatkowe pole zawierające adres/położenie następnego w kolejności rekordu. Powstaje w ten sposób łańcuch, w którym rekordy znają jedynie swoje następniki. Ilustruje to poniższy schemat:



Oprócz tego musimy wydzielić też specjalny rekord zawierający tzw. *korzeń listy*, czyli odnośnik do pierwszego elementu. Ostatni rekord na liście posiada pusty wskaźnik, dzięki czemu możliwe jest powiadomienie o końcu łańcucha. Zauważmy, że w takiej strukturze bezpośredni dostęp do dowolnego elementu jest niemożliwy. Odszukanie interesującego nas rekordu zajmuje rosnącą liniowo ilość czasu. Najpierw musimy ustawić się na początku listy, a następnie przechodzić do kolejnych elementów, dopóki nie natrafimy na nasz właściwy.

Pokazana powyżej lista jest w zasadzie tylko jedną z jej odmian, zwaną *listą jednokierunkową*, ponieważ możemy po niej poruszać się tylko w jednym kierunku: do przodu. Inne rodzaje list to:

- *Listy dwukierunkowe* - każdy rekord zawiera dodatkowo drugi wskaźnik pokazujący *poprzedni* element listy.
- *Listy cykliczne* - pierwszy i ostatni rekord listy są ze sobą połączone, tworząc zamknięty cykl.

Na liście można wykonywać następujące operacje:

- *CREATE(l: List)* - tworzy nową listę
- *INSERT(l: List; d: Data)* - dodaje na końcu listy *l* rekord *d*. Zauważmy, że w przedstawionej powyżej naiwnej implementacji listy operacja ta wymaga czasu liniowego, ponieważ tyle zajmuje dotarcie na jej koniec. Istnieje jednak możliwość zaimplementowania tej operacji działającej w stałym czasie. Wystarczy, aby nagłówek listy przechowywał także adres ostatniego z rekordów.
- *DELETE(l: List; n: Integer)* - usuwa *n*-ty w kolejności rekord z listy. Usunięcie pierwszego i ostatniego rekordu można wykonać w czasie stałym. Dla pozostałych będzie to czas liniowy.
- *FIND(l: List; n: Integer)* - pobiera *n*-ty w kolejności rekord z listy.
- *RESET(l: List, n: Integer)* - ustawia kursor na podanym rekordzie listy.
- *NEXT(l: List)* - zwraca rekord, na którym aktualnie ustawiony jest kursor, po czym przesuwa się na kolejny element listy.
- *PREV(l: List)* - analogiczna operacja do *NEXT()* dla list dwukierunkowych umożliwiająca przesuwanie się w tył.
- *MAKENULL(l: List)* - czyści listę, usuwając wszystkie rekordy.

W operacjach wyszukiwania elementów na liście przyjmujemy, że *1* identyfikuje pierwszy rekord na liście, *2* drugi (itd.), *n-1* przedostatni, *n* ostatni.

Operacje *RESET()*, *NEXT()* oraz *PREV()* umożliwiają samodzielne poruszanie się po liście. Na początek musimy ustawić się tam, gdzie potrzebujemy, a następnie dwoma pozostałymi operacjami przesuwamy się w żądanym przez nas kierunku, pobierając po kolei wszystkie napotkane tam rekordy.

Lista jednokierunkowa o *n* rekordach zajmuje w pamięci $n(4 + d) + h$ bajtów przy założeniu, że *d* oznacza wielkość danych rekordu, *h* wielkość nagłówka listy, a wskaźniki są czterobajtowe (długość charakterystyczna dla 32-bitowych aplikacji).

Implementacja

Pierwszą z omówionych implementacji list będzie implementacja tablicowa. Do przechowywania rekordów wykorzystywane są tu poznane w poprzednim rozdziale tablice. Wskaźniki rekordów są tutaj zwyczajnymi indeksami wskazującymi komórkę, pod którą znajduje się kolejny rekord. Nasuwa się tutaj pytanie, jaki jest sens implementacji listy za pomocą tablic. Wbrew pozorom pomysł ten jest całkiem sensowny. Przypomnij sobie zaproponowane w poprzednim rozdziale sposoby na oznaczanie w tablicach "wolnych" komórek. Listy na tablicach to jedna z metod rozwiązania tego problemu.

Alternatywą dla tablic jest implementacja wskaźnikowa, gdzie wykorzystuje się wskaźniki z prawdziwego zdarzenia oraz dynamiczny przydział pamięci dla nowych rekordów. Posiada ona wszystkie opisane wyżej cechy: zajmuje dokładnie tyle pamięci, ile potrzebujemy, oraz umożliwia przechowywanie dowolnie dużej ilości danych. Aby prawidłowo wybrać metodę implementacji, zastanówmy się, jakie są nasze potrzeby. Listy na tablicach są często wybierane przez uczestników olimpiad informatycznych. Pojawiające się tam zadania dokładnie precyzują rozmiar maksymalnych danych oraz dostępną do wykorzystania pamięć. Nie bez znaczenia jest tu również fakt, że alokacja pamięci dla jednego ogromnego bloku jest znacznie szybsza, niż dla setek małych o identycznym sumarycznym rozmiarze.

Do zrobienia:
Pokazać przykładowe implementacje najważniejszych operacji (na wskaźnikach)

Ćwiczenia

Podstawowe

Ćwiczenie 1: Napisz program wypisujący wszystkie elementy listy.

Ćwiczenie 2: Napisz pełną implementację listy:

1. Na tablicach
2. Na wskaźnikach

Ćwiczenie 3: Zastanów się, jak w liście dwukierunkowej o długości n zmniejszyć pesymistyczny czas wyszukiwania rekordu x z czasu n do $n/2$. Wprowadź stosowne poprawki do implementacji.

Do zrobienia:
Ułożyć jeszcze trochę ćwiczeń podstawowych

Zaawansowane

Do zrobienia:
Ułożyć ćwiczenia zaawansowane

Stosy

Zastosowanie

Stos jest dobrym rozwiązaniem, gdy gromadząc jakieś dane przeprowadzamy operacje najpierw na tych najnowszych (ostatnio dodanych), a gdy te nie są nam już potrzebne, to zabieramy się za te nieco starsze.

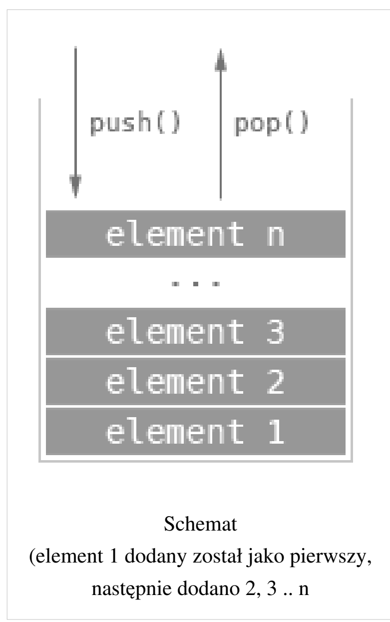
Dobrym przykładem jest tu stos talerzy, ułożonych jeden na drugim, które - załóżmy - przypadło nam zmywać. W pierwszej kolejności zabierzemy się za talerz na samej górze. Gdy już go wyczyścimy to odłożymy na bok i weźmiemy następny - tak do ostatniego który jest na samym dnie stosu.

Opis

Zważywszy na kolejność dodawania i usuwania elementów ze stosu, struktura ta określana jest jako **LIFO** (ang. ***L***ast ***I***n ***F***irst ***O***ut - *Ostatni na wejściu, Pierwszy na wyjściu*).

Operacje na stosie

- **push()** - wrzuca nowy element na szczyt stosu
- **pop()** - zdejmuje element ze szczytu stosu zwracając jego wartość



Schemat
(element 1 dodany został jako pierwszy, następnie dodano 2, 3 .. n

Implementacja

Do implementacji stosu o stałym rozmiarze *n* wystarczy *n*-elementowa tablica i jedna zmienna, która będzie przetrzymywać aktualną liczbę elementów.

Python

Do emulacji działania stosu wykorzystamy listę i dwie jej metody: **append()** oraz **pop()**.

```
stos = [] # tworzymy listę

stos.append(1) # metoda append() działa identycznie jak opisywany push()
stos.append(2)
stos.append(3)

# w tym momencie na stosie są 3 elementy
print "Na stosie sa " + str(len(stos)) + " elementy"

x = stos.pop() # zdejmujemy przed chwilą dodane elementy ze stosu
y = stos.pop()
z = stos.pop()
```

C/C++

```
#define STOS_MAX 10 // stos 10-elementowy

int stos[STOS_MAX];
int szczyt=0;

push(element) {
      if(szczyt < STOS_MAX) {
            /* wrzuć na stos */
            stos[szczyt] = element;
            szczyt++;
      }
      else {
            /* stos jest już pełny */
            ...
      }
}

pop() {
      if(szczyt != 0) {
            /* zdejmij ze stosu */
            szczyt--;
            return stos[szczyt];
      }
      else {
            /* stos jest już pusty */
            ...
      }

}
```

Pascal (FPC)

```
const
      StosMax = 10; //Stos 10 elementowy

var
      stos : array[1..StosMax] of integer;
      szczyt : integer = 0;

procedure push(element : integer);
begin
      if szczyt < StosMax then
            begin
             //wrzuć na stos
                   Inc(szczyt);
             stos[szczyt] := element;
            end
```

```
      else
            begin
                  writeln('Stos jest pelny'); //Stos jest pełny
            end;
end;

function pop : integer;
begin
      if szczyt <> 0 then
            begin
                  //zdejmij ze stosu
                  pop := stos[szczyt];
                        Dec(szczyt);
            end
      else
            begin
                  writeln('Nie ma juz nic na stosie'); //Stos jest pusty
                        pop := -1; //-1 lub inna wartość kontrolna (wartownik) który poinformuje o
                        //pustym stosie (najlepiej wybrać wartość którą będzie unikalna dla końca
                        //stosu, tzn żaden inny element na stosie nie będzie mógł jej osiągnąć)
            end;
end;
```

Java

```
class Stos {

      private int[] wektor;
      private int top;

      public Stos() {
            this(10);
      }

      public Stos(int rozmiar) {
            wektor = new int[rozmiar];
            top = -1;
      }

      public void push(int element) {
            wektor[++top] = element;
      }

      public int pop() {
            return wektor[top--];
      }

      public boolean czyPusty() {
```

```
            return (top == -1);
      }

      public boolean czyPelny() {
            return (top == wektor.length);
      }

      public int peek() {
            return wektor[top];
      }

      public void wyswietl() {
         for(int i = 0; i <= top; i++)
         System.out.print(wektor[i] + " ");
         System.out.println();
      }

      public int rozmiar() {
            return top + 1;
      }

}
```

Ćwiczenia

Do zrobienia:

- Rozwinąć opis
- Implementacja w języku Pascal
- Implementacja w języku Java

Kolejki

Kolejka - To liniowo uporzadkowany w którym mamy dostep tylko do dwóch składnikow . Operacje wykonywane na kolejce to dołożenie nowego składnika do kolejki (wyłacznie na koniec) badz usuniecie składnika z początku kolejki

Drzewa

1. Drzewa wyszukiwań binarnych
2. Drzewa czerwono-czarne
3. Drzewa AVL
4. Drzewa przedziałowe
5. Drzewa czwórkowe

Złożoność obliczeniowa

Spis treści	
Wstęp	Wstęp - Konwencje
Struktury danych	Tablice - Listy - Stosy - Kolejki - Drzewa - Zbiory - Tablice haszujące - Grafy
Dodatki	Złożoność obliczeniowa - Implementacje w C++ - Implementacje w Pascalu - Bibliografia - Dla twórców podręcznika

Złożoność obliczeniowa

Wybierając konkretny algorytm do rozwiązania problemu, programista powinien wiedzieć, czego może się po nim spodziewać oraz jak jego zastosowanie wpłynie na działanie aplikacji. Często trzeba wybierać spośród kilku możliwych algorytmów i wtedy konieczne jest wprowadzenie jakiegoś kryterium pozwalającego precyzyjnie określić, który z nich jest "najlepszy". Operując na dużych zbiorach danych, najbardziej korzystne jest przyjęcie definicji, w myśl której najlepszy algorytm najefektywniej wykorzystuje dostępne zasoby komputera (pamięć oraz procesor) do jak najszybszego rozwiązania problemu i zaprezentowania wyników. W ten sposób odpowiedzieliśmy sobie na pytanie, czym jest **złożoność obliczeniowa**

	Złożoność obliczeniowa jako dział teorii obliczeń zajmuje się określaniem ilości zasobów (np. pamięci, czasu, liczby procesorów) niezbędnych do rozwiązania problemu obliczeniowego.

Pomiar szybkości algorytmu

Czas wykonania danego algorytmu zależy od wielu czynników:

1. Jakości kodu napisanego przez programistę i wygenerowanego przez kompilator
2. Szybkości sprzętu, na którym jest on wykonywany
3. Rozmiaru danych wejściowych
4. Użytego algorytmu

Jakość kodu oraz sprzętu to czynniki czysto subiektywne, które czasem ciężko jest nawet dokładnie określić. Jeśli przyjmiemy, że wszystkie nasze algorytmy badamy na tym samym komputerze i kompilujemy tym samym kompilatorem, oba te czynniki będą mogły zostać wyrażone przez pewną stałą, którą oznaczymy sobie literą *c*. Idźmy dalej - skoro szybkość algorytmu zależy od rozmiaru danych wejściowych, od razu powinno nasunąć to nam podejrzenia, że szybkość algorytmu możemy wyrazić za pomocą pewnej funkcji matematycznej, której argumentem jest rozmiar danych wejściowych *n*. Na przykład w przypadku sortowania, *n* może określać liczbę elementów, które chcemy posortować. Ostateczna postać funkcji zależy od użytego algorytmu.

Przyjęło się, że czas wykonywania algorytmu oznaczany jest przez *T(n)*. Jednostka, w jakiej podawany jest wynik, jest nieważna. Można przyjąć, że jest to liczba instrukcji, jakie musi wykonać komputer. Oto przykład:

$$T(n) = cn^2$$

c, jak wspomnieliśmy, charakteryzuje tutaj jakość kodu oraz parametry komputera. Wzór ten możemy odczytać następująco: jeżeli liniowo zwiększamy ilość danych, czas wykonywania algorytmu rośnie kwadratowo. Inny algorytm rozwiązujący ten sam problem może być opisany wzorem $T(n) = cn$. Poniżej prezentujemy tabelkę pokazującą, jak wydłuża się czas działania obu algorytmów w zależności od rozmiaru danych wejściowych:

n	1	2	3	4	5	6	7	8	9	10
$T(n) = cn^2$	*c*	*4c*	*9c*	*16c*	*25c*	*36c*	*49c*	*64c*	*81c*	*100c*
$T(n) = cn$	*c*	*2c*	*3c*	*4c*	*5c*	*6c*	*7c*	*8c*	*9c*	*10c*

Z tabelki jasno widać, że drugi z algorytmów cechuje się wyższą wydajnością. Dla danych rozmiaru *7* wykona się on szybciej, niż algorytm pierwszy dla danych rozmiaru *3*. Dla ostatniego argumentu w tabelce - 10 - jest on dziesięciokrotnie szybszy, a im dalej będziemy szli, tym bardziej różnica będzie się pogłębiać.

Okazuje się, że nie zawsze rozmiar jest jedynym czynnikiem, od którego zależy szybkość działania. Niektóre algorytmy bowiem dla pewnych szczególnych danych mogą np. znacząco przyspieszyć, a z kolei inne dane mogą być przetwarzane na nich o wiele dłużej, niż wynika to z naszych obliczeń. W takim przypadku *T(n)* opisuje najgorszy przypadek zwany też *pesymistycznym czasem wykonania* - jest to jednak sytuacja skrajna, wobec czego można także zdefiniować najlepszy możliwy przypadek lub średni czas wykonywania. Nie dajmy się przy tym zwieść pozorom. Stawianie założenia, że wszystkie rodzaje danych są jednakowo prawdopodobne, może prowadzić do nieprzewidzianych rezultatów.

Notacja "dużego O" i "dużej Ω"

Załóżmy, że mamy dwa algorytmy, o których wiemy, że:

$$T_1(n) = c(n+1)^2$$

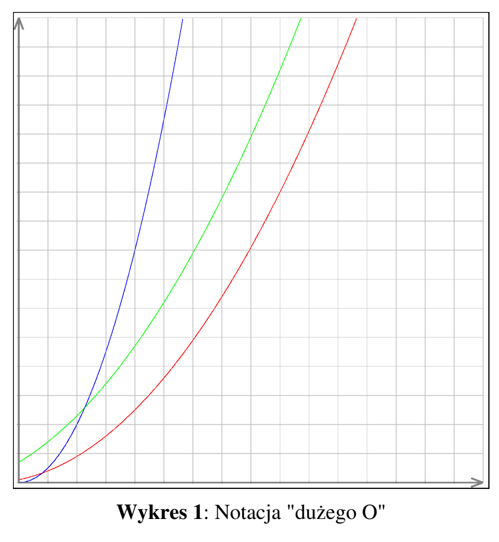
$$T_2(n) = c((n+3)^2 - 2)$$

Oczywiście można tutaj bez większych problemów określić, który z nich jest lepszy, po prostu obliczając wartości obu z nich dla kilku dowolnych argumentów. Jednak na dłuższą metę posługiwanie się takimi wzorami bywa po prostu uciążliwe, nie mówiąc już o niezwykłym zadaniu ich wyznaczenia poprzez studiowanie opisu algorytmu. Przy bardziej rozbudowanych algorytmach sprawa komplikuje się do tego stopnia, że możemy mieć nawet trudności z określeniem jakości każdego z nich. Dlatego do porównywania złożoności algorytmów stosuje się nieco inną

notację zwaną notacją "dużego O".

Zapis **T(n) = O(f(n))** oznacza, że istnieją takie stałe dodatnie *c* oraz n_0, że dla każdego $n \geq n_0$ zachodzi $T(n) \leq cf(n)$.

Mówiąc mniej matematycznym językiem, zawsze jesteśmy w stanie znaleźć taką stałą *c* oraz taki argument n_0, że dla wszystkich kolejnych argumentów "czas" wykonywania algorytmu opisanego funkcją T(n) będzie zawsze mniejszy lub równy od "czasu" opisanego funkcją f(n).



Wykres 1: Notacja "dużego O"

Spróbujmy to przenieść na grunt podanych powyżej dwóch funkcji. Dla $T_1(n)$ mamy $T_1(n) = O(n^2)$ oraz $c = 5, n_0 = 1$. Faktycznie, dla każdego *n* większego lub równego 1 zachodzi nam nierówność $(n+1)^2 \leq 5n^2$. Dla $T_2(n)$ mamy także $T_2(n) = O(n^2)$, ponieważ dla *c = 5* i $n_0 = 3$ zachodzi nierówność $(n+3)^2 - 2 \leq 5n^2$. Okazuje się zatem, że oba te algorytmy charakteryzują się tym samym rzędem złożoności, zatem dla identycznego rozmiaru danych powinny wykonywać się w podobnym czasie. Funkcja *f(n)* stanowi tutaj górne ograniczenie tempa wzrostu (innymi słowy: algorytm nigdy nie wykona się wolniej), co pokazuje wykres 1. Czerwona krzywa to pierwsza z przedstawionych powyżej funkcji, zielona - druga, a niebieska to funkcja $f(n) = n^2$.
Na oznaczenie dolnej granicy tempa wzrostu (algorytm nigdy nie wykona się szybciej) wprowadzona została notacja "dużej Ω", której definicja jest następująca:

Zapis **T(n) = Ω(g(n))** oznacza, że istnieje taka stała dodatnia *c*, że dla nieskończenie wielu *n* zachodzi $T(n) \geq cg(n)$.

Rząd złożoności

Dobór algorytmu o odpowiedniej złożoności zależy od kilku czynników, m.in. rozmiaru danych oraz mocy komputera. Jeśli mamy dane dwa algorytmy o złożoności sześciennej: $5n^3$ oraz kwadratowej: $100n^2$, to mimo pozornej przewagi drugiego z nich, zauważmy, że dla n < 20 działający w czasie sześciennym algorytm okaże się szybszy! Naukowcy zajmujący się algorytmami oraz teorią obliczeń podzielili problemy na kilkanaście klas złożoności. Najważniejsze z nich to:

- *Problemy P* - są to problemy łatwo obliczalne, ich rozwiązanie można znaleźć w czasie wielomianowym (np. n^2, n^6).
- *Problemy NP* - są to problemy trudno obliczalne, ich rozwiązanie można najwyżej *sprawdzić* w czasie wielomianowym (ale już nie rozwiązać w tym czasie).

Wiadomo, że wszystkie problemy P są też problemami NP, natomiast w drugą stronę sprawa jest nierozstrzygnięta. Jest to jedno z wielkich nierozwiązanych zagadnień matematycznych. Wśród problemów NP można wyróżnić jeszcze kilka podklas. Na szczególną uwagę zasługują problemy NP-zupełne. Ich właściwością jest to, że można do niego w czasie wielomianowym zredukować każdy inny problem NP-zupełny. Znaczy to tyle, że gdyby ktoś znalazł algorytm rozwiązujący w czasie wielomianowym jeden problem NP-zupełny, automatycznie dałoby się w podobnym czasie rozwiązać je wszystkie. Dotychczas jednak wszystkie wymyślone algorytmy mają znacznie większy rząd złożoności, niż większość ludzi jest w stanie zaakceptować, np. *n!* albo wykładniczy 2^n.

Istnieje też grupa problemów nierozwiązywalnych algorytmicznie.

Rząd złożoności, a moc komputera

Przełóżmy to teraz na praktykę. Łatwo sprawdzić, że wzrost mocy komputera powoduje znaczący przyrost szybkości jedynie dla algorytmów o niskim rzędzie złożoności, np. liniowym *n* czy logarytmicznym *n log n*. Załóżmy, że mamy dwa komputery A i B oraz drugi z nich jest dziesięciokrotnie szybszy od pierwszego. Każdemu z nich dajemy 100 sekund na rozwiązanie czterech problemów algorytmicznych o różnej złożoności i obserwujemy, z jakim rodzajem danych wejściowych poradzi sobie każdy z nich.

Algorytm	**Komputer A**	**Komputer B**	**Przyrost**
$10n$	10	100	10
$4n^2$	5	15	3
n^3	4	10	2,5
2^n	6	10	1,6

Widzimy teraz, że zwiększenie mocy komputera miało najbardziej znaczący wpływ na algorytm liniowy, podczas gdy dla najbardziej zasobożernego algorytmu wykładniczego zaobserwowaliśmy najmniejszy przyrost. Wynika z tego wniosek, że postęp technologiczny ma sens jedynie dla algorytmów o niskim rzędzie złożoności.

Analogiczne rozumowanie z całego tego rozdziału można przeprowadzić także dla *np. pamięci, dochodząc do identycznych wniosków - wtedy mamy do czynienia ze złożonością pamięciową.*

Podsumowanie

Rozdział ten ma charakter jedynie ogólnego wprowadzenia do problemu, abyś nie czuł się zagubiony, widząc używane w tym podręczniku pojęcia w stylu *element można odnaleźć w czasie liniowym* oraz miał świadomość, od czego zależy wydajność algorytmu, a tym samym tworzonych przez Ciebie programów. Po zaznajomieniu się z tym rozdziałem, powinieneś wiedzieć:

- Co to jest złożoność obliczeniowa oraz pamięciowa
- Czym jest notacja dużego O oraz dużej Ω
- Jakie są podstawowe klasy złożoności obliczeniowej
- Jak złożoność obliczeniowa przekłada się na faktyczną wydajność programu

Jeżeli jesteś dalej zainteresowany tym zagadnieniem, polecamy zajrzeć do książek wymienionych w bibliografii.

Implementacje w C++

Spis treści	
Wstęp	Wstęp - Konwencje
Struktury danych	Tablice - Listy - Stosy - Kolejki - Drzewa - Zbiory - Tablice haszujące - Grafy
Dodatki	Złożoność obliczeniowa - Implementacje w C++ - Implementacje w Pascalu - Bibliografia - Dla twórców podręcznika

Implementacje w C++

W niniejszym dodatku zamieściliśmy przykładowe implementacje w języku C++ zaprezentowanych w podręczniku struktur danych. Wykorzystują one programowanie obiektowe oraz szablony.

1. Tablice
2. Listy
3. Stosy
4. Kolejki
5. Drzewa
6. Zbiory
7. Tablice haszujące
8. Grafy

Implementacje w Pascalu

Spis treści	
Wstęp	Wstęp - Konwencje
Struktury danych	Tablice - Listy - Stosy - Kolejki - Drzewa - Zbiory - Tablice haszujące - Grafy
Dodatki	Złożoność obliczeniowa - Implementacje w C++ - Implementacje w Pascalu - Bibliografia - Dla twórców podręcznika

Implementacje w Pascalu

W niniejszym dodatku prezentujemy przykładowe implementacje w Pascalu zaprezentowanych w podręczniku struktur danych. Wykorzystaliśmy do tego celu programowanie obiektowe.

1. Tablice
2. Listy
3. Stosy
4. Kolejki
5. Drzewa
6. Zbiory
7. Tablice haszujące
8. Grafy

Bibliografia

Spis treści	
Wstęp	Wstęp - Konwencje
Struktury danych	Tablice - Listy - Stosy - Kolejki - Drzewa - Zbiory - Tablice haszujące - Grafy
Dodatki	Złożoność obliczeniowa - Implementacje w C++ - Implementacje w Pascalu - Bibliografia - Dla twórców podręcznika

Bibliografia

1. *Algorytmy i struktury danych*, Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, wyd. Helion
2. *C++. Algorytmy i struktury danych*, Adam Drozdek, wyd. Helion
3. *Sztuka programowania. Tomy 1-3*, Donald E. Knuth, Wydawnictwa Naukowo-Techniczne
4. *Wprowadzenie do algorytmów*, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Wydawnictwa Naukowo-Techniczne

Dla twórców podręcznika

Spis treści	
Wstęp	Wstęp - Konwencje
Struktury danych	Tablice - Listy - Stosy - Kolejki - Drzewa - Zbiory - Tablice haszujące - Grafy
Dodatki	Złożoność obliczeniowa - Implementacje w C++ - Implementacje w Pascalu - Bibliografia - Dla twórców podręcznika

Dla twórców podręcznika

Niniejsza strona zbiera zalecenia dla autorów i standardy, w jakich pisany będzie podręcznik. Aktualnie znajduje się on w BARDZO początkowej fazie tworzenia, dlatego możliwe są zmiany. Wszelkie nowe propozycje prosimy zgłaszać na stronie dyskusji.

Styl

Podręcznik adresowany jest przede wszystkim do osób rozpoczynających swą przygodę z algorytmami, a także pragnących uporządkować swą wiedzę na ten temat. Staramy się pisać tak, aby nie przypominał on pracy zaliczeniowej jakiegoś znużonego studenta - najważniejsza jest przystępność oraz staranność opisów. Nie mamy nic przeciwko opisom obrazowym, ponieważ jest to świetne uzupełnienie naukowych i technicznych definicji.

Przy omawianiu każdej struktury danych trzymamy się następującego schematu:

1. Wstęp ilustrujący problem, który mamy do rozwiązania (np. wady jakiejś innej struktury)
2. Krótki opis obrazowy streszczający ideę kryjącą się za daną strukturą
3. Bardziej naukowa definicja struktury danych
4. Operacje, jakie można wykonywać na danej strukturze
5. Sposoby implementacji

6. Ćwiczenia (podział na część *Podstawową* oraz *Zaawansowaną*)

Podręcznik dodatkowo zawiera dwa rozdziały zatytułowane odpowiednio: Implementacje w C++ oraz Implementacje w Pascalu pokazujące działające przykładowe implementacje w dwóch językach programowania C++ oraz Pascal wraz z przykładami użycia. Mają one charakter zapoznawczy.

Podręcznik zakłada, że czytelnik ma jakieś pojęcie nt tego, czym jest **złożoność obliczeniowa**. W razie czego, jest ona dokładniej wyjaśniona w jednym z dodatków, zatem można śmiało stosować w opisach notację dużego O oraz dużej Ω.

Przykłady

We właściwej części podręcznika do prezentacji algorytmów stosujemy pseudokod ze składnią Pascalową. Uproszczenia polegają na tym, że niektóre fragmenty zastępujemy krótkim opisem słownym - przeważnie dotyczy to miejsc, gdzie implementacja tego, co napisaliśmy, okropnie zagmatwałaby kod. Ponadto korzystamy z operacji zdefiniowanych na początku rozdziału (np. *INSERT*) jak funkcji, również, aby uprościć kod. Przyjmujemy, że nazwy operacji zapisane są dużymi literami. Opis słowny umieszczamy między znakami mniejszości i większości. Oto przykład:

```
procedure ABC(V: Data);
var S: Set;
    P: Data;

begin
   for <każdy element P w zbiorze S> do
   begin
      write(P);
   end;
end;
```

Stosujemy nazewnictwo:

1. Nazwy operacji wykonywanych na strukturze danych - dużymi literami, np. *INSERT*, *DELETE*
2. Angielskie nazwy struktur danych jako typy, np. *Set*, *List*
3. Typ *Data* do reprezentowania jakichś abstrakcyjnych danych umieszczanych w strukturze.
4. Reszta małymi literami.

Przyjmujemy, że funkcja *write* potrafi wyświetlić wszystko.

Jeśli chodzi o dział *Implementacje*, stosujemy następujące reguły:

C++

1. Nawiasy klamrowe otwieramy w nowej linijce
2. Wcięcia trójznakowe
3. W nazewnictwie posługujemy się camelStyle (tj. zmienne, funkcje itd. nazywamy jako *nazwaFunkcji*, a nie *nazwa_funkcji* albo *nazwafunkcji*).
4. Poszczególne części algorytmu staramy się separować linijką przerwy
5. Kod musi być skomentowany, najlepiej komentarzami jednolinijkowymi
6. Implementacja musi być zapisana z użyciem programowania obiektowego oraz szablonów
7. W implementacji **NIE** korzystamy ze struktur danych dostępnych w STL - w końcu po to jest ten dział, by czytelnik sam nauczył się dane struktury pisać.

Pascal

1. Wcięcia trójznakowe
2. "begin" w nowej linijce
3. W nazewnictwie posługujemy się camelStyle (tj. zmienne, funkcje itd. nazywamy jako *nazwaFunkcji*, a nie *nazwa_funkcji* albo *nazwafunkcji*).
4. Poszczególne części algorytmu staramy się separować linijką przerwy.

Formatowanie

W opisie stosujemy następującą konwencję:

1. *kursywa* - nazwy zmiennych, funkcji przy nawiązaniach do kodu algorytmu
2. **pogrubienie** - nazwy struktur kontrolnych, np. **for**, **if** oraz predefiniowanych wartości, np. **null**
3. Akapitów używamy zgodnie z przeznaczeniem, tj. do zmiany wątku, a nie do separacji zdań, jak czasem się niektórym zdarza robić.
4. Kolorujemy składnię! Szczegóły: Pomoc:Podświetlanie_składni

Szablony

Oto wykaz szablonów używanych w podręczniku:

Opis	Kod	Efekt
Ostrzeżenie czytelnika	`{{Uwaga|Tekst ostrzeżenia}}`	**Uwaga!** Tekst ostrzeżenia
Porada	`{{Porada|Tekst porady}}`	**Porada** Tekst porady
Informacja	`{{Infobox|Tekst informacji}}`	Tekst informacji
Definicja	`{{Definicja|Tekst definicji}}`	Tekst definicji
Do zrobienia	`{{TODO|co zrobić}}`	**Do zrobienia:** co zrobić
Do zrobienia • do wstawiania w sekcji • stosować tylko w rozdziałach, w których większość tekstu jest napisana	`{{RDoZrobienia}}`	W *Ta sekcja jest zalążkiem. Jeśli możesz, rozbuduj ją* [1].
Artykuł do poprawy	`{{dopracować|powód}}`	

Nawigacja

Wykaz szablonów nawigacyjnych:

<table>
<tr><th>Opis</th><th>Kod</th><th>Efekt</th></tr>
<tr><td>Główna nawigacja</td><td>{{Struktury danych/SpisTreści}}</td><td>
<table>
<tr><td>Spis treści</td><td></td></tr>
<tr><td>Wstęp</td><td>Wstęp - Konwencje</td></tr>
<tr><td>Struktury danych</td><td>Tablice - Listy - Stosy - Kolejki - Drzewa - Zbiory - Tablice haszujące - Grafy</td></tr>
<tr><td>Dodatki</td><td>Złożoność obliczeniowa - Implementacje w C++ - Implementacje w Pascalu - Bibliografia - Dla twórców podręcznika</td></tr>
</table>
</td></tr>
<tr><td>Menu implementacji: C++</td><td>{{Struktury danych/ImplC++}}</td><td>
<table>
<tr><td>Implementacje w C++</td><td>Tablice - Listy - Stosy - Kolejki - Drzewa - Zbiory - Tablice haszujące - Grafy</td></tr>
</table>
</td></tr>
<tr><td>Menu implementacji: Pascal</td><td>{{Struktury danych/ImplPascal}}</td><td>
<table>
<tr><td>Implementacje w Pascalu</td><td>Tablice - Listy - Stosy - Kolejki - Drzewa - Zbiory - Tablice haszujące - Grafy</td></tr>
</table>
</td></tr>
</table>

Przypisy

[1] http://pl.wikibooks.org/w/index.php?title=Struktury_danych/Dla_tw%C3%B3rc%C3%B3w_podr%C4%99cznika&action=edit

Źródła i autorzy artykułu

Struktury danych *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=194230 *Autorzy*: Derbeth, Karol Karolus, Kazet, Kompowiec2, Zyx

Wstęp *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=136435 *Autorzy*: Lethern, Zyx, 1 anonimowych edycji

Konwencje *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=91498 *Autorzy*: Lethern, Zyx

Tablice *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=91499 *Autorzy*: Lethern, Webprog, Zyx, 1 anonimowych edycji

Listy *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=51549 *Autorzy*: Derbeth, Piotr, Zyx, 3 anonimowych edycji

Stosy *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=188136 *Autorzy*: Hyper, 15 anonimowych edycji

Kolejki *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=155793 *Autorzy*: 1 anonimowych edycji

Drzewa *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=85197 *Autorzy*: Kazet, 1 anonimowych edycji

Złożoność obliczeniowa *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=128243 *Autorzy*: Lethern, Vatzec, Zyx, 6 anonimowych edycji

Implementacje w C++ *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=91496 *Autorzy*: Lethern, Zyx

Implementacje w Pascalu *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=91497 *Autorzy*: Lethern, Zyx

Bibliografia *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=91494 *Autorzy*: Lethern, Zyx

Dla twórców podręcznika *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=154152 *Autorzy*: Karol Dąbrowski, Lethern, Zyx

Źródła, licencje i autorzy grafik

Grafika:Strukturydanych-okladka.jpg *Źródło*: http://pl.wikibooks.org/w/index.php?title=Plik:Strukturydanych-okladka.jpg *Licencja*: GNU Free Documentation License *Autorzy*: Zyx

Grafika:Fairytale messagebox info.png *Źródło*: http://pl.wikibooks.org/w/index.php?title=Plik:Fairytale_messagebox_info.png *Licencja*: GNU Lesser General Public License *Autorzy*: Amada44, Anime Addict AA, Bayo, Dake, Jon Harald Søby, Mapmarks, Rocket000, ZooFari

Grafika:Plume ombre.png *Źródło*: http://pl.wikibooks.org/w/index.php?title=Plik:Plume_ombre.png *Licencja*: GNU Free Documentation License *Autorzy*: Darkdadaah, Javierme, Mindmatrix, Rilegator, Rocket000

Grafika:Nuvola apps important.svg *Źródło*: http://pl.wikibooks.org/w/index.php?title=Plik:Nuvola_apps_important.svg *Licencja*: GNU Lesser General Public License *Autorzy*: Bastique

Grafika:Lista-jednokierunkowa.gif *Źródło*: http://pl.wikibooks.org/w/index.php?title=Plik:Lista-jednokierunkowa.gif *Licencja*: Public Domain *Autorzy*: Zyx

Grafika:Evolution-tasks.png *Źródło*: http://pl.wikibooks.org/w/index.php?title=Plik:Evolution-tasks.png *Licencja*: GNU General Public License *Autorzy*: Artwork by Tuomas Kuosmanen <tigert_at_ximian.com> and Jakub Steiner <jimmac_at_ximian.com>

Grafika:Stack_data_structure.gif *Źródło*: http://pl.wikibooks.org/w/index.php?title=Plik:Stack_data_structure.gif *Licencja*: GNU Free Documentation License *Autorzy*: Hyperthermia

Grafika:Strukturydanych-zo-wykresczasu.svg *Źródło*: http://pl.wikibooks.org/w/index.php?title=Plik:Strukturydanych-zo-wykresczasu.svg *Licencja*: Public Domain *Autorzy*: Zyx

Image:Wiki letter w.svg *Źródło*: http://pl.wikibooks.org/w/index.php?title=Plik:Wiki_letter_w.svg *Licencja*: GNU Free Documentation License *Autorzy*: Jarkko Piiroinen

Licencja

Creative Commons Attribution-Share Alike 3.0
//creativecommons.org/licenses/by-sa/3.0/