

Materiały do przedmiotu:

Algorytmy i struktury danych

Laboratorium nr 1

**Analiza poprawności i złożoności
obliczeniowej algorytmów**

Autorzy:

dr hab. Adam Kiersztyn
mgr Krystyna Kiersztyn



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Spis treści

Wprowadzenie, tematyka zajęć	3
Cel ćwiczenia/laboratorium	4
Instrukcja laboratoryjna/ćwiczeniowa	4
Ćwiczenia samodzielne	9
Pytania pomocnicze	12
Podsumowanie	12
Literatura	12



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wprowadzenie, tematyka zajęć

Algorytm jest formalnym opisem sposobu rozwiązania określonego problemu obliczeniowego. Stanowi on jednoznacznie określony, skończony ciąg kroków (instrukcji), które dla zadanych danych wejściowych prowadzą do uzyskania wyniku zgodnego ze specyfikacją problemu.

W praktyce projektowania oprogramowania nie wystarcza samo skonstruowanie algorytmu rozwiązującego dany problem. Konieczna jest również jego analiza, pozwalająca odpowiedzieć na pytania, czy algorytm działa poprawnie dla wszystkich dopuszczalnych danych wejściowych, czy zawsze kończy swoje działanie oraz jaki jest koszt jego wykonania. Brak takiej analizy może prowadzić do błędów logicznych, obniżonej wydajności, a w skrajnych przypadkach do podatności systemów na różnego rodzaju awarie.

Jednym z podstawowych aspektów analizy algorytmów jest **poprawność algorytmu**, rozumiana jako zgodność jego działania z przyjętą specyfikacją. W tym kontekście wyróżnia się **poprawność częściową**, która oznacza, że algorytm — o ile zakończy działanie — zwraca poprawny wynik, oraz **poprawność całkowitą**, która dodatkowo wymaga, aby algorytm zawsze kończył działanie dla wszystkich dopuszczalnych danych wejściowych.

W przypadku algorytmów iteracyjnych istotnym narzędziem formalnej analizy poprawności jest **niezmiennik (inwariant) pętli**. Jest to własność spełniona przed rozpoczęciem pętli, zachowywana w każdej jej iteracji oraz prawdziwa po zakończeniu działania pętli. Niezmiennik pętli umożliwia formalne uzasadnienie poprawności algorytmu.

Innym kluczowym aspektem analizy algorytmów jest **złożoność obliczeniowa**, rozumiana jako miara kosztu wykonania algorytmu w zależności od rozmiaru danych wejściowych. W ramach tego laboratorium szczególny nacisk położony jest na analizę maksymalnej **złożoności czasowej**, opisującej najgorszy możliwy przypadek działania algorytmu. Uzupełnieniem analizy czasowej jest analiza **złożoności pamięciowej**, określająca ilość dodatkowej pamięci wykorzystywanej przez algorytm.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Cel ćwiczenia/laboratorium

Po zakończeniu laboratorium student:

- rozumie pojęcie algorytmu,
- rozumie znaczenie analizy algorytmów,
- rozróżnia poprawność częściową i poprawność całkowitą algorytmu,
- rozumie pojęcie niezmiennika pętli oraz jego rolę w analizie poprawności algorytmów iteracyjnych,
- potrafi wyznaczyć maksymalną złożoność czasową prostych algorytmów iteracyjnych,
- potrafi oszacować złożoność pamięciową algorytmu.

Instrukcja laboratoryjna/ćwiczeniowa

Przykład 1.1. Analiza poprawności oraz złożoności algorytmu

Dane wejściowe: Liczba naturalna $n \geq 0$.

Wynik: Suma pierwszych n liczb naturalnych: $S(n) = 1 + 2 + \dots + n$.

Dla $n = 0$ przyjmujemy $S(0) = 0$.

Algorytm P.1.1.

```
sum = 0
for i = 1 to n do
    sum = sum + i
return sum
```

Algorytm inicjalizuje zmienną `sum` wartością 0, a następnie w kolejnych iteracjach pętli dodaje do niej kolejne liczby naturalne od 1 do n . Po zakończeniu pętli zmienna `sum` zawiera sumę wszystkich wymaganych składników.

Pętla `for` posiada skończoną liczbę iteracji wynoszącą n , więc algorytm zawsze kończy działanie.

Dowód poprawności

Poprawność algorytmu wykażemy za pomocą niezmiennika pętli.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Niezmiennik pętli

Po zakończeniu k -tej iteracji pętli (dla $k = 0, 1, \dots, n$) zmienna sum przechowuje wartość

$$sum = 1 + 2 + \dots + k$$

1. Inicjalizacja

Przed pierwszą iteracją pętli zmienna sum ma wartość 0. Dla $k = 0$ suma pustego zbioru wynosi 0, więc niezmiennik jest spełniony.

2. Utrzymanie

Zakładamy, że po k -tej iteracji:

$$sum = 1 + 2 + \dots + k$$

W następnej iteracji do zmiennej sum dodawana jest wartość $(k + 1)$, zatem:

$$sum = (1 + 2 + \dots + k) + (k + 1)$$

Niezmiennik pozostaje spełniony.

3. Zakończenie

Pętla wykonuje się dokładnie n razy. Po jej zakończeniu:

$$sum = 1 + 2 + \dots + n$$

Algorytm zwraca poprawną wartość $S(n)$.

Pętla for wykonuje się skończoną liczbę razy (dokładnie n), zatem algorytm zawsze kończy działanie dla każdego $n \geq 0$.

Złożoność czasowa

W każdej iteracji wykonywana jest stała liczba operacji (dodawanie i przypisanie). Pętla wykonuje się n razy, dlatego całkowity czas działania jest proporcjonalny do n . Zatem złożoność czasowa wynosi $O(n)$.

Złożoność pamięciowa

Algorytm wykorzystuje stałą liczbę zmiennych (sum , i). Nie tworzy dodatkowych struktur danych zależnych od n . Zatem złożoność pamięciowa wynosi $O(1)$.

Czy istnieje sposób obliczenia sumy pierwszych n liczb naturalnych bez użycia pętli – w czasie stałym, tj. $O(1)$?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykład 1.2. Analiza złożoności algorytmu

Dane wejściowe: liczba naturalna $n \geq 1$.

Algorytm P.1.2.

```
i = 1
while i < n do
    i = i + 3
```

Algorytm inicjalizuje zmienną i wartością 1, a następnie w każdej iteracji zwiększa ją o 3, dopóki spełniony jest warunek $i < n$.

Zakończenie działania

W każdej iteracji wartość zmiennej i rośnie o stałą dodatnią wartość [3]. Ciąg wartości przyjmowanych przez i ma postać:

1, 4, 7, 10, ...

Jest to ciąg arytmetyczny o różnicy 3.

Ponieważ i rośnie monotonicznie oraz jest ograniczone z góry przez n , po skończonej liczbie iteracji osiągnie wartość $i \geq n$. Zatem pętla zawsze kończy działanie.

Liczba iteracji

Liczbę iteracji wyznaczymy, znajdując najmniejsze k , dla którego pętla się zatrzyma, tzn. $i \geq n$. Po k iteracjach zmienna i ma wartość:

$$i = 1 + 3k$$

Warunek zakończenia pętli spełniony jest, gdy:

$$1 + 3k \geq n$$

Stąd:

$$3k \geq n - 1$$

$$k \geq \frac{n - 1}{3}$$

Zatem liczba iteracji wynosi:

$$\left\lceil \frac{n - 1}{3} \right\rceil$$



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Złożoność czasowa

Liczba iteracji wynosi $\left\lceil \frac{n-1}{3} \right\rceil$, a więc jest proporcjonalna do n . Zatem złożoność czasowa algorytmu jest liniowa i wynosi $O(n)$.

Złożoność pamięciowa

Algorytm wykorzystuje jedną zmienną (i), zatem złożoność pamięciowa jest stała i wynosi $O(1)$.

Przykład 1.3. Analiza algorytmu z pętlą logarytmiczną

Dane wejściowe: liczba naturalna $n \geq 1$.

Algorytm P.1.3.

```
i = n
while i > 1 do
    i = i / 2
```

Algorytm inicjalizuje zmienną i wartością n , a następnie w każdej iteracji dzieli ją przez 2, dopóki spełniony jest warunek $i > 1$.

Zakończenie działania

W każdej iteracji wartość i zmniejsza się dwukrotnie. Ciąg wartości przyjmowanych przez i ma postać:

$$n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \dots$$

Łatwo zauważyć, że jest to ciąg geometryczny o ilorazie $\frac{1}{2}$.

Ponieważ i maleje monotonicznie oraz jest ograniczone z dołu przez 1, po skończonej liczbie iteracji osiągnie wartość $i \leq 1$. Zatem pętla zawsze kończy działanie.

Liczba iteracji

Po k iteracjach zmienna i ma wartość:

$$i = \frac{n}{2^k}$$

Pętla kończy się, gdy:

$$\frac{n}{2^k} \leq 1$$

Stąd:

$$n \leq 2^k$$



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Szukamy najmniejszej liczby całkowitej k spełniającej warunek:

$$k \geq \log_2 n.$$

Zatem liczba iteracji wynosi:

$$\lceil \log_2 n \rceil$$

Złożoność czasowa

Liczba iteracji rośnie proporcjonalnie do $\log_2 n$, dlatego złożoność czasowa wynosi $O(\log n)$.

Złożoność pamięciowa

Algorytm wykorzystuje jedną zmienną (i), zatem złożoność pamięciowa jest stała i wynosi $O(1)$.

Przykład 1.4. Analiza algorytmu z pętlą zagnieżdżoną

Dane wejściowe: liczba całkowita $n \geq 0$.

Algorytm P.1.4.

```
for i = 1 to n do
  for j = 1 to 5 do
    x = x + 1
```

Pętla zewnętrzna wykonuje się n razy. Pętla wewnętrzna wykonuje się stałą liczbę razy (5), niezależnie od n .

Liczba operacji

W każdej iteracji pętli zewnętrznej instrukcja $x = x + 1$ wykonywana jest 5 razy. Zatem całkowita liczba wykonań tej instrukcji wynosi $5n$.

Złożoność czasowa

Stały czynnik 5 nie wpływa na rząd złożoności, dlatego złożoność czasowa wynosi $O(n)$.

Złożoność pamięciowa

Algorytm wykorzystuje stałą liczbę zmiennych (i, j, x), więc złożoność pamięciowa jest stała i wynosi $O(1)$.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykład 1.5. Analiza złożoności algorytmu z dodatkową tablicą

Dane wejściowe: tablica liczb $A[1..n]$, gdzie $n \geq 1$.

Algorytm P.1.5.

```
S[1] = A[1]
for i = 2 to n do
    S[i] = S[i-1] + A[i]
return S
```

Algorytm tworzy nową tablicę S , w której i -ty element jest sumą pierwszych i elementów tablicy wejściowej A .

Zakończenie działania

Pętla for wykonuje się dokładnie $n - 1$ razy, zatem algorytm kończy działanie dla każdego $n \geq 1$.

Złożoność czasowa

W każdej iteracji wykonywana jest stała liczba operacji (dodawanie i przypisanie). Zatem złożoność czasowa wynosi $O(n)$.

Złożoność pamięciowa

Algorytm oprócz danych wejściowych alokuje w pamięci dodatkową tablicę S długości n . Zatem dodatkowe zużycie pamięci rośnie liniowo wraz z n . Złożoność pamięciowa wynosi $O(n)$.

Ćwiczenia samodzielne

Zadanie 1.1.

Dla każdej procedury określ, czy jest algorytmem. Odpowiedź uzasadnij.

- Powtarzaj losowanie liczby, aż wylosowana wartość będzie równa 6.
- Pobieraj dane, aż użytkownik poda wystarczająco dobre hasło.
- Wypisuj kolejne liczby naturalne od 1 do 10.
- Pobieraj hasło od użytkownika, aż poda hasło składające się z co najmniej 8 znaków i zawierające przynajmniej jedną cyfrę.
- Przeskanuj plik i znajdź podejrzane wpisy.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 1.2.

Dany jest algorytm:

Algorytm Z.1.1.

```
i = 0
while i <= n do
  print(i)
```

Czy algorytm jest poprawny dla wszystkich wartości n ? Jakie warunki wejściowe muszą być spełnione, aby algorytm zakończył działanie? Zaproponuj poprawioną wersję algorytmu, jeżeli to konieczne.

Zadanie 1.3.

Rozważ algorytm wyznaczania maksimum w tablicy:

Algorytm Z.1.2.

Dane: tablica $A[1..n]$, $n \geq 1$
Wynik: największy element tablicy

```
max = A[1]
for i = 2 to n do
  if A[i] > max then
    max = A[i]
return max
```

Sformułuj niezmiennik pętli. Udowodnij poprawność algorytmu, korzystając z niezmiennika pętli. Określ, czy algorytm jest poprawny częściowo czy całkowicie. Odpowiedź uzasadnij.

Zadanie 1.4.

Dany jest algorytm:

Algorytm Z.1.3.

```
for i = 1 to n do
  for j = 1 to n do
    x = x + 1
```

Wyznacz dokładną liczbę wykonania instrukcji $x = x + 1$. Podaj złożoność czasową algorytmu w notacji O .

Zadanie 1.5.

Dany algorytm wyszukiwania elementu k w tablicy A :



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Algorytm Z.1.4.

```
for i = 1 to n do
  if A[i] = k then
    return i
return -1
```

Wyznacz złożoność czasową w:

- a) przypadku optymistycznym,
- b) przypadku pesymistycznym.

Zadanie 1.6.

Rozważ algorytm:

Algorytm Z.1.5.

```
i = n
while i != 0 do
  i = i - 2
```

Dla jakich wartości n algorytm zakończy działanie? Zaproponuj poprawioną wersję, jeżeli to konieczne.

Zadanie 1.7.

Dany jest algorytm:

Algorytm Z.1.6.

```
for i = 1 to n do
  x = x + 1
```

Ile razy wykona się instrukcja $x = x + 1$? Jaka jest maksymalna liczba operacji? Podaj złożoność czasową algorytmu w notacji O .

Zadanie 1.8.

Dany jest algorytm:

Algorytm Z.1.7.

```
i = 1
while i <= n do
  i = 2 * i
```

Ile razy wykona się pętla? Podaj złożoność czasową algorytmu w notacji O .



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Pytania pomocnicze

- 1) Jakie są kluczowe cechy algorytmu?
- 2) Podaj przykłady algorytmów spotykanych w urządzeniach codziennego użytku.
- 3) Co oznacza poprawność algorytmu? Czym różni się poprawność częściowa od poprawności całkowitej?
- 4) Jakie elementy należy określić przed rozpoczęciem analizy poprawności algorytmu?
- 5) Czym jest niezmiennik pętli i jaką rolę pełni w analizie poprawności algorytmu?
- 6) Co rozumiemy przez maksymalną złożoność czasową algorytmu?
- 7) Co oznacza pojęcie „operacja dominująca” w algorytmie?

Podsumowanie

W trakcie laboratorium omówiono podstawowe zagadnienia związane z analizą algorytmów, ze szczególnym uwzględnieniem poprawności działania oraz złożoności obliczeniowej. Celem laboratorium było wykształcenie umiejętności formalnego oceniania algorytmów, niezależnie od języka programowania.

Przeprowadzona analiza pokazała, że przed oceną algorytmu konieczne jest jednoznaczne określenie jego specyfikacji, w tym danych wejściowych, oczekiwanego wyniku oraz warunków początkowych. Pozwala to na rzetelną ocenę poprawności algorytmu.

W ramach laboratorium zwrócono uwagę na rolę niezmiennika pętli jako narzędzia wspomagającego dowodzenie poprawności algorytmów iteracyjnych. Istotnym elementem laboratorium była analiza złożoności czasowej.

Literatura

- [1] Drozdek A., *C++. Algorytmy i struktury danych*, Helion, Gliwice 2004. s. 61-71.
- [2] Aho A., Hopcroft J., Ullman J., *Algorytmy i struktury danych*, Helion, Gliwice, 2003. s. 28-39, 305.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Materiały do przedmiotu:
Algorytmy i struktury danych

Laboratorium nr 2

Porównanie metod projektowania algorytmów

Autorzy:
dr hab. Adam Kiersztyn
mgr Krystyna Kiersztyn



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Spis treści

Wprowadzenie, tematyka zajęć -----	15
Cel ćwiczenia/laboratorium -----	16
Instrukcja laboratoryjna/ćwiczeniowa -----	17
Ćwiczenia samodzielne -----	26
Pytania pomocnicze -----	27
Podsumowanie -----	28
Literatura -----	28



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU
SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Problemy informatyczne mogą być rozwiązywane na wiele różnych sposobów, a wybór najlepszego z nich nie zawsze jest oczywisty. Aby uzyskać poprawny i efektywny algorytm, konieczne jest przeanalizowanie problemu oraz zaplanowanie kolejnych etapów postępowania. Proces ten nazywany jest **projektowaniem algorytmów** i polega na doborze oraz uporządkowaniu działań prowadzących od danych wejściowych do oczekiwanego wyniku.

W praktyce wyróżnia się dwa podstawowe podejścia do projektowania algorytmów: **metodę zstępującą (top-down)** oraz **metodę wstępującą (bottom-up)**. Niezależnie od zastosowanej metody projektowania, proces tworzenia rozwiązania powinien być realizowany zgodnie z określonym schematem, obejmującym analizę problemu i wymagań, projektowanie, implementację oraz kolejne etapy testowania i utrzymania systemu.

Metoda zstępująca polega na rozpoczęciu pracy od ogólnego problemu i stopniowym dzieleniu go na coraz mniejsze podproblemy, aż do momentu osiągnięcia elementarnych operacji, których nie można już bardziej uprościć. Następnie rozwiązania podproblemów są łączone w całość, prowadząc do rozwiązania problemu głównego. Podejście to sprzyja systematyzacji pracy, przejrzystości kodu oraz modularności, a także umożliwia ponowne wykorzystanie opracowanych rozwiązań. Jednocześnie wymaga ono starannego planowania, ponieważ błędy popełnione na wyższych poziomach rozważań mogą wpłynąć na działanie całego algorytmu. Charakterystycznym przykładem podejścia zstępującego jest **rekurencja**, w której problem rozwiązywany jest poprzez odwoływanie się do jego mniejszych wersji.

Z kolei metoda wstępująca zakłada budowanie rozwiązania od podstaw, poprzez tworzenie prostych, niezależnych elementów, a następnie łączenie ich w coraz bardziej złożone struktury. W trakcie tego procesu tworzony jest zestaw modułów i funkcji, które stopniowo rozszerzają możliwości programu. Rozwiązanie powstaje więc poprzez składanie niezależnych komponentów, co sprzyja testowaniu oraz łatwej rozbudowie systemu. Wadą tego podejścia może być jednak początkowy brak spójnej wizji końcowego rozwiązania, zwłaszcza gdy problem nie został wcześniej dokładnie przeanalizowany.

Oprócz ogólnych metod projektowania algorytmów, takich jak podejście zstępujące i wstępujące, rozróżniamy konkretne strategie algorytmiczne, określające sposób rozwiązywania danego typu problemów. Do najczęściej wykorzystywanych należą:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



„dziel i zwyciężaj”, programowanie dynamiczne, algorytmy zachłanne oraz algorytmy siłowe.

Strategia **dziel i zwyciężaj** (ang. *divide and conquer*) polega na podziale problemu na mniejsze, niezależne podproblemy, rozwiązaniu ich osobno, a następnie połączeniu otrzymanych wyników w rozwiązanie końcowe. Podejście to jest naturalnie związane z metodą zstępującą, ponieważ proces rozwiązywania rozpoczyna się od problemu ogólnego i stopniowo przechodzi do coraz prostszych przypadków.

Programowanie dynamiczne (ang. *dynamic programming*) stosowane jest w sytuacjach, gdy problem można rozbić na nachodzące na siebie podproblemy. Kluczową ideą jest zapamiętywanie wyników wcześniej rozwiązanych podproblemów w celu uniknięcia wielokrotnych obliczeń. Programowanie dynamiczne może być realizowane zarówno w podejściu zstępującym (z wykorzystaniem memoizacji), jak i w podejściu wstępującym, polegającym na budowaniu rozwiązania od najprostszych przypadków.

Algorytmy zachłanne (ang. *greedy algorithms*) opierają się na podejmowaniu lokalnie najlepszych decyzji na każdym etapie działania algorytmu, bez analizowania wszystkich możliwych rozwiązań. Takie podejście często prowadzi do szybkich i prostych implementacji, jednak nie zawsze gwarantuje uzyskanie rozwiązania optymalnego. Skuteczność algorytmu zachłannego zależy od struktury konkretnego problemu.

Algorytmy siłowe (ang. *brute force*) to najprostsza metoda rozwiązywania problemów informatycznych, polegająca na bezpośrednim sprawdzeniu wszystkich możliwych kombinacji lub rozwiązań, aż do znalezienia poprawnego. Jest gwarancją znalezienia wyniku, ale cechuje się bardzo niską wydajnością, szczególnie dla dużych zbiorów danych, ze względu na wysoką złożoność czasową.

Cel ćwiczenia/laboratorium

Po zakończeniu laboratorium student:

- zna podstawowe metody projektowania algorytmów,
- rozumie różnice pomiędzy podejściem top-down a bottom-up,
- rozróżnia podstawowe strategie algorytmiczne: „dziel i zwyciężaj”, programowanie dynamiczne, algorytmy zachłanne oraz algorytmy siłowe,
- potrafi dobrać odpowiednią strategię algorytmiczną do charakteru rozwiązywanego problemu,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- implementuje wybrane algorytmy z wykorzystaniem różnych metod projektowania,
- analizuje i porównuje złożoność czasową oraz pamięciową zaprojektowanych rozwiązań.

Instrukcja laboratoryjna/ćwiczeniowa

Przykład 2.1. Strategia „dziel i zwyciężaj” – metoda bisekcji

Rozważmy problem wyznaczenia przybliżonego pierwiastka równania $f(x) = 0$ w przedziale $[a, b]$, przy założeniu, że funkcja f jest ciągła i ściśle monotoniczna na tym przedziale oraz że wartości funkcji na jego końcach mają różne znaki, tzn.

$$f(a) \cdot f(b) < 0$$

Zgodnie z twierdzeniem Darboux, w takim przypadku w przedziale $[a, b]$ istnieje dokładnie jeden pierwiastek równania $f(x) = 0$.

Metoda bisekcji korzysta ze strategii „dziel i zwyciężaj”. Zamiast próbować znaleźć pierwiastek bezpośrednio, algorytm:

- 1) dzieli aktualny przedział na dwie równe części,
- 2) wybiera tę część, w której na podstawie zmiany znaku funkcji musi znajdować się pierwiastek,
- 3) odrzuca drugą połowę,
- 4) powtarza ten proces aż do uzyskania wymaganej dokładności.

Algorytm P.2.1.

```
Bisekcja(a, b, eps)
{
    if f(a) * f(b) > 0 then
        return brak rozwiązania
    while |b - a| > eps do
        x = (a + b) / 2

        if |f(x)| < eps then
            return x

        if f(a) * f(x) < 0 then
            b = x
        else
            a = x
        return (a + b) / 2
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



W każdej iteracji długość przedziału zostaje zmniejszona o połowę. Po k krokach długość przedziału wynosi $\frac{b-a}{2^k}$, warunek zakończenia to $\frac{b-a}{2^k} \leq \varepsilon$. Stąd liczba iteracji jest równa $k \approx \log_2 \frac{b-a}{\varepsilon}$. Zatem złożoność czasowa algorytmu wynosi $O\left(\log \frac{1}{\varepsilon}\right)$. Złożoność pamięciowa jest rzędu $O(1)$, ponieważ algorytm przechowuje aktualne krańce przedziału i środek, niezależnie od liczby iteracji. Metoda jest więc bardzo stabilna i efektywna obliczeniowo.

Aby zilustrować działanie metody bisekcji, rozważmy konkretną funkcję

$$f(x) = x^3 - x - 2.$$

Funkcja ta jest wielomianem, a więc jest ciągła na całym $\mathbb{R}$. Sprawdźmy wartości funkcji na końcach przedziału $[1, 2]$:

$$f(1) = -2 < 0, f(2) = 4 > 0.$$

Ponieważ zachodzi zmiana znaku funkcji f w przedziale $[1, 2]$ istnieje co najmniej jedno miejsce zerowe. Możemy zatem zastosować metodę bisekcji w celu wyznaczenia jego przybliżonej wartości. Przyjmijmy dokładność $\varepsilon = 0,001$. Kolejne iteracje przedstawiono w poniższej tabeli.

Iteracja	a	b	$\frac{a+b}{2}$	$b-a$
1	1,00	2,00	1,50	1,00
2	1,50	2,00	1,75	0,50
3	1,50	1,75	1,625	0,25
4	1,50	1,625	1,5625	0,125
5	1,50	1,5625000	1,5312500	0,0625
6	1,50	1,5312500	1,5156250	0,03125
7	1,5156250	1,5312500	1,5234375	0,015625
8	1,5156250	1,5234375	1,5195313	0,0078125
9	1,5195313	1,5234375	1,5214844	0,0039062
10	1,5195313	1,5214844	1,5205078	0,0019531
11	1,5195313	1,5205079	1,520019575	0,0009766

Po iteracji nr 11 długość przedziału wynosi $0,0009766 < \varepsilon$. Oznacza to, że warunek zakończenia algorytmu został spełniony, a środek ostatniego przedziału stanowi przybliżenie pierwiastka z zadaną dokładnością $\varepsilon = 0,001$.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykład 2.2. Programowanie dynamiczne w podejściach top-down i bottom-up

Ciąg Fibonacciego jest zdefiniowany rekurencyjnie za pomocą następującego wzoru:

$$F(n) = \begin{cases} n & \text{dla } n \leq 1 \\ F(n-2) + F(n-1) & \text{dla } n > 1 \end{cases}$$

Zależność ta oznacza, że aby obliczyć wartość $F(n)$, należy znać dwie poprzednie wartości ciągu. Jest to klasyczny przykład problemu z nakładającymi się podproblemami, co czyni go naturalnym kandydatem do zastosowania programowania dynamicznego.

W niniejszym przykładzie przedstawiono trzy metody obliczania wartości $F(n)$:

- rekurencja naiwna,
- programowanie dynamiczne w podejściu top-down (memoizacja),
- programowanie dynamiczne w podejściu bottom-up

I. Rekurencja naiwna

Rekurencyjna implementacja wynikająca bezpośrednio z definicji ciągu Fibonacciego nie wykorzystuje mechanizmu zapamiętywania wcześniej obliczonych wartości. W konsekwencji te same podproblemy są rozwiązywane wielokrotnie, co prowadzi do gwałtownego wzrostu liczby wywołań funkcji wraz ze wzrostem parametru n . Zagadnienie to zostanie szerzej omówione podczas laboratorium poświęconego rekurencji.

Aby obliczyć wartość $F(n)$, algorytm wywołuje funkcje obliczające $F(n-1)$ oraz $F(n-2)$. Z kolei obliczenie $F(n-1)$ wymaga wyznaczenia $F(n-2)$ oraz $F(n-3)$, itd. Proces ten powtarza się aż do osiągnięcia przypadków bazowych, tj. $F(1)$ oraz $F(0)$.

Algorytm P2.2.A

```
Fibonacci(n)
{
    if n == 0 or n == 1 then
        return n
    return Fibonacci(n-2) + Fibonacci(n-1)
}
```

Rysunek 1 przedstawia drzewo wywołań funkcji Fibonacci dla parametru $n = 5$. Każdy węzeł drzewa odpowiada jednemu wywołaniu funkcji. Już dla niewielkich wartości n widoczne jest wielokrotne pojawianie się tych samych podproblemów, takich jak $F(3)$ czy $F(2)$.



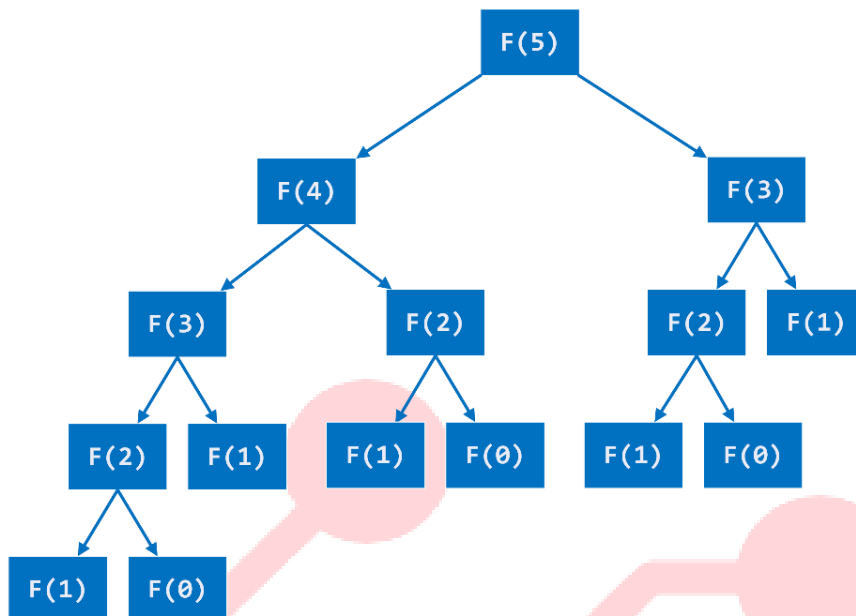
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską





Rysunek 1 Drzewo podproblemów dla $\text{Fibonacci}(n)$ przy $n = 5$ (źródło: opracowanie własne)

Niewątpliwie wadą tego podejścia jest to, że wymaga ono wielokrotnego obliczania tych samych liczb Fibonacciego, w celu uzyskania rozwiązania końcowego. Złożoność czasowa tej metody ma charakter wykładniczy $O(2^n)$. Natomiast złożoność pamięciowa wynika z głębokości stosu wywołań i wynosi $O(n)$.

II. Podejście top-down

W celu wyeliminowania powtarzających się obliczeń można zastosować *programowanie dynamiczne w podejściu top-down*. Idea polega na zapamiętywaniu wyników już rozwiązanych podproblemów i ponownym ich wykorzystaniu (w razie potrzeby).

W implementacji wykorzystywana jest tablica pomocnicza do przechowywania wcześniej obliczonych wartości. Tablica ta tworzona jest w funkcji pomocniczej `FibonacciHelper`, a następnie przekazywana do funkcji rekurencyjnej `TopDownFibonacci`.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



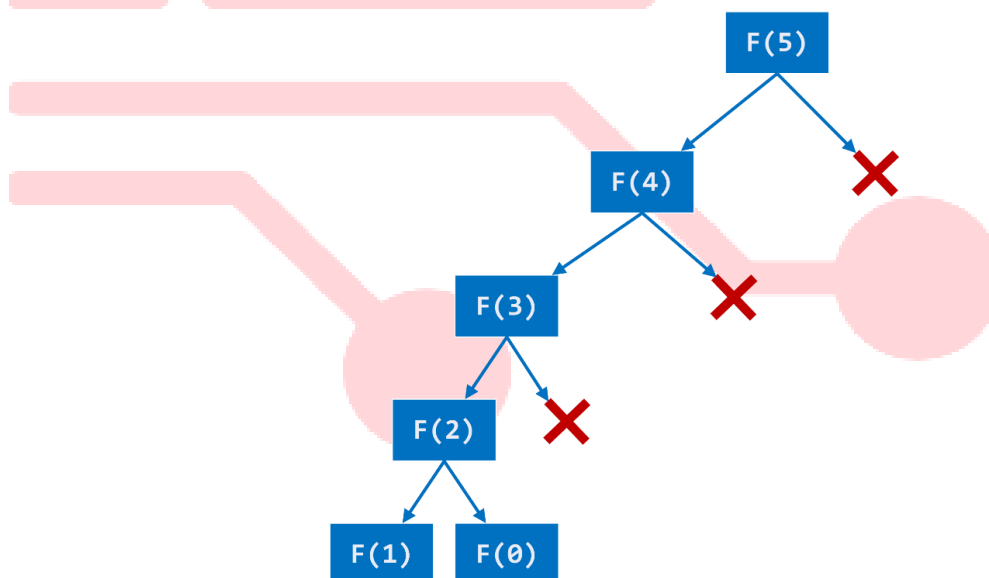
Algorytm P.2.2.B

```

FibonacciHelper(n)
{
    memo = tablica o rozmiarze n+1 wypełniona zerami
    return TopDownFibonacci(n, memo)
}

TopDownFibonacci(n, memo)
{
    if n = 0 or n = 1 then
        return n
    if memo[n] != 0 then
        return memo[n]
    memo[n] = TopDownFibonacci(n-1, memo) + TopDownFibonacci(n-2, memo)
    return memo[n]
}
    
```

Dzięki tej technice programowania dynamicznego, zwanej **memoizacją**, każda wartość $F(i)$ jest obliczana dokładnie jeden raz. Jeżeli dany podproblem pojawi się ponownie, jego wynik zostanie odczytany z tablicy. Rysunek 2 przedstawia drzewo wywołań dla $n = 5$ z zastosowaniem memoizacji. Widoczna jest eliminacja powtarzających się podproblemów.



Rysunek 2 Drzewo wywołań FibonacciHelper dla $n=5$ w podejściu top-down wykorzystującym memoizację
(źródło: opracowanie własne)

W efekcie złożoność czasowa algorytmu zostaje zredukowana do $O(n)$ w porównaniu z podejściem naiwnym. Złożoność pamięciowa również ma rząd $O(n)$,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



ponieważ obejmuje zarówno tablicę pomocniczą, jak i stos wywołań o maksymalnej głębokości n .

III. Podejście bottom-up

Jak już wcześniej wspomniano, w programowaniu dynamicznym wyniki mniejszych podproblemów są zapamiętywane i wykorzystywane ponownie. W podejściu bottom-up rezygnujemy z rekurencji i budujemy rozwiązanie iteracyjnie – od najmniejszych wartości do żadanego $F(n)$. Zamiast konstruować drzewo wywołań, kolejne wartości $F(0), F(1), \dots, F(n)$ są wyznaczane [dokładnie jeden raz] w porządku rosnącym i zapisywane w tablicy.

Algorytm P.2.2.C

```
BottomUpTabFibonacci(n)
{
    Utwórz tablicę  $F[0..n]$ 
     $F[0] = 0$ 

    if  $n > 0$  then
         $F[1] = 1$ 

    for  $i = 2$  to  $n$  do
         $F[i] = F[i-2] + F[i-1]$ 

    return  $F[n]$ 
}
```

Rysunek 3 przedstawia kolejne etapy obliczania $F(n)$ dla $n = 5$ w podejściu bottom-up.



Rysunek 3 Kolejne kroki (od lewej do prawej) w obliczaniu $F(n)$ dla $n = 5$ za pomocą podejścia bottom-up
(źródło: opracowanie własne)

Jeżeli celem jest wyznaczenie i zwrócenie tylko n -tego elementu ciągu, przechowywanie całej tablicy nie jest konieczne. W danym kroku potrzebne są wyłącznie dwie poprzednie wartości ciągu. Na przykład, gdy próbujemy obliczyć $F(3)$, potrzebujemy tylko wartości $F(2)$ i $F(1)$.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Algorytm P.2.2.D

```
BottomUpFibonacci(n)
{
    if n = 0 or n = 1 then
        return n
    a = 0
    b = 1

    for i = 2 to n do
        temp = a + b
        a = b
        b = temp

    return b
}
```

W tej wersji złożoność czasowa algorytmu pozostaje liniowa – $O(n)$, a złożoność pamięciowa zostaje zredukowana do $O(1)$.

Przykład 2.3. Podejście brute force

Problem: Dana jest tablica nieujemnych liczb całkowitych. Należy wybrać jak najmniejszą liczbę elementów z tej tablicy, tak aby suma wybranych elementów była większa niż suma wszystkich pozostałych elementów. Wynikiem algorytmu powinna być minimalna liczba elementów spełniających ten warunek.

Brute force

W podejściu naiwnym [brute force] rozważamy wszystkie możliwe podzbiory elementów tablicy. Dla każdego podzbioru obliczamy sumę jego elementów i porównujemy ją z sumą pozostałych elementów tablicy. Jeśli suma wybranego podzbioru jest większa niż suma reszty, sprawdzamy, czy liczba jego elementów jest mniejsza od dotychczas znalezionej liczby elementów. Ostatecznie wybieramy najmniejszy podzbiór spełniający warunek.

Ponieważ tablica o długości n ma 2^n podzbiorów, a obliczenie sumy jednego podzbioru może zająć do n operacji, całkowita złożoność czasowa wynosi $O(n \cdot 2^n)$. Metoda jest nieefektywna dla większych danych wejściowych.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Algorytm P.2.3

```

MinSubsetBruteForce(A)
{
    n = length(A)
    totalSum = sum(A)
    minCount = n

    for mask = 0 to (2^n) - 1 do
        currentSum = 0
        count = 0
        for i = 0 to n - 1 do
            if ((mask >> i) AND 1) = 1 then
                currentSum = currentSum + A[i]
                count = count + 1
            end if
        end for

        if currentSum > totalSum - currentSum then
            if count < minCount then
                minCount = count
            end if
        end if
    end for

    return minCount
}

```

Rozważmy to podejście na przykładowej tablicy liczb całkowitych [2, 9, 5, 3]. Suma wszystkich elementów wynosi 19.

Rozpatrujemy wszystkie podzbiory S (jest ich $2^4 = 16$, wliczając zbiór pusty).

Podzbiór	Maska bitowa	Suma(S)	Suma reszty	Warunek spełniony?
{ }	0000	0	19	Nie
{2}	1000	2	17	Nie
{9}	0100	9	10	Nie
{5}	0010	5	14	Nie
{3}	0001	3	16	Nie
{2,9}	1100	11	8	Tak
{2,5}	1010	7	12	Nie
{2,3}	1001	5	14	Nie
{9,5}	0110	14	5	Tak



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



{9,3}	0101	12	7	Tak
{5,3}	0011	8	11	Nie
{2,9,5}	1110	16	3	Tak
{2,9,3}	1101	14	5	Tak
{2,5,3}	1011	10	9	Tak
{9,5,3}	0111	17	2	Tak
{2,9,5,3}	1111	19	0	Tak

Można zauważyć, że najmniejszy podzbiór spełniający warunek ma 2 elementy (np. {2,9}, {9,5}, {9,3}).

Przykład 2.4. Podejście zachłanne

W tym przykładzie rozważymy rozwiązanie problemu z **Przykładu 2.3** za pomocą algorytmu zachłannego.

W podejściu zachłannym wykorzystujemy fakt, że wszystkie elementy są nieujemne. Aby uzyskać sumę większą niż suma pozostałych elementów przy minimalnej liczbie wybranych wartości, należy w pierwszej kolejności wybierać elementy największe (tu właśnie mamy myślenie zachłanne).

Algorytm P.2.4

```

MinSubsetGreedy(A)
{
    Sortuj A malejąco
    totalSum = suma wszystkich elementów z A
    halfSum = totalSum / 2
    currentSum = 0
    count = 0

    for i = 0 to length(A) - 1 do
        currentSum = currentSum + A[i]
        count = count + 1

        if currentSum > halfSum then
            return count

    return count
}
    
```

Algorytm rozpoczyna od sortowania tablicy malejąco, następnie oblicza sumę wszystkich elementów (totalSum) i dodaje kolejne największe wartości do bieżącej sumy (currentSum). Proces kończy się w momencie, gdy suma wybranych elementów przekroczy sumę pozostałych lub połowę sumy wszystkich elementów.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



W tym podejściu stosowane jest sortowanie, dlatego złożoność czasowa wynosi $O(n \cdot \log n)$.

Rozważmy teraz to podejście na przykładowej tablicy liczb całkowitych [2, 9, 5, 3].

- 1) Sortujemy tablicę malejąco: [9, 5, 3, 2].
- 2) totalSum = 19, halfSum = 9,5, currentSum = 0, count = 0.
- 3) Dodajemy kolejne elementy do currentSum (zaczynając od największego):
 - a) Dodajemy 9
currentSum = 9, sprawdzamy warunek $9 \nless 9,5$.
Warunek nie jest spełniony, dlatego dodajemy kolejną wartość.
 - b) Dodajemy 5
currentSum = 14, sprawdzamy warunek $14 > 9,5$.
Warunek zostaje spełniony po dodaniu dwóch elementów {9, 5}, więc algorytm zwraca wartość 2.

Ćwiczenia samodzielne

Zadanie 2.1. Znajdowanie maksimum w tablicy – metoda liniowa i „dziel i zwyciężaj”

Zaimplementuj algorytm znajdowania maksimum w tablicy w dwóch wersjach:

- a) metodą prostą (przebieg liniowy wszystkich elementów),
- b) metodą „dziel i zwyciężaj”.

Wyznacz liczbę porównań w poszczególnych podejściach. Określ złożoność czasową obu wersji algorytmu.

Zadanie 2.2. Ciąg Tribonacciego - programowanie dynamiczne

Zaimplementuj algorytm obliczania n-tego wyrazu ciągu Tribonacciego z wykorzystaniem programowania dynamicznego w dwóch podejściach:

- a) top-down (rekurencja z memoizacją),
- b) bottom-up (iteracyjnie).

Ciąg Tribonacciego zdefiniowany jest następująco:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



$$T(n) = \begin{cases} 0 & \text{dla } n = 0 \\ 0 & \text{dla } n = 1 \\ 1 & \text{dla } n = 2 \\ T(n-1) + T(n-2) + T(n-3) & \text{dla } n \geq 3 \end{cases}$$

Określ złożoność czasową w obu przypadkach. Wyjaśnij, dlaczego podejście oparte na programowaniu dynamicznym działa szybciej niż naiwna wersja rekurencyjna (bez zapamiętywania wyników).

Zadanie 2.3. Wydawanie reszty o minimalnej liczbie monet – podejście zachłanne

Zaimplementuj algorytm zachłanny dla problemu wydawania reszty, który dla zadanej kwoty K oraz zestawu nominałów N dobiera je w taki sposób, aby zminimalizować liczbę użytych jednostek. Zakładamy, że każdy nominał jest dostępny w nieograniczonej liczbie.

Metoda powinna wypisać liczbę użytych jednostek każdego nominału oraz zwrócić łączną liczbę użytych jednostek. Sprawdź działanie algorytmu dla następującego zestawu nominałów: 1, 2, 5, 10, 20, 50, 100, 200, 500.

Zadanie 2.4. Para liczb o zadanej sumie – podejście siłowe i zoptymalizowane

Zaimplementuj algorytm sprawdzający, czy w tablicy istnieje para liczb, której suma jest równa zadanej wartości s . Przygotuj dwa rozwiązania:

- a) w wersji naiwnej (brute force), polegającej na sprawdzeniu wszystkich możliwych par elementów,
- b) w wersji zoptymalizowanej, wykorzystującej sortowanie tablicy oraz technikę dwóch wskaźników.

Porównaj złożoność czasową obu rozwiązań.

Pytania pomocnicze

- 1) Jakie są główne różnice między podejściem zstępującym a wstępującym?
- 2) Na czym polega strategia „dziel i zwyciężaj”?
- 3) Czym różni się programowanie dynamiczne od strategii „dziel i zwyciężaj”?
- 4) Na czym polega idea algorytmu zachłannego?
- 5) Dlaczego algorytm zachłanny nie zawsze prowadzi do rozwiązania optymalnego?
- 6) Czym charakteryzują się algorytmy siłowe (brute force)?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- 7) Jakie są konsekwencje wysokiej złożoności czasowej algorytmów siłowych?
- 8) Czy rekurencja zawsze oznacza zastosowanie strategii „dziel i zwyciężaj”?
- 9) W projekcie zespołowym kilka osób implementuje różne moduły systemu. Które podejście projektowe będzie bardziej odpowiednie?
- 10) Czy można połączyć różne strategie algorytmiczne w jednym rozwiązaniu? Podaj przykład.

Podsumowanie

Podczas laboratorium przeanalizowano podstawowe metody projektowania algorytmów: zstępującą [top-down] oraz wstępującą [bottom-up]. Omówiono również wybrane strategie algorytmiczne, takie jak „dziel i zwyciężaj”, programowanie dynamiczne, algorytmy zachłanne oraz algorytmy siłowe. Zwrócono uwagę na ich charakterystyczne cechy, różnice w podejściu do rozwiązywania problemów oraz wpływ na złożoność czasową i pamięciową algorytmów.

Literatura

- [1] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C., *Wprowadzenie do algorytmów*, PWN, Warszawa 2023. s. 71-100, 342-413.
- [2] Aho A. V., Hopcroft J.E., Ullman J.D., *Algorytmy i struktury danych*, Helion, Gliwice 2003. s. 321-350.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Materiały do przedmiotu:

Algorytmy i struktury danych

Laboratorium nr 3

**Implementacja wybranych algorytmów
sortujących i wyszukiwujących**

Autorzy:

dr hab. Adam Kiersztyn
mgr Krystyna Kiersztyn



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Spis treści

Wprowadzenie, tematyka zajęć	31
Cel ćwiczenia/laboratorium	31
Instrukcja laboratoryjna/ćwiczeniowa	32
Ćwiczenia samodzielne	54
Pytania pomocnicze	56
Podsumowanie	57
Literatura	57



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wprowadzenie, tematyka zajęć

Algorytmy wyszukiwania i sortowania stanowią jeden z podstawowych elementów każdego systemu informatycznego. Występują zarówno w prostych programach przetwarzających niewielkie zbiory danych, jak i w rozbudowanych aplikacjach operujących na dużych ilościach informacji. Trudno wyobrazić sobie efektywne przetwarzanie danych bez mechanizmów pozwalających na ich uporządkowanie oraz szybkie odnajdywanie określonych wartości, a znajomość tych algorytmów jest niezbędna w pracy każdego programisty.

Algorytmy wyszukiwania służą do znajdowania wskazanego elementu w analizowanym zbiorze danych. W ramach laboratorium analizowane są podstawowe metody wyszukiwania, takie jak **wyszukiwanie liniowe** (*Linear Search*), **wyszukiwanie liniowe z wartownikiem** (*Linear Search with Sentinel*) oraz **wyszukiwanie binarne** (*Binary Search*). Algorytmy te różnią się zarówno wymaganiami dotyczącymi organizacji danych wejściowych, jak i liczbą wykonywanych operacji, co bezpośrednio wpływa na ich wydajność. Wyszukiwanie liniowe może być stosowane do danych nieuporządkowanych, natomiast wyszukiwanie binarne wymaga wcześniejszego posortowania danych, oferując w zamian znaczące przyspieszenie procesu wyszukiwania.

Algorytmy sortowania służą do uporządkowania elementów zbioru według określonego kryterium, na przykład w porządku rosnącym lub malejącym. W ramach niniejszego laboratorium omawiane są wybrane algorytmy sortowania oparte na porównaniach elementów, takie jak **sortowanie bąbelkowe** (*Bubble Sort*), **sortowanie przez wybór** (*Selection Sort*), **sortowanie przez wstawianie** (*Insertion Sort*) oraz **sortowanie szybkie** (*QuickSort*).

Cel ćwiczenia/laboratorium

Po zakończeniu laboratorium student:

- rozumie zasadę działania podstawowych algorytmów sortowania i wyszukiwania,
- potrafi opisać krok po kroku działanie algorytmu na konkretnych danych wejściowych,
- rozróżnia przypadek optymistyczny, średni i pesymistyczny w analizie algorytmów,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- zna wpływ struktury danych wejściowych na liczbę wykonywanych operacji,
- potrafi zaimplementować wybrane algorytmy sortowania i wyszukiwania.

Instrukcja laboratoryjna/ćwiczeniowa

Przykład 3.1. Sortowanie bąbelkowe

W celu lepszego zrozumienia działania algorytmów sortowania zostanie przeanalizowane **sortowanie bąbelkowe** na przykładzie zadanej tablicy liczb całkowitych. Rozważone zostaną trzy przypadki: optymistyczny, średni oraz pesymistyczny. Sortowanie bąbelkowe w swojej podstawowej wersji jest jednym z najprostszych algorytmów sortowania. Jego działanie polega na wielokrotnym przechodzeniu przez tablicę i porównywaniu sąsiadujących ze sobą elementów. Jeżeli elementy te znajdują się w niewłaściwej kolejności, następuje ich zamiana. Proces ten jest powtarzany określoną liczbę razy, niezależnie od stopnia uporządkowania danych wejściowych. W wyniku kolejnych przebiegów największe elementy „wypływają” na koniec tablicy.

Poniżej przedstawiono pseudokod algorytmu sortowania bąbelkowego, stanowiący podstawę dalszej analizy jego działania.

Algorytm P.3.1. Sortowanie bąbelkowe

```
BubbleSort(A)
{
    n = length(A)
    for i = 1 to n - 1 do
    {
        for j = 1 to n - i do
        {
            if (A[j] > A[j + 1]) then
                Swap(A[j], A[j + 1])
        }
    }
}
```

Na podstawie pseudokodu przeanalizujemy działanie algorytmu dla przypadku optymistycznego, średniego oraz pesymistycznego.

A. Przypadek optymistyczny

W przypadku optymistycznym zakłada się, że dane wejściowe są już posortowane rosnąco:

A = [1, 2, 3, 4, 5]



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Intuicyjnie mogłoby się wydawać, że algorytm bardzo szybko zakończy działanie, ponieważ nie ma żadnych elementów do przestawienia. Jednak algorytm konsekwentnie wykonuje wszystkie zaplanowane przebiegi, porównując każdą parę sąsiednich elementów.

W zapisie przebiegu algorytmu na pomarańczowo zaznaczono parę elementów aktualnie porównywanych w danym kroku (j) pętli wewnętrznej.

I. Pierwsza iteracja (i = 1)

j=1	1	2	3	4	5	1 < 2 – bez zmian
j=2	1	2	3	4	5	2 < 3 – bez zmian
j=3	1	2	3	4	5	3 < 4 – bez zmian
j=4	1	2	3	4	5	4 < 5 – bez zmian

Podczas każdego porównania algorytm „stwierdza”, że elementy znajdują się w poprawnej kolejności, dlatego nie dochodzi do żadnej zamiany. Po zakończeniu pętli wewnętrznej tablica pozostaje bez zmian. Największy element (zaciemiony na zielono) znajduje się na końcu tablicy (choć w tym przypadku nie jest to efektem działania algorytmu, lecz początkowego uporządkowania danych).

1	2	3	4	5
---	---	---	---	---

II. Druga iteracja (i = 2)

W tej iteracji liczba porównań w pętli wewnętrznej będzie już mniejsza o jeden.

j=1	1	2	3	4	5	1 < 2 – bez zmian
j=2	1	2	3	4	5	2 < 3 – bez zmian
j=3	1	2	3	4	5	3 < 4 – bez zmian

Również w tym przebiegu nie występują żadne zamiany. Drugi co do wielkości element jest na właściwej pozycji w tablicy.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



1	2	3	4	5
---	---	---	---	---

III. Trzecia iteracja (i = 3)

W tej iteracji liczba porównań w pętli wewnętrznej znowu się zmniejsza o jeden.

j=1	1	2	3	4	5	1 < 2 – bez zmian
-----	---	---	---	---	---	-------------------

j=2	1	2	3	4	5	2 < 3 – bez zmian
-----	---	---	---	---	---	-------------------

Po zakończeniu tej iteracji trzeci co do wielkości element również znajduje się na swojej docelowej pozycji.

1	2	3	4	5
---	---	---	---	---

IV. Czwarta iteracja (i = 4)

j=1	1	2	3	4	5	1 < 2 – bez zmian
-----	---	---	---	---	---	-------------------

W tej iteracji mamy już tylko jedno porównanie, ponieważ pozostałe elementy w tablicy są już uporządkowane.

1	2	3	4	5
---	---	---	---	---

Mimo że w trakcie działania algorytmu nie dochodzi do ani jednej zamiany elementów, czas działania algorytmu nie ulega skróceniu. Wszystkie porównania są wykonywane tak samo, jak w przypadku danych całkowicie nieuporządkowanych. W efekcie nawet dla posortowanej tablicy algorytm realizuje pełną sekwencję działań.

Liczba porównań:

$$(n - 1) + (n - 2) + \dots + 1 = \frac{n(n - 1)}{2}$$

Liczba zamian: 0

Złożoność czasowa: $O(n^2)$

B. Przypadek średni

Przypadek średni odpowiada sytuacji, w której dane wejściowe są w losowej kolejności:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



$$A=[3, 1, 4, 2, 5]$$

Na początku algorytm wykonuje wiele zamian, ponieważ część elementów znajduje się w niewłaściwej kolejności. W trakcie kolejnych przebiegów większe wartości stopniowo przesuwają się w prawo, aż tablica staje się uporządkowana.

Podobnie jak wcześniej, w zapisie przebiegu algorytmu na pomarańczowo zaznaczono parę elementów aktualnie porównywanych w danym kroku pętli wewnętrznej.

I. Pierwsza iteracja ($i = 1$)

$j=1$

3	1	4	2	5
---	---	---	---	---

 $3 > 1$ - zamiana

$j=2$

1	3	4	2	5
---	---	---	---	---

 $3 < 4$ - bez zmian

$j=3$

1	3	4	2	5
---	---	---	---	---

 $4 > 2$ - zamiana

$j=4$

1	3	2	4	5
---	---	---	---	---

 $4 < 5$ - bez zmian

Po zakończeniu pierwszej iteracji największy element (5) znajduje się na końcu tablicy.

1	3	2	4	5
---	---	---	---	---

II. Druga iteracja ($i = 2$)

W tej iteracji liczba porównań w pętli wewnętrznej będzie już mniejsza o jeden.

$j=1$

1	3	2	4	5
---	---	---	---	---

 $1 < 3$ - bez zmian

$j=2$

1	3	2	4	5
---	---	---	---	---

 $3 > 2$ - zamiana

$j=3$

1	2	3	4	5
---	---	---	---	---

 $3 < 4$ - bez zmian

Po zakończeniu tej iteracji, drugi co do wielkości element (4) jest na właściwej pozycji w tablicy. Jak można zauważyć, tablica jest już uporządkowana, jednak algorytm nadal wykonuje zaplanowane porównania.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



1	2	3	4	5
---	---	---	---	---

III. Trzecia iteracja (i = 3)

W tej iteracji liczba porównań w pętli wewnętrznej znowu się zmniejsza o jeden.

j=1	1	2	3	4	5	$1 < 2$ – bez zmian
-----	---	---	---	---	---	---------------------

j=2	1	2	3	4	5	$2 < 3$ – bez zmian
-----	---	---	---	---	---	---------------------

Po zakończeniu tej iteracji trzeci co do wielkości element znajduje się na swojej docelowej pozycji.

1	2	3	4	5
---	---	---	---	---

IV. Czwarta iteracja (i = 4)

Ostatnia iteracja obejmuje już tylko jedno porównanie.

j=1	1	2	3	4	5	$1 < 2$ – bez zmian
-----	---	---	---	---	---	---------------------

Po zakończeniu działania algorytmu otrzymujemy posortowaną tablicę:

1	2	3	4	5
---	---	---	---	---

W początkowych iteracjach liczba zamian jest stosunkowo duża, jednak z każdym kolejnym przebiegiem tablica staje się coraz bardziej uporządkowana.

Liczba porównań:

$$(n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2}$$

Liczba zamian: 3

Złożoność czasowa: $O(n^2)$

C. Przypadek pesymistyczny

Najmniej korzystna sytuacja dla sortowania występuje wtedy, gdy dane są posortowane w kolejności odwrotnej (malejącej):

$$A=[5, 4, 3, 2, 1]$$



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



W zapisie przebiegu algorytmu na pomarańczowo oznaczono parę elementów aktualnie porównywanych w danym kroku pętli wewnętrznej.

I. Pierwsza iteracja ($i = 1$)

$j=1$

5	4	3	2	1
---	---	---	---	---

 $5 > 4$ – zamiana

$j=2$

4	5	3	2	1
---	---	---	---	---

 $5 > 3$ – zamiana

$j=3$

4	3	5	2	1
---	---	---	---	---

 $5 > 2$ – zamiana

$j=4$

4	3	2	5	1
---	---	---	---	---

 $5 > 1$ – zamiana

Po zakończeniu pierwszej iteracji największy element (5) znajduje się na końcu tablicy.

4	3	2	1	5
---	---	---	---	---

II. Druga iteracja, $i = 2$

$j=1$

4	3	2	1	5
---	---	---	---	---

 $4 > 3$ – zamiana

$j=2$

3	4	2	1	5
---	---	---	---	---

 $4 > 2$ – zamiana

$j=3$

3	2	4	1	5
---	---	---	---	---

 $4 > 1$ – zamiana

Po zakończeniu tej iteracji dwa największe elementy (4 i 5) znajdują się już na właściwych pozycjach.

3	2	1	4	5
---	---	---	---	---

III. Trzecia iteracja ($i = 3$)

$j=1$

3	2	1	4	5
---	---	---	---	---

 $3 > 2$ – zamiana



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



j=2

2	3	1	4	5
---	---	---	---	---

 3 > 1 – zamiana

Po tej iteracji trzy największe elementy są już uporządkowane.

2	1	3	4	5
---	---	---	---	---

IV. Czwarta iteracja (i = 4)

j=1

2	1	3	4	5
---	---	---	---	---

 2 > 1 – zamiana

Po wykonaniu ostatniej iteracji tablica jest posortowana rosnąco.

1	2	3	4	5
---	---	---	---	---

W przypadku pesymistycznym każde porównanie prowadzi do zamiany elementów, co oznacza maksymalną możliwą liczbę operacji. Każdy przebieg pętli zewnętrznej powoduje przesunięcie dokładnie jednego elementu na jego docelową pozycję.

Liczba porównań:

$$(n - 1) + (n - 2) + \dots + 1 = \frac{n(n - 1)}{2}$$

Liczba zamian: 10

Złożoność czasowa: $O(n^2)$

Wnioski

Analiza trzech przypadków pokazuje, że w tej wersji sortowanie bąbelkowe nie potrafi dostosować swojego działania do charakteru danych wejściowych. Niezależnie od tego, czy tablica jest już posortowana, zawiera elementy w losowej kolejności, czy też jest uporządkowana malejąco, algorytm wykonuje tę samą liczbę porównań. Różnice pomiędzy poszczególnymi przypadkami dotyczą głównie liczby zamian elementów, co wpływa na rzeczywisty czas wykonania algorytmu, lecz nie zmienia jego klasy złożoności obliczeniowej.

Przykład 3.2. Wyszukiwanie liniowe

Rozważmy algorytm:

Algorytm P.3.2. Wyszukiwanie liniowe

```
LinearSearch(A, x)
{
  n = length(A)
  i = 1
  while (i < n) do
  {
    if A[i] == x then
      return i
    i = i + 1
  }
  return -1
}
```

W celu lepszego zrozumienia działania algorytmów wyszukiwania zostanie przeanalizowany algorytm wyszukiwania liniowego na przykładzie zadanej tablicy liczb całkowitych. Rozważone zostaną trzy przypadki: optymistyczny, średni oraz pesymistyczny.

Wyszukiwanie liniowe polega na sekwencyjnym przeglądaniu elementów tablicy, począwszy od pierwszego elementu, aż do odnalezienia szukanej wartości lub do sprawdzenia wszystkich elementów. Algorytm nie wymaga wcześniejszego uporządkowania danych.

Rozważmy tablicę $A = [4, 7, 2, 9, 1]$ oraz wyszukiwaną wartość $x = 4$.

A. Przypadek optymistyczny

W przypadku optymistycznym zakłada się, że szukany element x znajduje się na pierwszej pozycji w tablicy $A = [4, 7, 2, 9, 1]$.

Algorytm rozpoczyna działanie od porównania szukanej wartości z pierwszym elementem tablicy. Już przy pierwszym porównaniu okazuje się, że szukany element (w tym przypadku 4) jest równy szukanej wartości x . Algorytm natychmiast kończy działanie i zwraca indeks znalezionego elementu.

W tym przypadku wykonane zostaje tylko jedno porównanie, a dalsze elementy tablicy nie są sprawdzane.

Liczba porównań: 1

Złożoność czasowa: $O(1)$



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



B. Przypadek średni

Przypadek średni odpowiada sytuacji, w której szukany element znajduje się w losowym miejscu tablicy, na przykład na trzeciej pozycji [w tablicy pięcioelementowej].

Rozważmy wyszukiwanie wartości $x = 2$ w tablicy $A = [4, 7, 2, 9, 1]$.

Przebieg algorytmu:

W pierwszej iteracji wykonujemy porównanie: $4 \neq 2$

	1	2	3	4	5
i=1	4	7	2	9	1
	$\neq 2$				

W drugiej iteracji wykonujemy porównanie: $7 \neq 2$

	1	2	3	4	5
i=2	4	7	2	9	1
		$\neq 2$			

W trzeciej iteracji wykonujemy porównanie: $2 = 2$

	1	2	3	4	5
i=3	4	7	2	9	1
			$= 2$		

Algorytm kończy działanie po wykonaniu kilku porównań, nie sprawdzając pozostałych elementów tablicy. W typowym przypadku średnim liczba porównań jest proporcjonalna do połowy rozmiaru tablicy.

Liczba porównań (około): $\frac{n}{2}$

Złożoność czasowa: $O(n)$

C. Przypadek pesymistyczny

W przypadku pesymistycznym szukany element x znajduje się na ostatniej pozycji w tablicy, albo nie występuje w tablicy w ogóle.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Rozważmy wyszukiwanie wartości $x = 5$ w tablicy $A = [4, 7, 2, 9, 1]$.

Algorytm kolejno porównuje x z każdym elementem tablicy:

	1	2	3	4	5
i=1	4	7	2	9	1
	≠ 5				

	1	2	3	4	5
i=2	4	7	2	9	1
		≠ 5			

	1	2	3	4	5
i=3	4	7	2	9	1
			≠ 5		

	1	2	3	4	5
i=4	4	7	2	9	1
				≠ 5	

	1	2	3	4	5
i=5	4	7	2	9	1
					≠ 5

Po sprawdzeniu wszystkich elementów algorytm stwierdza brak szukanej wartości i kończy działanie. W tym przypadku wykonana zostaje maksymalna liczba porównań.

Liczba porównań: n

Złożoność czasowa: $O(n)$

Wnioski

Analiza trzech przypadków pokazuje, że złożoność algorytmu wyszukiwania liniowego silnie zależy od położenia szukanego elementu w tablicy. W najlepszym przypadku algorytm kończy działanie po jednym porównaniu, natomiast w



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



przypadku pesymistycznym musi sprawdzić wszystkie elementy tablicy. Średni czas działania algorytmu rośnie liniowo wraz z rozmiarem danych wejściowych, co prowadzi do złożoności czasowej rzędu $O(n)$.

Przykład 3.3. Sortowanie przez wybór

Selection Sort jest prostym algorytmem sortowania opartym na metodzie wybierania kolejnych elementów o najmniejszej wartości z nieposortowanej części tablicy i umieszczania ich na odpowiednich pozycjach w części już posortowanej.

Algorytm działa in-place (nie wymaga dodatkowej pamięci pomocniczej) i dzieli tablicę na dwie części:

- część posortowaną – znajdującą się na początku tablicy,
- część nieposortowaną – zawierającą elementy jeszcze nieprzetworzone.

W każdej iteracji algorytm:

1. Wyszukuje najmniejszy element w nieposortowanej części tablicy,
2. Zamienia go z pierwszym elementem tej części,
3. „Powiększa” fragment posortowany o jeden element.

Algorytm P.3.3. Sortowanie przez wybór

```
SelectionSort(A)
{
    n = length(A)

    for i = 1 to n - 1 do
    {
        min = i

        for j = i + 1 to n do
        {
            if (A[j] < A[min]) then
                min = j
        }

        if (i != min) then
            Swap(A[i], A[min])
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Rozważmy tablicę wejściową: $A=[64, 25, 12, 22, 11]$, gdzie liczba elementów wynosi $n = 5$. Przyjmujemy indeksowanie tablicy od 1.

Na początku wszystkie elementy znajdują się w części nieposortowanej. W każdej iteracji algorytmu Selection Sort wyszukiwany jest najmniejszy element z części nieposortowanej, a następnie umieszczany na pierwszej pozycji tej części.

Iteracja 1

Przyjmujemy, że najmniejszy element znajduje się na pierwszej pozycji nieposortowanej części, czyli **min = 1** (wartość 64). Następnie porównujemy kolejne elementy z aktualnym minimum:

- $25 < 64 \rightarrow \mathbf{min = 2}$
- $12 < 25 \rightarrow \mathbf{min = 3}$
- $22 > 12 \rightarrow \mathbf{bez zmiany (min = 3)}$
- $11 < 12 \rightarrow \mathbf{min = 5}$

Najmniejszy element to **11**, jego indeks to **5**. Zatem wykonujemy zamianę elementów: $A[1] \leftrightarrow A[5]$. Nowa tablica ma postać:

	1	2	3	4	5
	11	25	12	22	64

przy czym:

- część posortowana (zaczieniona na zielono): [11]
- część nieposortowana: [25, 12, 22, 64]

Iteracja 2

Rozpatrujemy część nieposortowaną. Przyjmujemy **min = 2** (wartość 25). Następnie wykonujemy porównania:

- $12 < 25 \rightarrow \mathbf{min = 3}$
- $22 > 12 \rightarrow \mathbf{bez zmiany}$
- $64 > 12 \rightarrow \mathbf{bez zmiany}$

Najmniejszy element to **12** z indeksem **3**. Wykonujemy zamianę elementów $A[2] \leftrightarrow A[3]$. Nowa tablica ma postać (na zielono zaznaczono część posortowaną):



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



1	2	3	4	5
11	12	25	22	64

Iteracja 3

Rozpatrujemy część nieposortowaną. Przyjmujemy **min = 3** (wartość 25). Następnie wykonujemy porównania:

- $22 < 25 \rightarrow \mathbf{min = 4}$
- $64 > 22 \rightarrow \mathbf{bez zmiany}$

Najmniejszy element ma indeks 4, a jego wartość to **22**. Wykonujemy zamianę: $A[3] \leftrightarrow A[4]$. Nowa tablica ma postać:

1	2	3	4	5
11	12	22	25	64

Iteracja 4

Rozpatrujemy część nieposortowaną. Przyjmujemy **min = 4** (wartość 25). Następnie wykonujemy porównanie:

- $64 > 25 \rightarrow \mathbf{bez zmiany}$

Najmniejszy element znajduje się już na właściwej pozycji, więc zamiana nie jest konieczna.

Tablica po zakończeniu działania algorytmu ma postać:

1	2	3	4	5
11	12	22	25	64

Liczba porównań: $\frac{n(n-1)}{2}$.

Złożoność czasowa: w przypadku najlepszym, średnim i najgorszym to $O(n^2)$.

Przykład 3.4. Sortowanie przez wstawianie

Algorytm Insertion Sort (sortowanie przez wstawianie) polega na stopniowym budowaniu części posortowanej tablicy. W każdej iteracji pierwszy element z części nieposortowanej (oznaczony jako *key*) jest wstawiany na właściwe miejsce w części posortowanej poprzez przesuwanie większych elementów o jedną pozycję w prawo.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Algorytm P.3.4. Sortowanie przez wstawianie

```

InsertionSort(A)
{
    n = length(A)

    for i = 2 to n do
    {
        key = A[i]
        j = i - 1

        while (j >= 1 and A[j] > key) do
        {
            A[j + 1] = A[j]
            j = j - 1
        }
        A[j + 1] = key
    }
}
    
```

Rozważmy tablicę wejściową **A=[8, 3, 5, 2, 6]**, liczba elementów tablicy to $n = 5$. Przyjmujemy indeksowanie tablicy od 1.

Na początku pierwszy element tablicy traktujemy jako część posortowaną.

Iteracja 1

- Część posortowana (zaczynając od zielono): [8]
- Część nieposortowana: [3, 5, 2, 6]

1	2	3	4	5
8	3	5	2	6

Rozpatrywany element to $key = 3$. Porównujemy go z elementem części posortowanej:

- $8 > 3$ – przesuwamy 8 o jedną pozycję w prawo.

Wstawiamy element **3** na właściwe miejsce. Nowa tablica ma postać:

1	2	3	4	5
3	8	5	2	6



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Iteracja 2

Rozpatrywany element to $key = 5$. Porównujemy go z elementami części posortowanej:

- $8 > 5$ – przesuwamy 8 w prawo
- $3 < 5$ – kończymy przesuwanie

Wstawiamy element **5** na odpowiednią pozycję w części posortowanej (między 3 i 8). Nowa tablica ma postać:

1	2	3	4	5
3	5	8	2	6

Iteracja 3

Rozpatrywany element to $key = 2$. Porównujemy go z elementami części posortowanej:

- $8 > 3$ – przesuwamy 8
- $5 > 2$ – przesuwamy 5
- $3 > 2$ – przesuwamy 3

Wstawiamy element **2** na początek tablicy. Nowa tablica ma postać:

1	2	3	4	5
2	3	5	8	6

Iteracja 4

Rozpatrywany element to $key = 6$. Porównujemy go z elementami części posortowanej:

- $6 < 8$ – przesuwamy 8
- $6 > 5$ – kończymy przesuwanie

Wstawiamy element **6** między **5** i **8** w części posortowanej. Po wykonaniu $(n - 1)$ iteracji wszystkie elementy tablicy znajdują się w części posortowanej, co oznacza zakończenie działania algorytmu.

Tablica po zakończeniu działania algorytmu ma postać:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



1	2	3	4	5
2	3	5	6	8

Złożoność czasowa algorytmu:

- Przypadek optymistyczny: $O(n)$
- Przypadek średni i pesymistyczny: $O(n^2)$

Przykład 3.5. Sortowanie szybkie

Algorytm P.3.5. Sortowanie szybkie

```

QuickSort(A, start, end)
{
    if (start < end) then
    {
        p = Partition(A, start, end)
        QuickSort(A, start, p - 1)
        QuickSort(A, p + 1, end)
    }
}

Partition(A, start, end)
{
    pivot = A[end]
    i = start - 1

    for j = start to end - 1 do
    {
        if (A[j] <= pivot) then
        {
            i = i + 1
            Swap(A[i], A[j])
        }
    }

    Swap(A[i + 1], A[end])

    return i + 1
}

```

Algorytm sortowania szybkiego [QuickSort] należy do algorytmów typu dziel i zwyciężaj. Polega na wybraniu jednego elementu tablicy, zwanego **pivotem**, a następnie podziale tablicy na dwie części:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- elementy mniejsze lub równe pivotowi,
- elementy większe od pivota.

Po wykonaniu podziału pivot znajduje się na właściwej pozycji w tablicy. Następnie algorytm jest wywoływany rekurencyjnie dla lewej i prawej części tablicy.

Rozważmy tablicę wejściową $A=[9, 5, 7, 4, 8, 6]$.

Pierwszy podział tablicy

Jako pivot wybieramy ostatni element, **pivot = 6**.

1	2	3	4	5	6
9	5	7	4	8	6

Teraz porównujemy kolejne elementy z pivotem i porządkujemy je względem jego wartości. Elementy mniejsze od pivota przenosimy na lewo.

- $5 < 6$

1	2	3	4	5	6
5	9	7	4	8	6

- $4 < 6$

1	2	3	4	5	6
5	4	7	9	8	6

Pozostałe elementy są większe od pivota. Ustawiamy pivot na granicy między elementami mniejszymi i większymi od niego:

1	2	3	4	5	6
5	4	6	9	8	7

Tablica została podzielona na dwa fragmenty (< 6 oraz > 6):



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



1	2	3	4	5	6
5	4	6	9	8	7

Sortowanie lewego fragmentu

Rozważmy fragment tablicy, gdzie elementy mają wartości mniejsze od 6. Wyznaczamy nowego pivota, **pivot = 4**.

1	2
5	4

Nie ma elementów mniejszych od pivota, dlatego przenosimy go na początek:

1	2
4	5

Ten fragment tablicy jest już uporządkowany.

Sortowanie prawego fragmentu

Wracamy do pozostałych elementów (większych od 6). Wyznaczamy nowego pivota, **pivot = 7**.

4	5	6
9	8	7

Nie ma elementów mniejszych od pivota, dlatego przenosimy go na początek tej podtablicy:

4	5	6
7	8	9

Nie ma elementów < 7 w tej podtablicy, dlatego nie analizujemy lewej części podtablicy.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Sortowanie ostatniego fragmentu (prawy)

Rozważmy fragment tablicy, gdzie elementy są większe od 7. Wyznaczamy nowego pivota, **pivot = 9**.

5	6
8	9

Element 8 jest mniejszy od pivota, więc pozostaje po jego lewej stronie. W tym przypadku pivot pozostaje na końcu rozważanego fragmentu, ponieważ wszystkie wcześniejsze elementy są od niego mniejsze.

Po zakończeniu wszystkich wywołań rekurencyjnych otrzymujemy posortowaną tablicę:

1	2	3	4	5	6
4	5	6	7	8	9

Złożoność czasowa

- najlepszy przypadek: $O(n \log n)$
- średni przypadek: $O(n \log n)$
- najgorszy przypadek: $O(n^2)$

Najgorszy przypadek występuje wtedy, gdy pivot za każdym razem dzieli tablicę bardzo nierównomiernie, na przykład gdy jest najmniejszym albo największym elementem.

Przykład 3.6. Wyszukiwanie liniowe z wartownikiem

Wyszukiwanie liniowe z wartownikiem jest modyfikacją klasycznego wyszukiwania liniowego. Algorytm polega na przeszukiwaniu elementów tablicy jeden po drugim aż do momentu znalezienia szukanego elementu. Aby uprościć działanie algorytmu, na końcu tablicy umieszcza się dodatkowy element, zwany **wartownikiem**, którego wartość jest równa wartości elementu poszukiwanego. Dzięki temu w trakcie przeszukiwania nie trzeba sprawdzać, czy osiągnięto koniec tablicy – pętla zawsze zatrzyma się po znalezieniu szukanego elementu lub wartownika. Po zakończeniu wyszukiwania przywracana jest pierwotna wartość ostatniego elementu tablicy, a następnie sprawdzane jest, czy element został znaleziony w oryginalnej tablicy.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Algorytm P.3.6. Wyszukiwanie liniowe z wartownikiem

```
LinearSearchWithSentinel(A, x)
{
    n = length(A)
    last = A[n]

    A[n] = x
    i = 1

    while (A[i] != x) do
        i = i + 1

    A[n] = last

    if (i < n or A[n] == x) then
        return i
    else
        return -1
}
```

Przypadek 1. Szukany element występuje w tablicy

Rozważmy tablicę wejściową $A=[4, 7, 2, 9, 5]$. Załóżmy, że szukamy elementu $x = 9$.

Krok 1

Zapamiętujemy ostatni element tablicy:

$$last = 5$$

Następnie ustawiamy wartownika, czyli wpisujemy szukany element na końcu tablicy:

$$A[5] = 9$$

Tablica przyjmuje postać:

$$[4, 7, 2, 9, 9]$$
Krok 2

Rozpoczynamy przeszukiwanie tablicy od pierwszego elementu.

- $A[1] = 4 \neq 9$
- $A[2] = 7 \neq 9$
- $A[3] = 2 \neq 9$
- $A[4] = 9$



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



W tym momencie kończymy wyszukiwanie, ponieważ znaleźliśmy szukany element.

Krok 3

Przywracamy ostatni element tablicy:

$$A[5] = last = 5$$

Tablica ponownie ma postać:

$$[4, 7, 2, 9, 5]$$

Znaleziony indeks to $i = 4$.

Ponieważ $i < n$, oznacza to, że element został znaleziony w oryginalnej tablicy.

Wynik: szukany element 9 znajduje się na pozycji 4 w tablicy.

Przypadek 2. Szukany element nie występuje w tablicy

Rozważmy tę samą tablicę $A = [4, 7, 2, 9, 5]$. Załóżmy, że szukamy elementu $x = 6$.

Zapamiętujemy ostatni element:

$$last = 5$$

Ustawiamy wartownika:

$$A[5] = 6$$

Tablica chwilowo ma postać:

$$[4, 7, 2, 9, 6]$$

Wykonujemy przeszukiwanie tablicy od pierwszego elementu.

- $A[1] = 4 \neq 6$
- $A[2] = 7 \neq 6$
- $A[3] = 2 \neq 6$
- $A[4] = 9 \neq 6$
- $A[5] = 6$

Wyszukiwanie zatrzymuje się na ostatniej pozycji, czyli na wartowniku. Przywracamy ostatni element:

$$A[5] = 5$$

Ponieważ element został znaleziony dopiero na pozycji wartownika, oznacza to, że nie występował w oryginalnej tablicy.

Wynik: szukany element 6 nie występuje w tablicy.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykład 3.7. Wyszukiwanie binarne

Algorytm **wyszukiwania binarnego** służy do znajdowania elementu w **posortowanej tablicy**. Jego działanie polega na wielokrotnym dzieleniu przeszukiwanego fragmentu tablicy na połowy i sprawdzaniu elementu środkowego.

Jeżeli element środkowy jest równy szukanemu, algorytm kończy działanie. Jeżeli jest większy, przeszukujemy lewą połowę tablicy, a jeżeli mniejszy — prawą połowę tablicy.

Algorytm P.3.7 Wyszukiwanie binarne

```
BinarySearch(A, x)
{
    left = 1
    right = length(A)

    while (left <= right) do
    {
        mid = floor((left + right) / 2)

        if (A[mid] == x) then
            return mid

        if (A[mid] < x) then
            left = mid + 1
        else
            right = mid - 1
    }

    return -1
}
```

Rozważmy posortowaną tablicę:

$A = [2, 4, 6, 8, 10, 12, 14]$

Założmy, że szukamy elementu $x = 10$.

Krok 1

Rozpatrujemy całą tablicę. Środkowy element to 8

2	4	6	8	10	12	14
---	---	---	---	----	----	----

Porównujemy $10 > 8$.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Szukany element może znajdować się tylko po prawej stronie, więc odrzucamy lewą część tablicy.

Krok 2

Rozpatrujemy prawy fragment tablicy. Środkowy element to 12.

10	12	14
----	----	----

Porównujemy $10 < 12$, więc szukany element znajduje się po lewej stronie.

Krok 3

Rozpatrujemy lewą część podtablicy, gdzie jedyny element to 10.

$10 = 10$ – element został znaleziony.

Wynik: szukany element **10** znajduje się w tablicy na **5** pozycji.

W każdym kroku liczba rozważanych elementów zmniejsza się o połowę. Dlatego liczba kroków wynosi w przybliżeniu: $\log_2 n$.

Złożoność czasowa:

- najlepszy przypadek: $O(1)$
- średni przypadek: $O(\log n)$
- najgorszy przypadek: $O(\log n)$

Ćwiczenia samodzielne

Zadanie 3.1. Algorytm sortowania bąbelkowego w wersji podstawowej

Zaimplementuj *algorytm sortowania bąbelkowego w wersji podstawowej* (bez optymalizacji) [*Bubble Sort*] dla tablicy liczb całkowitych. Program powinien umożliwiać wyświetlenie zawartości tablicy przed i po sortowaniu.

Sprawdź działanie algorytmu dla różnych przypadków:

- tablica posortowana rosnąco (optymistyczny),
- tablica w losowej kolejności (średni),
- tablica posortowana malejąco (pesymistyczny).

Dodatkowo program powinien zliczać:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- liczbę porównań,
- liczbę zamian elementów.

Zadanie 3.2. Zoptymalizowany algorytm sortowania bąbelkowego

Zaimplementuj *zoptymalizowaną wersję algorytmu sortowania bąbelkowego* (z wykorzystaniem flagi informującej o wystąpieniu zamiany elementów w danym przebiegu). Przetestuj działanie algorytmu dla danych odpowiadających przypadkowi optymistycznemu, średniemu oraz pesymistycznemu.

Program powinien dodatkowo zliczać:

- liczbę porównań,
- liczbę zamian elementów.

Porównaj uzyskane wyniki z wynikami otrzymanymi dla wersji podstawowej.

Zadanie 3.3. Algorytm sortowania przez wybór

Zaimplementuj *algorytm sortowania przez wybór* (*Selection Sort*) dla tablicy liczb całkowitych. W każdej iteracji należy wyszukać najmniejszy element w nieposortowanej części tablicy. Po zakończeniu każdej iteracji jeden element powinien znajdować się na swojej docelowej pozycji.

Program powinien dodatkowo zliczać:

- liczbę porównań,
- liczbę zamian elementów.

Zadanie 3.4. Algorytm sortowania przez wstawianie

Zaimplementuj *algorytm sortowania przez wstawianie* (*Insertion Sort*). Algorytm powinien budować posortowaną część tablicy od lewej strony, każdy kolejny element powinien być wstawiany na odpowiednią pozycję.

Porównaj wyniki uzyskane dla:

- a) tablicy w losowej kolejności,
- b) tablicy posortowanej malejąco.

Zadanie 3.5. Algorytm sortowania szybkiego

Zaimplementuj *algorytm sortowania szybkiego* (*Quicksort*) dla tablicy liczb całkowitych. Przetestuj działanie algorytmu.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 3.6. Algorytm wyszukiwania liniowego

Zaimplementuj *algorytm wyszukiwania liniowego* (*Linear Search*) dla tablicy liczb całkowitych. Program powinien zwracać indeks szukanego elementu lub informację o jego braku. Przetestuj działanie algorytmu dla danych odpowiadających przypadkowi optymistycznemu, średniemu oraz pesymistycznemu.

Zadanie 3.7. Algorytm wyszukiwania liniowego z wartownikiem

Zaimplementuj *algorytm wyszukiwania liniowego z wartownikiem* w tablicy liczb całkowitych. Program powinien zwracać indeks szukanego elementu lub informację o jego braku. Przetestuj działanie algorytmu dla danych odpowiadających przypadkowi optymistycznemu, średniemu oraz pesymistycznemu.

Zadanie 3.8. Algorytm wyszukiwania binarnego

Zaimplementuj *algorytm wyszukiwania binarnego* (*Binary Search*) w wersji iteracyjnej lub rekurencyjnej. Pamiętaj, że dane wejściowe powinny być posortowane rosnąco przed rozpoczęciem wyszukiwania. Porównaj liczbę porównań z wyszukiwaniem liniowym dla tej samej tablicy.

Pytania pomocnicze

- 1) Czym różni się przypadek optymistyczny, średni i pesymistyczny w analizie algorytmów sortowania?
- 2) Dlaczego w sortowaniu bąbelkowym bez optymalizacji liczba porównań jest stała?
- 3) Jakie warunki muszą być spełnione, aby możliwe było zastosowanie wyszukiwania binarnego?
- 4) Czym różni się przypadek optymistyczny, średni i pesymistyczny w analizie algorytmów wyszukiwania?
- 5) Na czym polega idea wyszukiwania liniowego z wartownikiem i jaki ma ona wpływ na liczbę operacji?
- 6) Jaka operacja jest dominująca w algorytmach wyszukiwania?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Podsumowanie

W ramach laboratorium zaprezentowano działanie wybranych algorytmów sortowania i wyszukiwania oraz przeanalizowano je w przypadkach optymistycznym, średnim i pesymistycznym. Szczegółowa analiza umożliwiła zrozumienie mechanizmu działania poszczególnych algorytmów.

Literatura

- [1] Drozdek A., *C++. Algorytmy i struktury danych*, Helion, Gliwice 2004. s. 413-443.
- [2] Wirth N., *Algorytmy + struktury danych = programy*, WNT, Warszawa, 2002. s. 73-82.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Materiały do przedmiotu:

Algorytmy i struktury danych

Laboratorium nr 4

Implementacja algorytmów rekurencyjnych

Autor:

dr hab. Adam Kiersztyn
mgr Krystyna Kiersztyn



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Spis treści

Wprowadzenie, tematyka zajęć	60
Cel ćwiczenia/laboratorium	61
Instrukcja laboratoryjna/ćwiczeniowa	61
Ćwiczenia samodzielne	76
Pytania pomocnicze	78
Podsumowanie	79
Literatura	79



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wprowadzenie, tematyka zajęć

Rekurencja jest techniką programowania, w której rozwiązanie problemu polega na jego sprowadzeniu do mniejszego fragmentu tego samego problemu. Funkcja realizująca takie podejście wywołuje samą siebie i nazywana jest **funkcją rekurencyjną**. Aby działała poprawnie, musi zawierać dwa podstawowe elementy:

- **przypadek bazowy** (warunek stopu) – punkt kończący dalsze wywołania i zwracający bezpośredni wynik,
- **wywołanie rekurencyjne** – moment, w którym funkcja wywołuje samą siebie, by rozwiązać mniejszą część problemu.

Podczas wykonywania funkcji rekurencyjnej program korzysta ze **stosu wywołań**, na którym odkładane są informacje o kolejnych wywołaniach funkcji. Po osiągnięciu warunku stopu program zaczyna „wracać” po stosie, a uzyskane wyniki przekazywane są do wcześniejszych wywołań. W ten sposób rekurencja umożliwia rozwiązywanie problemów poprzez ich podział na mniejsze fragmenty.

Rekurencja znajduje zastosowanie w wielu algorytmach, m.in. w algorytmach sortowania (MergeSort, HeapSort, QuickSort) i przeszukiwania (przeszukiwanie binarne, DFS), a także w operacjach na strukturach takich jak drzewa czy grafy. Stosuje się ją również w zagadnieniach matematycznych, np. przy obliczaniu silni, elementów ciągów (np. Fibonacciego) czy największego wspólnego dzielnika.

W sytuacjach, gdy te same podproblemy są rozwiązywane wielokrotnie, czysta rekurencja może być nieefektywna i prowadzić do znacznego wzrostu liczby wywołań funkcji. W takich przypadkach stosuje się **programowanie dynamiczne**, które polega (jak już wspomniano w laboratorium 2.) na zapamiętywaniu wcześniej obliczonych wyników i ponownym ich wykorzystywaniu zamiast ponownego przeliczania. Metoda ta znajduje zastosowanie w problemach posiadających optymalną strukturę podproblemów oraz nakładające się podproblemy. Programowanie dynamiczne może być realizowane w podejściu top-down (rekurencja z memoizacją) lub bottom-up (iteracyjne budowanie rozwiązania od najmniejszych podproblemów).

Szczególną formą jest **rekurencja ogonowa**, w której wywołanie funkcji jest ostatnią wykonywaną operacją. W takich przypadkach możliwa jest optymalizacja polegająca na ograniczeniu użycia stosu. W praktyce jednak optymalizacja ta zależy od języka programowania i kompilatora (nie wszystkie środowiska ją wspierają).

Proces przekształcania funkcji rekurencyjnej w równoważną wersję iteracyjną nazywa się **derekursywacją**. Polega on na zastąpieniu wywołań rekurencyjnych za



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



pomocą pętli, co zwykle zmniejsza zużycie pamięci i może poprawić wydajność, choć niekiedy kosztem czytelności i przejrzystości kodu.

Cel ćwiczenia/laboratorium

Po zrealizowaniu laboratorium student:

- rozumie zasadę działania rekurencji,
- potrafi implementować funkcje rekurencyjne w wybranym języku programowania,
- rozumie mechanizm stosu wywołań,
- zna różnice między rekurencją a iteracją i potrafi wskazać zalety oraz wady obu podejść,
- zna pojęcie rekurencji ogonowej i rozumie jej wpływ na optymalizację programów,
- potrafi przekształcić implementację rekurencyjną w równoważną wersję iteracyjną.
- rozumie zależność między rekurencją a programowaniem dynamicznym w kontekście efektywności obliczeń.

Instrukcja laboratoryjna/ćwiczeniowa

Przykład 4.1. Obliczanie silni

Silnia liczby naturalnej n jest definiowana wzorem

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1,$$

przy założeniu, że $0! = 1$.

W niniejszym przykładzie przedstawiono obliczania silni liczby naturalnej n z wykorzystaniem trzech podejść:

- klasycznej rekurencji,
- rekurencji ogonowej,
- podejścia iteracyjnego.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



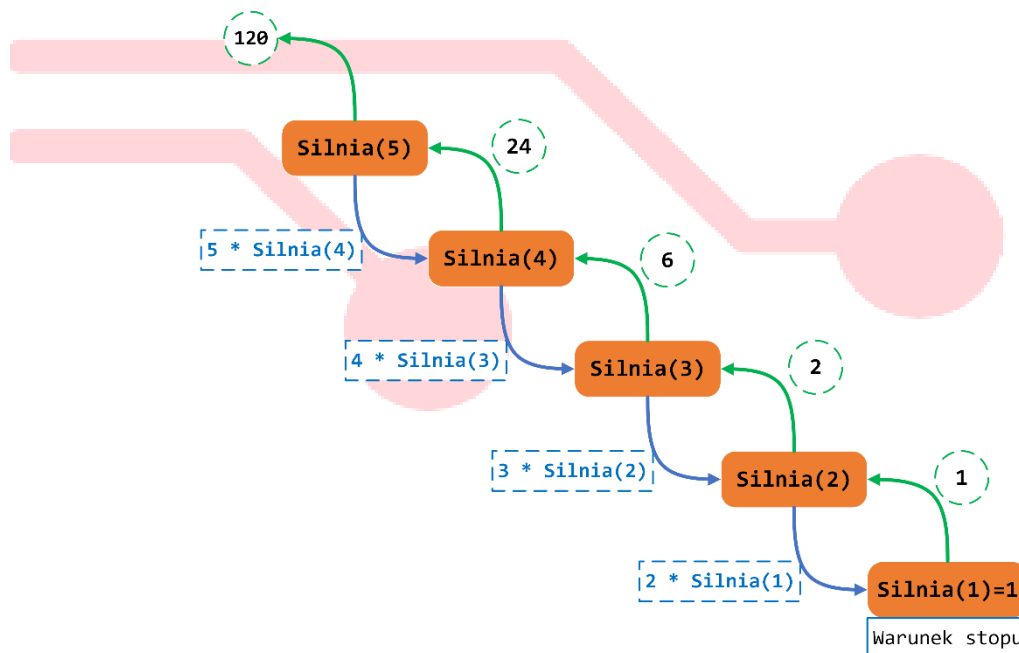
A. Wersja rekurencyjna

W podejściu rekurencyjnym funkcja $Silnia(n)$ wywołuje samą siebie, za każdym razem zmniejszając argument n o jeden, aż do osiągnięcia warunku stopu. Każde kolejne wywołanie odkładane jest na stos wywołań. Po osiągnięciu przypadku bazowego wykonywanie funkcji przebiega w odwrotnej kolejności, a uzyskany wynik jest przekazywany do wcześniejszych wywołań i mnożony przez aktualną wartość n .

Algorytm P.4.1.A. Silnia rekurencyjnie

```
Silnia(n)
{
    if n == 0 then
        return 1
    else
        return n * Silnia(n - 1)
}
```

Jest to podejście typu top-down, ponieważ obliczenia rozpoczynają się od problemu głównego n i są redukowane do coraz mniejszych podproblemów, aż do osiągnięcia przypadku bazowego. Rysunek 4 przedstawia przebieg działania funkcji rekurencyjnej $Silnia(n)$ dla parametru $n = 5$.



Rysunek 4 Przebieg wywołań metody rekurencyjnej $Silnia(n)$ dla $n = 5$ (źródło: opracowanie własne)



Fundusze Europejskie
dla Rozwoju Społecznego

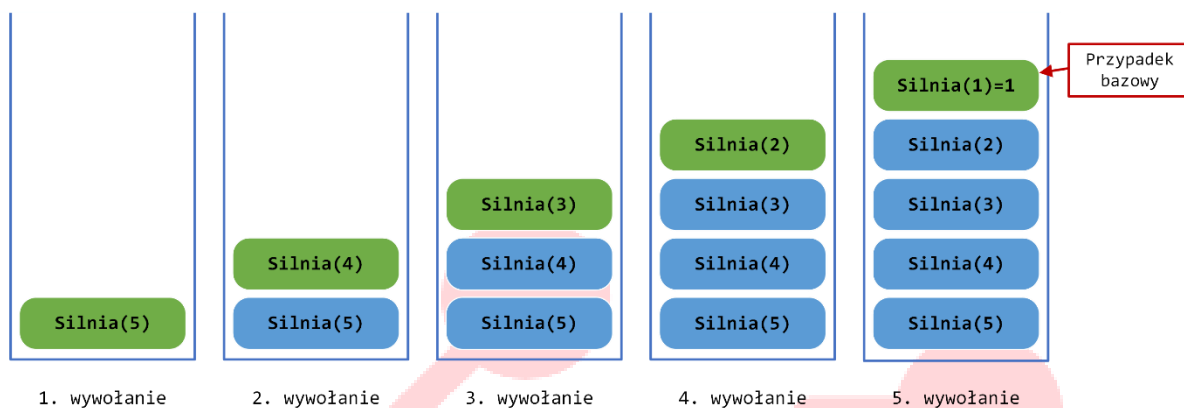


Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską

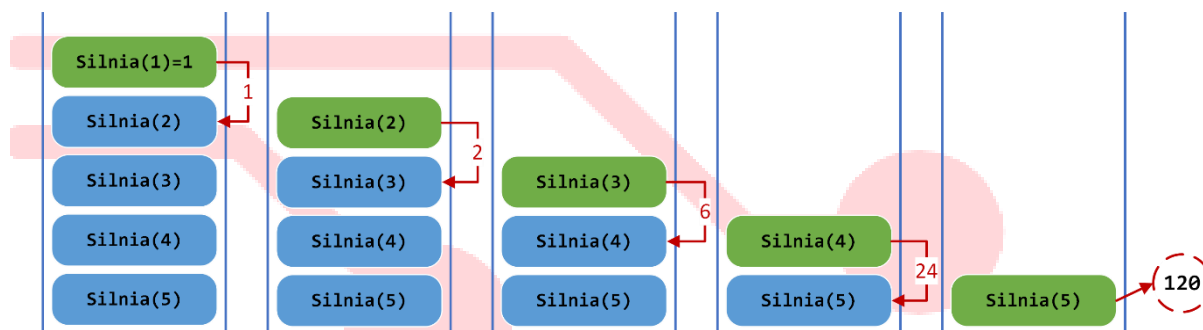


Podczas wywoływania funkcji rekurencyjnej kolejne wywołania są umieszczane na stosie wywołań (patrz Rysunek 5). Każde wywołanie przechowuje własne wartości parametrów oraz informacje potrzebne do kontynuowania obliczeń po zakończeniu wywołania podrzędnego. Proces ten trwa aż do osiągnięcia przypadku bazowego.



Rysunek 5 Stan stosu wywołań podczas „schodzenia” rekurencji dla funkcji $Silnia(n)$ dla $n = 5$. Głębokość stosu odpowiada liczbie aktywnych wywołań funkcji (źródło: opracowanie własne)

Po osiągnięciu warunku stopu wykonywanie funkcji przebiega w odwrotnej kolejności (patrz Rysunek 6). Kolejne wywołania są zdejmowane ze stosu, a zwracane wartości przekazywane do wcześniejszych wywołań.



Rysunek 6 Stan stosu wywołań podczas „powrotu” z rekurencji dla funkcji $Silnia(n)$, $n = 5$ (źródło: opracowanie własne)

B. Rekurencja ogonowa

W tym podejściu zastosowano rekurencję ogonową, w której oprócz argumentu n przekazywany jest dodatkowy parametr acc (o wartości początkowej 1), pełniący rolę akumulatora. Przechowuje on bieżący wynik obliczeń i jest aktualizowany przy każdym wywołaniu rekurencyjnym zgodnie ze wzorem $acc * n$.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



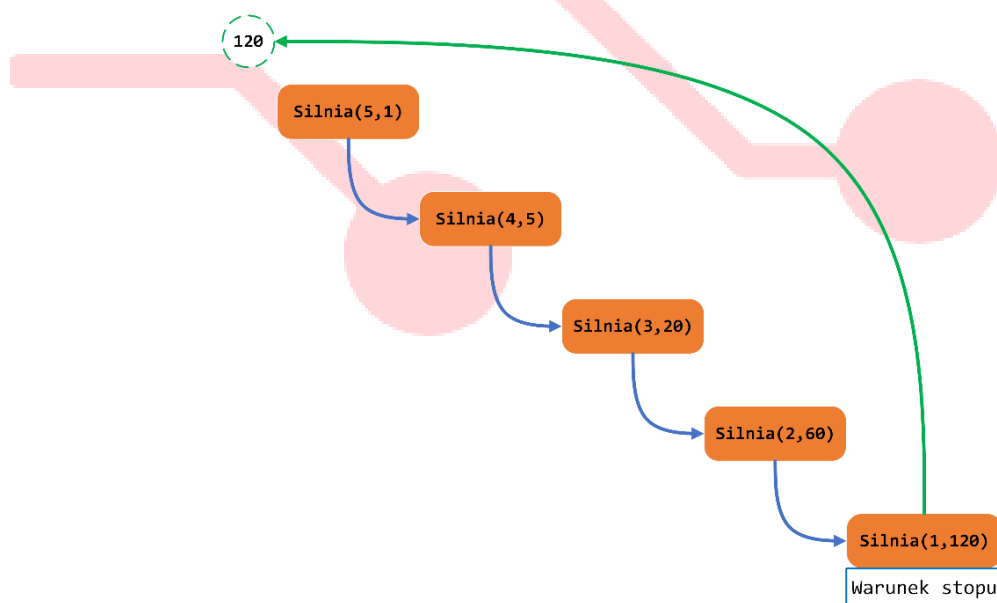
W odróżnieniu od klasycznej rekurencji, obliczenia wykonywane są „w trakcie schodzenia” do przypadku bazowego, a nie podczas powrotu ze stosu. W momencie osiągnięcia warunku stopu wynik końcowy jest już w pełni wyznaczony i zwracany bez wykonywania dodatkowych operacji. Dzięki temu wywołanie rekurencyjne stanowi ostatnią operację funkcji, co jest cechą charakterystyczną rekurencji ogonowej.

Uwaga! W teorii taka konstrukcja umożliwia optymalizację polegającą na ograniczeniu wykorzystania stosu wywołań (tzw. *tail call optimization*). W praktyce jednak nie wszystkie języki programowania i środowiska wykonawcze wspierają tę optymalizację, dlatego rekurencja ogonowa nie zawsze prowadzi do zmniejszenia zużycia pamięci.

Algorytm P.4.1.B. Silnia z rekurencją ogonową

```
SilniaOgonowa(n, acc = 1)
{
    if n == 0 then
        return acc
    else
        return SilniaOgonowa(n - 1, acc * n)
}
```

Rysunek 7 przedstawia przebieg działania funkcji `SilniaOgonowa(n, acc=1)` do obliczenia silni liczby 5.



Rysunek 7 Przebieg wywołań funkcji rekurencyjnej `SilniaOgonowa(n, acc=1)` dla $n = 5$ wykorzystującej rekurencję ogonową [źródło: opracowanie własne]



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



C. Wersja iteracyjna

W podejściu iteracyjnym obliczenie silni odbywa się za pomocą pętli (np. for lub while).

W podejściu iteracyjnym obliczanie silni realizowane jest z wykorzystaniem pętli. Stosowana jest zmienna pomocnicza przechowująca bieżący wynik obliczeń. W każdej iteracji pętli wynik jest aktualizowany poprzez mnożenie przez kolejną wartość z zakresu od 1 do n.

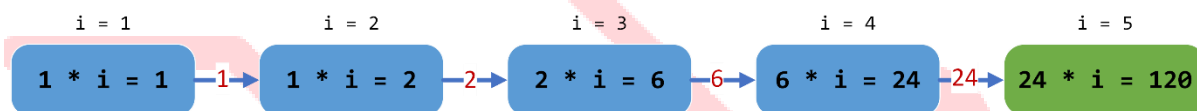
Algorytm P.4.1.C. Silnia iteracyjnie

```
SilniaIteracyjnie(n)
{
    wynik = 1

    for i = 1 to n do
        wynik = wynik * i

    return wynik
}
```

W przeciwieństwie do wersji rekurencyjnej nie są tworzone kolejne ramki stosu wywołań, co powoduje mniejsze zużycie pamięci. Algorytm wykonuje dokładnie n kroków mnożenia, a wynik jest obliczany sekwencyjnie w jednej pętli (patrz Rysunek 8).



Rysunek 8 Kolejne iteracje pętli for w funkcji SilniaIteracyjnie(n) dla n = 5 (źródło: opracowanie własne)

Przykład 4.2. Sortowanie przez scalanie – Merge Sort

Algorytm Merge Sort działa w dwóch etapach:

1. Dzielenie tablicy na mniejsze części,
2. Scalanie posortowanych fragmentów.

Algorytm P.4.2. Sortowanie przez scalanie

```
MergeSort(A, l, r)
{
    JEŻELI l < r
        mid = (l + r) / 2
        MergeSort(A, l, mid)           // lewa część
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
        MergeSort(A, mid + 1, r)    // prawa część
        Merge(A, l, mid, r)        // scalanie
    }

Merge(A, left, mid, right)
{
    n1 = mid - left + 1
    n2 = right - mid

    utwórz tablice L[0..n1-1]
    utwórz tablice R[0..n2-1]

    // kopiowanie danych
    DLA i od 0 do n1 - 1
        L[i] = A[left + i]
    DLA j od 0 do n2 - 1
        R[j] = A[mid + 1 + j]

    i = 0
    j = 0
    k = left

    // scalanie
    DOPÓKI i < n1 ORAZ j < n2
        JEŻELI L[i] ≤ R[j]
            A[k] = L[i]
            i = i + 1
        W PRZECIWNYM RAZIE
            A[k] = R[j]
            j = j + 1
        k = k + 1

    // pozostałe elementy L
    DOPÓKI i < n1
        A[k] = L[i]
        i = i + 1
        k = k + 1
    // pozostałe elementy R
    DOPÓKI j < n2
        A[k] = R[j]
        j = j + 1
        k = k + 1
}
```



Fundusze Europejskie
dla Rozwoju Społecznego

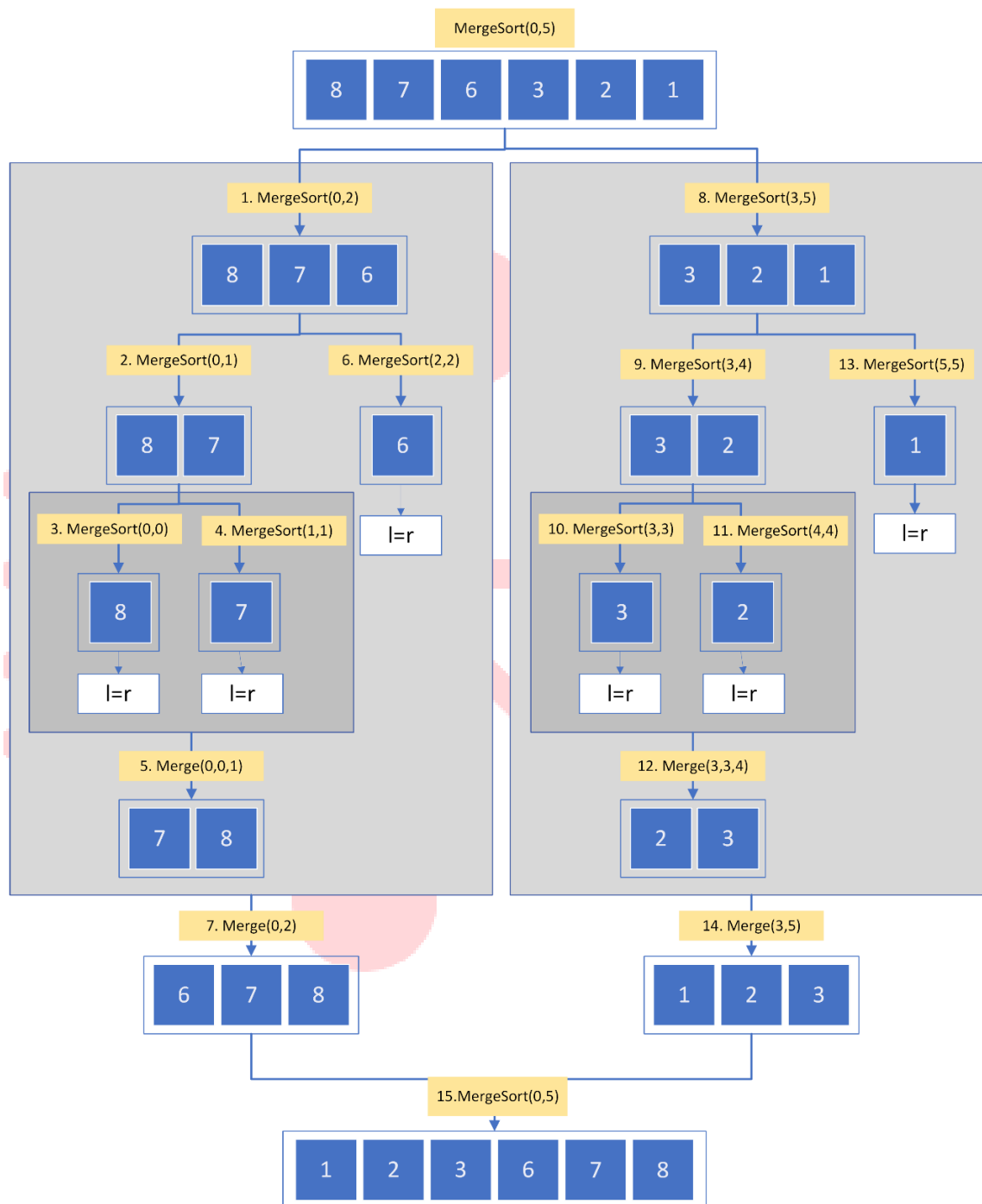


Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Aby lepiej zobrazować działanie algorytmu Merge Sort, rozważmy jego przebieg na przykładowej tablicy liczb całkowitych.



Rysunek 9 Przykładowy przebieg działania algorytmu Merge Sort na tablicy liczb całkowitych



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Złożoność czasowa: Algorytm Merge Sort ma złożoność czasową $O(n \log n)$ we wszystkich przypadkach, ponieważ na każdym poziomie rekurencji przetwarzane są wszystkie elementy, a liczba poziomów wynosi $\log n$.

Przykład 4.3. Heap Sort

Kopiec maksymalny (max-heap) to struktura danych reprezentowana np. za pomocą tablicy, w której dla każdego elementu o indeksie i :

- lewe dziecko ma indeks $2i + 1$,
- prawe dziecko ma indeks $2i + 2$,
- wartość w węźle jest większa lub równa wartości jego dzieci.

Algorytm P.4.3. Sortowanie przez kopcowanie

```
HeapSort(A, n)
{
    // budowanie kopca maksymalnego
    DLA i = n/2 - 1 DO 0 (malejąco)
        Heapify(A, n, i)

    // sortowanie
    DLA i = n - 1 DO 1 (malejąco)
    {
        zamień A[0] z A[i]
        Heapify(A, i, 0)
    }
}

Heapify(A, n, i)
{
    largest = i
    left = 2*i + 1
    right = 2*i + 2

    JEŻELI left < n ORAZ A[left] > A[largest]
        largest = left
    JEŻELI right < n ORAZ A[right] > A[largest]
        largest = right
    JEŻELI largest ≠ i
    {
        zamień A[i] z A[largest]
        Heapify(A, n, largest)
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

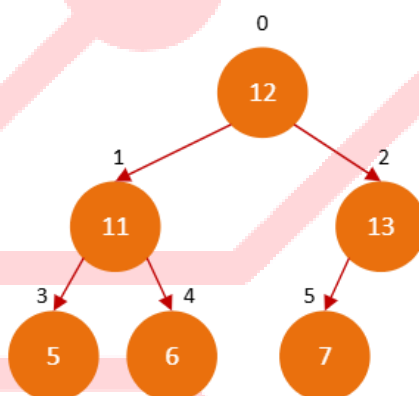
Dofinansowane przez
Unię Europejską



Aby lepiej zrozumieć działanie algorytmu Heap Sort, przeanalizujemy jego działanie krok po kroku na przykładowej tablicy liczb całkowitych. W pierwszym etapie z podanej tablicy zostanie zbudowany kopiec maksymalny, czyli taka struktura, w której każdy element jest większy lub równy swoim dzieciom, a największy element znajduje się na początku tablicy.

Następnie algorytm będzie wielokrotnie zamieniał pierwszy element (największy) z ostatnim elementem nieposortowanej części tablicy. Po każdej zamianie zostanie przywrócona własność kopca za pomocą procedury *Heapify*. Proces ten będzie powtarzany aż do całkowitego uporządkowania wszystkich elementów w tablicy.

Rozważmy tablicę $A = [12, 11, 13, 5, 6, 7]$. Reprezentację graficzną kopca przedstawia Rysunek 10.



Rysunek 10 Wizualizacja kopca

1. Budowanie kopca maksymalnego

Dla $n = 6$ zaczynamy od indeksu:

$$\frac{n}{2} - 1 = \frac{6}{2} - 1 = 2$$

Wywołujemy metodę $\text{Heapify}(A, n, i)$ dla $i = 2, i = 1$ oraz $i = 0$.

$\text{Heapify}(A, 6, 2)$

Analizujemy element $A[2] = 13$.

Dzieci:

- lewe: $2*2+1 = 5 - A[5] = 7$
- prawe: $2*2+2 = 6 - \text{brak}$.



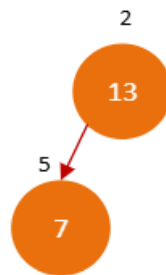
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską





Porównanie: $13 > 7$, więc nic nie zamieniamy.

Tablica ma postać:

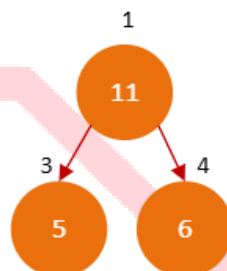
[12, 11, 13, 5, 6, 7]

Heapify(A, 6, 1)

Analizujemy element $A[1] = 11$.

Dzieci:

- lewe: 3 – $A[3] = 5$
- prawe: 4 – $A[4] = 6$



Porównanie:

- $11 > 5$
- $11 > 6$

Nic nie zmieniamy. Tablica ma postać:

[12, 11, 13, 5, 6, 7]

Heapify(A, 6, 0)

Analizujemy element $A[0] = 12$



Fundusze Europejskie
dla Rozwoju Społecznego



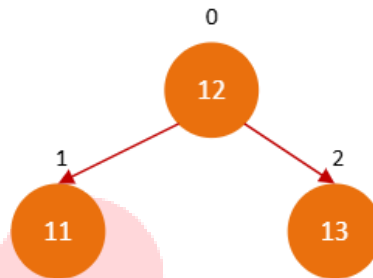
Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Dzieci:

- lewe: 1 – $A[1] = 11$
- prawe: 2 – $A[2] = 13$



Największy to 13, więc zamieniamy **$A[0]$ z $A[2]$** .

Tablica po zamianie ma postać:

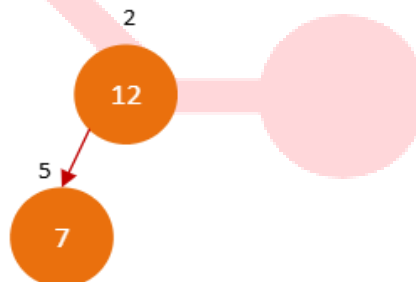
[13, 11, 12, 5, 6, 7]

Teraz wywołujemy **Heapify(A, 6, 2)**:

Analizowany element to **$A[2] = 12$**

Dzieci:

- lewe: 5 – 7
- prawe: 6 – poza tablicą



Porównanie: $12 > 7$. Koniec.

Kopiec maksymalny jest gotowy:

[13, 11, 12, 5, 6, 7]



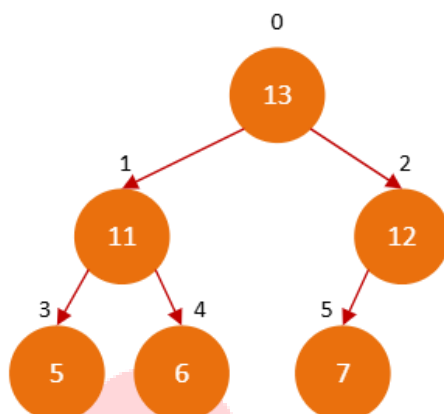
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską





2. Sortowanie

Teraz największy element jest na początku, więc zamieniamy go z ostatnim elementem kopca.

Iteracja 1

Wykonujemy zamianę największego elementu (korzenia) z ostatnim elementem, tzn. $A[0]$ z $A[5]$:

[7, 11, 12, 5, 6, 13]

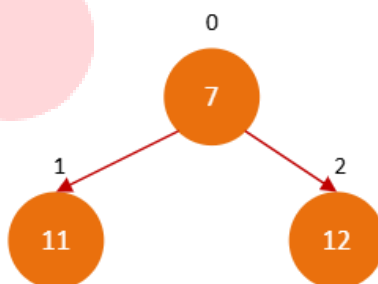
13 jest już na właściwej pozycji.

Przywracamy własność kopca dla 5 pierwszych elementów:

Heapify(A, 5, 0)

Analizowany element: 7.

Dzieci: 11, 12



Największy to **12**, więc wykonujemy zamianę $A[0]$ z $A[2]$:

[12, 11, 7, 5, 6, 13]



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wywołujemy **Heapify(A, 5, 2)**:

- lewe dziecko z indeksem 5 jest poza analizowanym zakresem (już uporządkowane), koniec iteracji.

Iteracja 2

Wykonujemy zamianę największego elementu (korzenia) z ostatnim możliwym elementem, tzn. **A[0]** z **A[4]**:

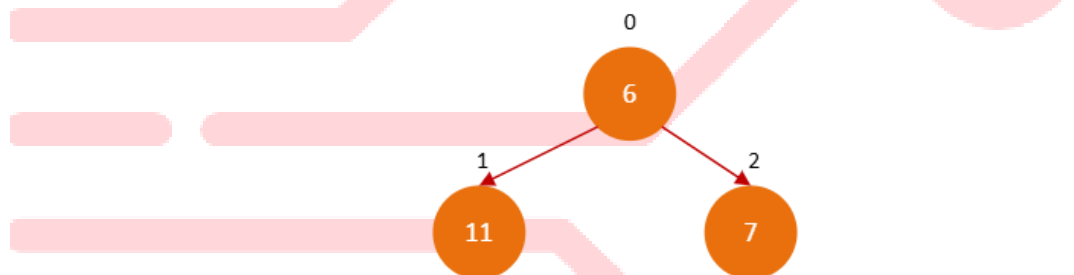
[6, 11, 7, 5, 12, 13]

Przywracamy własność kopca dla 4 pierwszych elementów:

Heapify(A, 4, 0)

Analizowany element: 6

Dzieci: 11, 7.



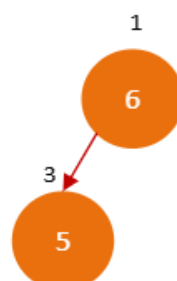
Największy element ma wartość 11 (lewe dziecko), więc wykonujemy zamianę **A[0]** z **A[1]**:

[11, 6, 7, 5, 12, 13]

Wywołujemy **Heapify(A, 4, 1)**:

Analizowany element: 6.

Dziecko: 5.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



$6 > 5$, więc nic nie zmieniamy.

Iteracja 3

Wykonujemy zamianę największego elementu (korzenia) z ostatnim możliwym elementem, tzn. **A[0]** z **A[3]**:

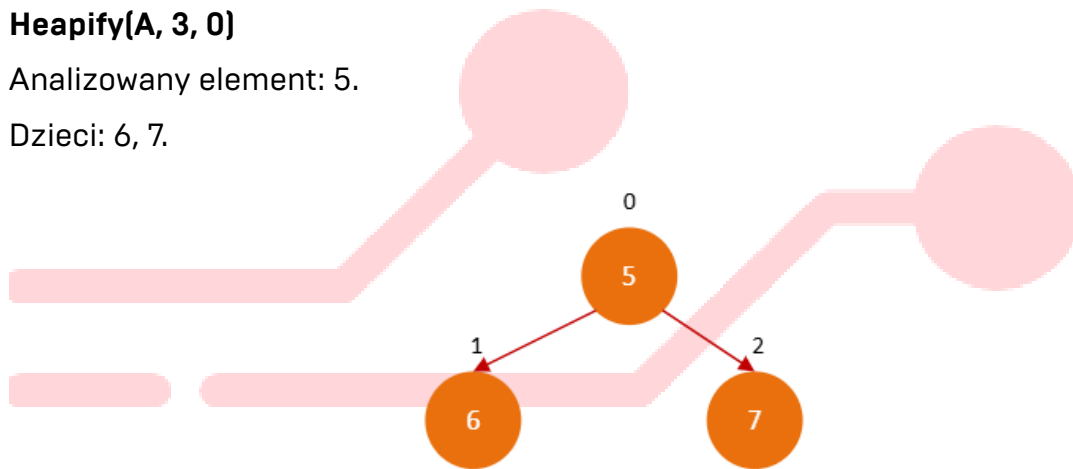
[5, 6, 7, **11**, 12, 13]

Przywracamy własność kopca dla 3 pierwszych elementów:

Heapify(A, 3, 0)

Analizowany element: 5.

Dzieci: 6, 7.



Największy element to 7 [prawe dziecko], więc wykonujemy zamianę:

[**7**, 6, **5**, 11, 12, 13]

Wywołujemy **Heapify(A, 3, 2)**:

- lewe dziecko z indeksem 5 jest poza analizowanym zakresem (już uporządkowane).

Iteracja 4

Wykonujemy zamianę największego elementu (korzenia) z ostatnim możliwym elementem, tzn. **A[0]** z **A[2]**:

[**5**, 6, **7**, 11, 12, 13]

Przywracamy własność kopca dla 2 pierwszych elementów:

Heapify(A, 2, 0)

Analizowany element: 5

Dziecko: 6



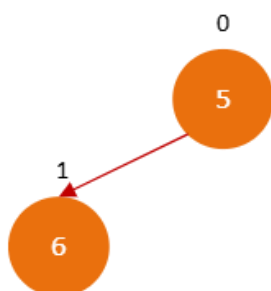
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską





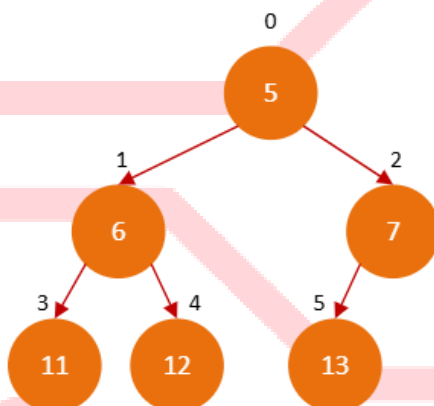
6 > 5, zatem wykonujemy zamianę **A[1]** z **A[0]**:

[**6**, 5, 7, 11, 12, 13]

Iteracja 5

Wykonujemy zamianę największego elementu (korzenia) z ostatnim możliwym elementem, tzn. **A[0]** z **A[1]**:

[**5**, 6, 7, 11, 12, 13]



Tablica jest sortowana.

Złożoność czasowa: Heap Sort ma złożoność $O(n \log n)$ we wszystkich przypadkach.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Ćwiczenia samodzielne

Zadanie 4.1. Ciąg Lucasa

Ciąg Lucasa zdefiniowany jest następująco:

$$L_n = \begin{cases} 2 & \text{dla } n = 0 \\ 1 & \text{dla } n = 1 \\ L_{n-1} + L_{n-2} & \text{dla } n > 1 \end{cases}$$

Zaimplementuj trzy wersje obliczania L_n :

- rekurencyjną (naiwną, bez zapamiętywania wyników),
- wykorzystującą programowanie dynamiczne z memoizacją,
- wykorzystującą programowanie dynamiczne w podejściu bottom-up (iteracyjną).

Określ złożoność czasową i pamięciową każdego podejścia. Oszacuj, ile wywołań funkcji rekurencyjnej zostanie wykonanych w wersji a) dla $n = 5$.

Zaimplementuj również obliczanie L_n w postaci rekurencji ogonowej, wykorzystując dodatkowe parametry przekazywane w wywołaniu funkcji.

Zadanie 4.2. Ciąg Tetranacciego

Ciąg Tetranacciego zdefiniowany jest następująco:

$$T_n = \begin{cases} 0 & \text{dla } n = 0 \\ 0 & \text{dla } n = 1 \\ 0 & \text{dla } n = 2 \\ 1 & \text{dla } n = 3 \\ T_{n-1} + T_{n-2} + T_{n-3} + T_{n-4} & \text{dla } n > 3 \end{cases}$$

Zaimplementuj trzy wersje obliczania T_n :

- rekurencyjną (naiwną, bez zapamiętywania wyników),
- wykorzystującą programowanie dynamiczne z memoizacją,
- wykorzystującą programowanie dynamiczne w podejściu bottom-up (iteracyjną).

Określ złożoność czasową i pamięciową każdego podejścia.

Zaimplementuj również obliczanie T_n w postaci rekurencji ogonowej, wykorzystując dodatkowe parametry przekazywane w wywołaniu funkcji.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 4.3. Największy wspólny dzielnik (NWD)

Największy wspólny dzielnik (NWD) dwóch liczb całkowitych a i b można obliczyć za pomocą algorytmu Euklidesa, wykorzystującego zależność:

$$\text{NWD}(a, b) = \text{NWD}(b, a \bmod b)$$

przy warunku $\text{NWD}(a, 0) = a$.

Zaimplementuj algorytm w wersji:

- a) rekurencyjnej,
- b) iteracyjnej.

Zadanie 4.4. Trójkąt Pascala

Trójkąt Pascala jest zdefiniowany rekurencyjnie w sposób następujący:

$$C(n, 0) = 1,$$

$$C(n, n) = 1,$$

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \text{ dla } 0 < k < n,$$

gdzie $C(n, k) = \binom{n}{k}$ oznacza element w n -tym wierszu i k -tej kolumnie.

Napisz program generujący i wyświetlający pierwsze n wierszy trójkąta Pascala w dwóch wersjach:

- a) rekurencyjnej (obliczanie wartości $C(n, k)$ z definicji rekurencyjnej),
- b) wersję iteracyjną.

Przetestuj działanie programu dla kilku wybranych wartości n .

Zadanie 4.5. Sortowanie przez scalanie – Merge Sort

Algorytm Merge Sort jest przykładem metody dziel i zwyciężaj. Polega na podziale tablicy na dwie części, rekurencyjnym posortowaniu każdej z nich, a następnie scaleniu dwóch uporządkowanych fragmentów w jedną posortowaną całość.

Zaimplementuj algorytm Merge Sort dla tablicy liczb całkowitych. Implementacja powinna zawierać metodę główną

```
MergeSort(int[] tab, int l, int r)
```

realizującą podział i rekurencyjne sortowanie oraz metodę pomocniczą

```
Merge(int[] tab, int left, int mid, int right)
```

odpowiedzialną za scalanie dwóch posortowanych części tablicy.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 4.6. Sortowanie przez kopcowanie – Heap Sort

Zaimplementuj algorytm Heap Sort dla tablicy liczb całkowitych. W rozwiązaniu należy wyodrębnić metodę odpowiedzialną za przywracanie własności kopca (heapify).

Zadanie 4.7. Wieże Hanoi

Problem Wież Hanoi polega na przeniesieniu n krążków z pręta A na pręt C, wykorzystując pręt pomocniczy B. Algorytm rekurencyjny składa się z następujących kroków:

- 1) Przenieś (rekurencyjnie) $n - 1$ krążków z pręta A na B używając pręta C,
- 2) Przenieś jeden krążek z pręta A na C,
- 3) Przenieś (rekurencyjnie) $n - 1$ krążków z pręta B na C posługując się prętem A.

Zaimplementuj rekurencyjną metodę `Hanoi(int n, char A, char B, char C)`, która wypisuje kolejne ruchy rozwiązania problemu (w postaci np. A -> C). Wyznacz liczbę wykonanych ruchów dla $n = 3, 4, 5, 6, 7$.

Pytania pomocnicze

- 1) Na czym polega mechanizm rekurencji?
- 2) Czym jest warunek stopu i jaką pełni rolę w rekurencji?
- 3) Jak zmienia się stan stosu wywołań podczas wykonywania funkcji rekurencyjnej?
- 4) Jakie są korzyści oraz ograniczenia z używania rekurencji?
- 5) Na czym polega rekurencja ogonowa?
- 6) Jakie są kluczowe różnice w sposobie działania algorytmu zapisanego rekurencyjnie i iteracyjnie?
- 7) Czy każdą implementację rekurencyjną można zastąpić konstrukcją iteracyjną?
- 8) Z czego wynika większe zużycie pamięci w rozwiązaniach rekurencyjnych w porównaniu z iteracyjnymi?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Podsumowanie

W ramach laboratorium zaprezentowano zasady działania rekurencji. Przedstawiono pojęcie rekurencji ogonowej. Przedstawiono również pojęcie rekurencji ogonowej. W dalszej części porównano implementacje rekurencyjne, iteracyjne oraz wykorzystujące programowanie dynamiczne, co pozwoliło przeanalizować różne sposoby realizacji algorytmów pod względem czytelności, złożoności czasowej i zużycia pamięci.

Literatura

- [1] Drozdek A., *C++. Algorytmy i struktury danych*, Helion, Gliwice 2004. s. 161-177, 440-443.
- [2] Wirth N., *Algorytmy + struktury danych = programy*, WNT, Warszawa, 2002. s. 145-151.
- [3] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C., *Wprowadzenie do algorytmów*, PWN, Warszawa, 2023. s. 149-162.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Materiały do przedmiotu:

Algorytmy i struktury danych

Laboratorium nr 5

Implementacja wybranych struktur danych

Autorzy:

dr hab. Adam Kiersztyn
mgr Krystyna Kiersztyn



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Spis treści

Wprowadzenie, tematyka zajęć	82
Cel ćwiczenia/laboratorium	83
Instrukcja laboratoryjna/ćwiczeniowa	83
Ćwiczenia samodzielne	101
Pytania pomocnicze	104
Podsumowanie	104
Literatura	105



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Struktury danych stanowią podstawowy element projektowania algorytmów i systemów informatycznych. Każdy program operuje na danych – przechowuje je, modyfikuje, przeszukuje i przetwarza. Sposób ich organizacji w pamięci komputera ma bezpośredni wpływ na wydajność działania aplikacji oraz złożoność implementowanych algorytmów. Właściwy dobór struktury danych pozwala znacząco skrócić czas wykonywania operacji oraz zmniejszyć zużycie zasobów systemowych.

Lista jest strukturą liniową przechowującą uporządkowaną sekwencję elementów. Elementy w liście posiadają określoną kolejność, a dostęp do nich może odbywać się poprzez indeks lub poprzez przechodzenie od jednego elementu do kolejnego.

Wyróżnia się listy **oparte na tablicach** (zapewniają szybki dostęp do elementu o danym indeksie) oraz **listy wiązane** (umożliwiają sprawne wstawianie i usuwanie elementów poprzez modyfikację połączeń między węzłami). Listy są strukturami ogólnego przeznaczenia i stanowią podstawę wielu innych bardziej złożonych struktur danych.

Stos jest strukturą danych działającą zgodnie z zasadą **LIFO (Last In, First Out)** – ostatni element dodany do struktury jest usuwany jako pierwszy. Dostęp do danych możliwy jest wyłącznie z jednego końca, zwanego szczytem stosu. Stos znajduje zastosowanie m.in. w obsłudze wywołań funkcji (stos wywołań), sprawdzaniu poprawności nawiasów w wyrażeniach, przetwarzaniu wyrażeń matematycznych, algorytmach przeszukiwania (DFS), mechanizmie cofania operacji (Undo).

Kolejka działa według zasady **FIFO (First In, First Out)** – pierwszy element dodany do struktury jest usuwany jako pierwszy. Elementy dodawane są na jednym końcu, a usuwane z drugiego. Kolejki wykorzystywane są m.in. symulacjach systemów obsługi (np. klienci w banku), planowaniu zadań, algorytmach przeszukiwania grafów (BFS), buforowaniu danych, systemach kolejkowania żądań (np. serwery).

Słownik (tablica asocjacyjna) przechowuje dane w postaci par **klucz–wartość**. Każdemu unikalnemu kluczowi przypisana jest jedna wartość. Struktura ta umożliwia szybkie wyszukiwanie danych na podstawie klucza. W praktyce słowniki często implementowane są przy użyciu mechanizmu haszowania lub struktur drzewiastych. Pozwalają one na efektywne operacje wyszukiwania, dodawania i usuwania elementów. Słowniki znajdują zastosowanie wszędzie tam, gdzie zachodzi potrzeba odwzorowania jednej informacji na inną, np. identyfikator → obiekt.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Cel ćwiczenia/laboratorium

Po zakończeniu laboratorium student:

- rozumie różnice pomiędzy podstawowymi strukturami danych,
- potrafi scharakteryzować podstawowe operacje dla listy, stosu, kolejki,
- rozumie zasady działania struktur LIFO i FIFO,
- potrafi wskazać typowe zastosowania poszczególnych struktur danych,
- potrafi wykorzystać poznane struktury do rozwiązywania prostych problemów algorytmicznych.

Instrukcja laboratoryjna/ćwiczeniowa

Przykład 5.1. Lista – implementacja wbudowana

W niniejszym przykładzie przedstawiono wykorzystanie klasy `List<T>` z przestrzeni nazw `System.Collections.Generic`.

Struktura `List<T>` jest dynamiczną tablicą – jej rozmiar zwiększa się w miarę dodawania elementów. Klasa `List<T>` udostępnia m.in.:

- `Add(element)` – dodanie elementu na koniec,
- `Insert(index, element)` – wstawienie elementu na określoną pozycję,
- `Remove(element)` – usunięcie pierwszego wystąpienia elementu,
- `RemoveAt(index)` – usunięcie elementu o podanym indeksie,
- `Contains(element)` – sprawdzenie, czy element istnieje,
- `IndexOf(element)` – zwrócenie indeksu elementu,
- `Count` – liczba elementów.

Program demonstruje podstawowe operacje na liście (lista ocen).



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        List<int> oceny = new List<int>();

        // Dodawanie ocen
        oceny.Add(4);
        oceny.Add(5);
        oceny.Add(3);

        Console.WriteLine("Lista ocen:");
        PrintList(oceny);

        // Wstawienie oceny na pozycję 1
        oceny.Insert(1, 2);
        Console.WriteLine("\nPo wstawieniu oceny 2 na indeks 1:");
        PrintList(oceny);

        // Usunięcie oceny 3
        oceny.Remove(3);
        Console.WriteLine("\nPo usunięciu oceny 3:");
        PrintList(oceny);

        // Sprawdzenie czy ocena 5 istnieje
        Console.WriteLine("\nCzy lista zawiera ocenę 5? "
            + oceny.Contains(5));
        // Wyszukanie indeksu oceny 4
        Console.WriteLine("Indeks oceny 4: " + oceny.IndexOf(4));

        Console.WriteLine("\nLiczba ocen: " + oceny.Count);
    }

    static void PrintList(List<int> list)
    {
        foreach (int element in list)
            Console.Write(element + " ");
        Console.WriteLine();
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykład 5.2. Lista jednokierunkowa

Lista jednokierunkowa (*singly linked list*) jest dynamiczną strukturą danych. Składa się z elementów zwanych węzłami (*nodes*), gdzie każdy węzeł przechowuje dane oraz odwołanie do następnego elementu w strukturze. Ostatni element listy wskazuje na brak następnego węzła (*null*).

Węzeł w liście jednokierunkowej zawiera:

- przechowywaną wartość,
- odwołanie do elementu następnego.

Dostęp do elementów odbywa się sekwencyjnie, tzn. aby dotrzeć do wybranego węzła, należy przejść przez wszystkie poprzedzające go elementy, rozpoczynając od pierwszego węzła (*head*). Oznacza to, że operacje wyszukiwania mają złożoność liniową względem liczby elementów.

Podstawowe operacje wykonywane na liście jednokierunkowej obejmują dodawanie elementów (na początek lub koniec), usuwanie elementów oraz przeszukiwanie struktury.

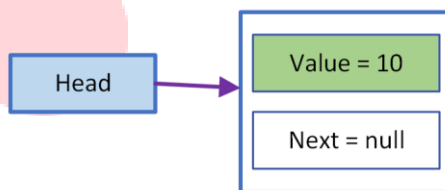
Rozważmy listę jednokierunkową przechowującą liczby całkowite. Jak już wcześniej wspomniano, każdy węzeł zawiera wartość oraz odwołanie do następnego elementu.

Po utworzeniu lista jest pusta, i referencja *head* wskazuje na *null*.

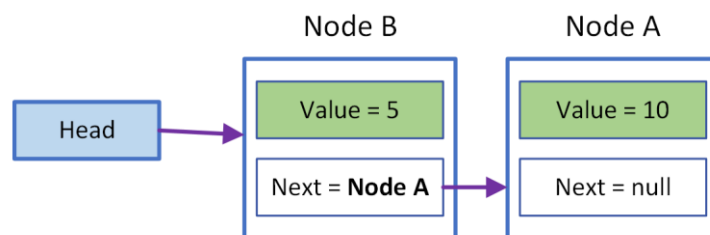
Dodawanie elementu na początek listy

Dodajemy element o wartości 10.

Nowy element staje się pierwszym węzłem listy, *head* wskazuje właśnie na ten węzeł.



Dodajemy kolejny element z wartością 5 na początek listy.

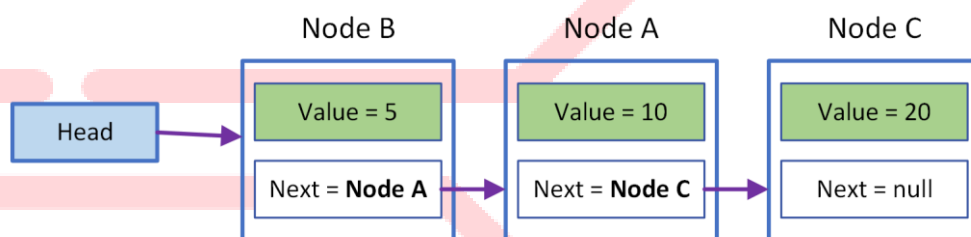


Operacja ta polega na:

- utworzeniu nowego węzła (Node B),
- ustawieniu jego wskaźnika na dotychczasowy pierwszy element,
- przesunięciu wskaźnika Head na nowy węzeł.

Dodawanie elementu na koniec listy

Dodajemy element z wartością 20 na koniec listy. Aby to zrobić, należy przejść przez wszystkie elementy do ostatniego węzła.



Operacja wymaga przejścia przez całą listę.

Wyszukiwanie elementu

Sprawdzamy, czy lista zawiera wartość 10. Przeszukujemy kolejne węzły od początku, aż do napotkania szukanej wartości. W tym przypadku nie ma bezpośredniego dostępu do elementu o danym indeksie.

Usuwanie elementu (Remove)

Usuujemy element o wartości 10.

Aby to zrobić, należy:

- znaleźć węzeł poprzedzający usuwany element,
- zmienić jego wskaźnik tak, aby pomijał usuwany węzeł.

Stan listy po usunięciu:



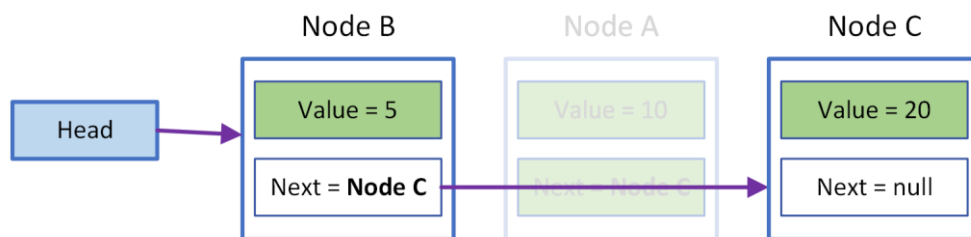
Fundusze Europejskie
dla Rozwoju Społecznego



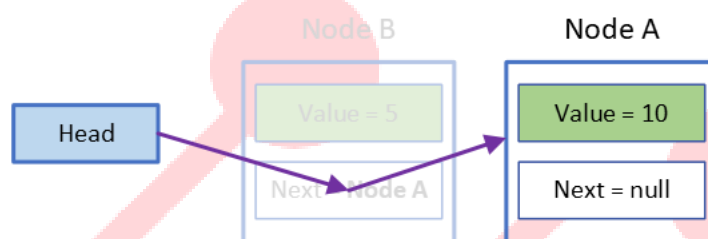
Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską





Jeżeli usuwany jest pierwszy element, wystarczy przesunąć wskaźnik head na kolejny element (używając referencji Next).



Najważniejsze cechy listy jednokierunkowej:

- elementy nie są przechowywane w sposób ciągły w pamięci,
- każdy węzeł zna tylko swojego następnika,
- brak dostępu przez indeks w czasie stałym,
- szybkie dodawanie na początek $O(1)$,
- operacje wymagające przeszukiwania mają złożoność $O(n)$.

Przykład 5.3. Lista dwukierunkowa

Lista dwukierunkowa (double linked list) jest dynamiczną strukturą danych służącą do przechowywania uporządkowanej sekwencji elementów. W przeciwieństwie do struktur opartych na tablicy, elementy nie muszą zajmować ciągłego obszaru pamięci. Każdy element listy (tzw. węzeł) zawiera:

- przechowywaną wartość,
- odwołanie do elementu poprzedniego,
- odwołanie do elementu następnego.

Dzięki temu możliwe jest przemieszczanie się po liście zarówno w kierunku od początku do końca, jak i w kierunku przeciwnym. Lista przechowuje dodatkowo informację o pierwszym (Head) i ostatnim elemencie (Tail).



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



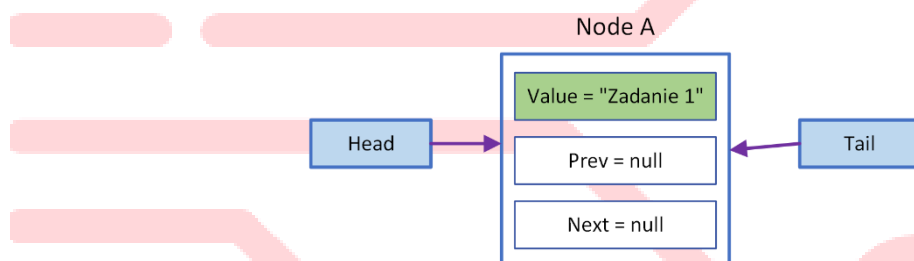
Jedną z najważniejszych cech listy dwukierunkowej jest możliwość wstawiania i usuwania elementów w dowolnym miejscu struktury bez konieczności przesuwania pozostałych elementów, co odróżnia ją od struktur sekwencyjnych opartych na tablicy.

Rozważmy listę dwukierunkową przechowującą zadania. Przeanalizujemy podstawowe operacje na liście dwukierunkowej:

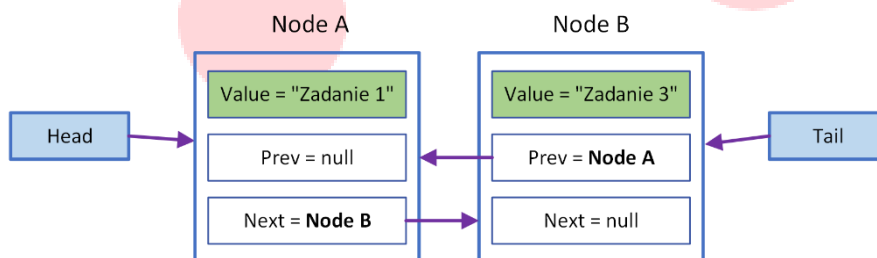
- dodawanie elementu na początek i na koniec listy,
- wyszukiwanie elementu,
- wstawianie elementu w określonym miejscu,
- usuwanie elementu,
- sprawdzanie liczby elementów w liście.

Dodawanie elementów na koniec listy

Dodanie elementu na koniec listy powoduje dołączenie go jako ostatniego w strukturze. Najpierw dodany zostaje element „Zadanie 1”. Ponieważ lista zawiera tylko jeden element, jest on jednocześnie pierwszym i ostatnim elementem.



Następnie na koniec listy dodany zostaje element „Zadanie 3”. Lista przyjmuje postać:



Element „Zadanie 3” staje się nowym ostatnim elementem listy, a oba elementy zostają ze sobą odpowiednio powiązane. Referencja Next pierwszego węzła wskazuje na drugi węzeł, a Prev drugiego węzła wskazuje na pierwszy węzeł.



Fundusze Europejskie
dla Rozwoju Społecznego



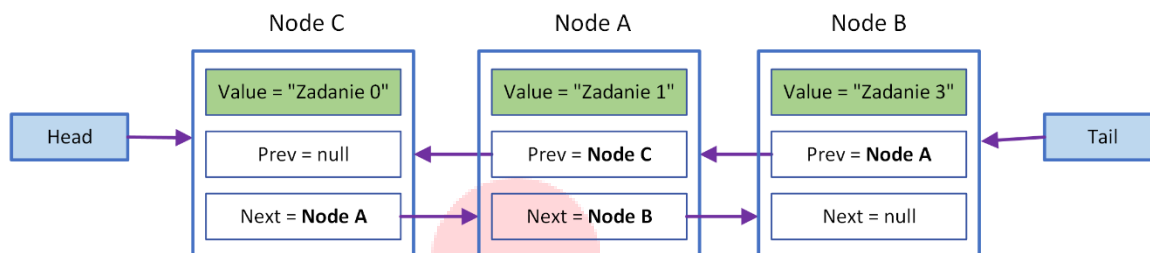
Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Dodawanie elementów na początek listy

Dodanie elementu na początek powoduje wstawienie go przed dotychczasowym pierwszym elementem. W ten sposób dodany zostaje element „Zadanie 0”. Lista przyjmuje postać:



„Zadanie 0” staje się nowym pierwszym elementem listy. Powiązania ponownie są aktualizowane.

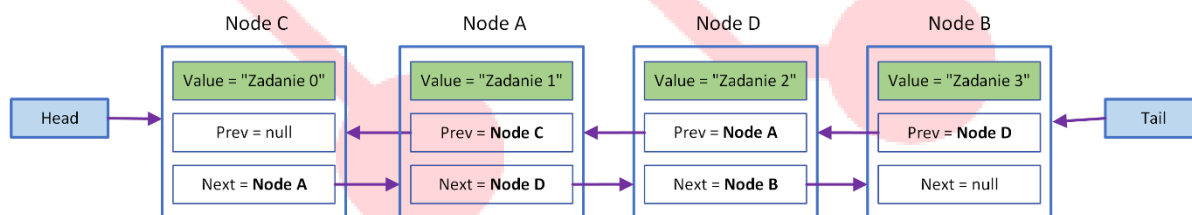
Wyszukiwanie elementu

Wyszukiwanie polega na przeglądaniu elementów od początku listy aż do momentu odnalezienia elementu o wskazanej wartości. Jeżeli szukany element nie występuje w liście, zwracana jest np. wartość null.

Wstawienie elementu w określonym miejscu

Możliwe jest wstawienie nowego elementu bezpośrednio po wskazany węzeł. W ten sposób po węźle zawierającym „Zadanie 1” dodany zostaje element „Zadanie 2”.

Lista po tej operacji ma postać (dodano Node D):

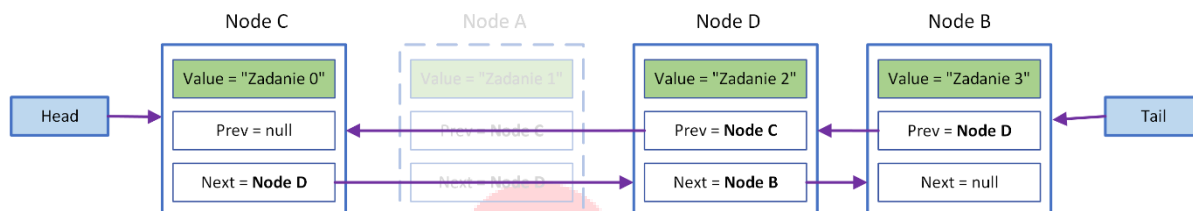


Nowy element zostaje włączony pomiędzy „Zadanie 1” i „Zadanie 3”, a powiązania między sąsiednimi węzłami zostają odpowiednio zaktualizowane.

Analogicznie możliwe jest wstawienie elementu przed wskazanym elementem.

Usuwanie elementu

Usunięcie elementu polega na odłączeniu go od listy oraz połączeniu jego sąsiadów. W przedstawionym przykładzie usunięty zostaje element „Zadanie 1”. Lista po usunięciu elementu ma postać:



Element „Zadanie 1” zostaje usunięty, a elementy „Zadanie 0” i „Zadanie 2” zostają bezpośrednio połączone.

Przykład 5.4. Lista dwukierunkowa – implementacja wbudowana

W niniejszym przykładzie przedstawiono wykorzystanie klasy `LinkedList<T>` z przestrzeni nazw `System.Collections.Generic`.

`LinkedList<T>` jest implementacją listy dwukierunkowej (doubly linked list). Każdy węzeł (Node) przechowuje następujące informacje:

- wartość (Value),
- referencję do poprzedniego elementu (Previous),
- referencję do następnego elementu (Next).

Podstawowe metody i właściwości `LinkedList`:

- `AddFirst(element)` – dodanie na początek,
- `AddLast(element)` – dodanie na koniec,
- `AddBefore(node, element)` – wstawienie przed wskazanym węzłem,
- `AddAfter(node, element)` – wstawienie po wskazanym węźle,
- `Remove(element)` – usunięcie pierwszego wystąpienia,
- `Remove(node)` – usunięcie wskazanego węzła,
- `Find(element)` – wyszukanie węzła,
- `First`, `Last` – dostęp do pierwszego i ostatniego elementu,
- `Count` – liczba elementów.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Poniżej przedstawiono implementację listy zadań z przykładu 5.6, ilustrującą zastosowanie metod klasy `LinkedList` do dodawania, wyszukiwania, wstawiania oraz usuwania elementów.

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        LinkedList<string> zadania = new LinkedList<string>();
        zadania.AddLast("Zadanie 1");
        zadania.AddLast("Zadanie 3");
        zadania.AddFirst("Zadanie 0");

        Console.WriteLine("Lista zadań:");
        PrintList(zadania);

        // Wyszukanie konkretnego węzła
        LinkedListNode<string> node = zadania.Find("Zadanie 1");

        // Wstawienie elementu po wskazanym węźle
        if (node != null)
            zadania.AddAfter(node, "Zadanie 2");

        Console.WriteLine("\nLista po wstawieniu nowego zadania:");
        PrintList(zadania);

        zadania.Remove("Zadanie 3"); // Usunięcie elementu

        Console.WriteLine("\nLista po usunięciu Zadanie 3:");
        PrintList(zadania);
        Console.WriteLine("\nLiczba elementów: " + zadania.Count);
    }

    static void PrintList(LinkedList<string> list)
    {
        foreach (var element in list)
            Console.Write(element + " | ");
        Console.WriteLine();
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



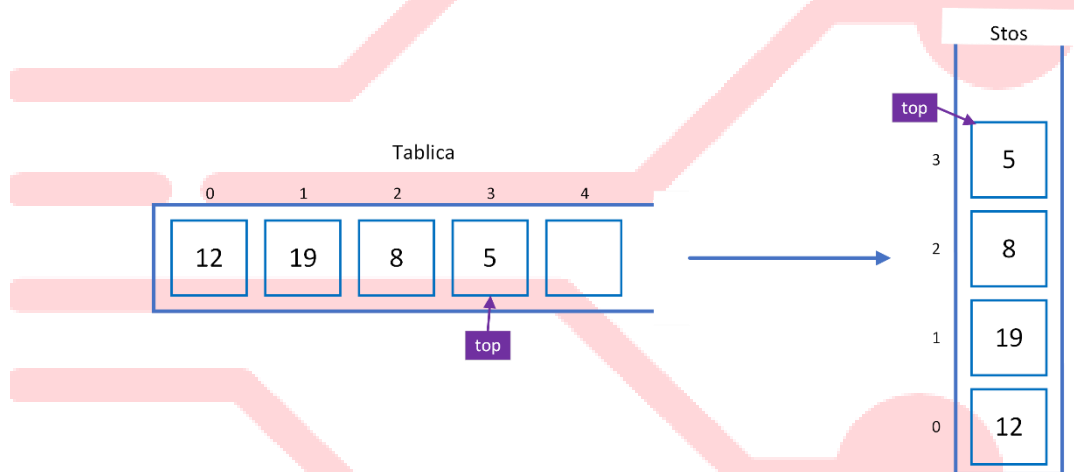
Przykład 5.5. Stos

W tym przykładzie rozważymy stos (*stack*). To struktura danych działająca zgodnie z zasadą LIFO. Oznacza to, że ostatni element dodany do struktury jest usuwany jako pierwszy.

Stos może być zaimplementowany na kilka sposobów. Wybór konkretnej implementacji wpływa na sposób zarządzania pamięcią oraz wydajność. Najczęściej spotykane implementacje to:

- implementacja tablicowa,
- implementacja z użyciem listy jednokierunkowej.

W implementacji tablicowej stos przechowywany jest w tablicy o określonym rozmiarze (patrz Rysunek II). Dodatkowo wykorzystywana jest zmienna (np. *top*), która wskazuje indeks aktualnego wierzchołka stosu.



Rysunek II Reprezentacja stosu za pomocą tablicy (źródło: opracowanie własne)

Zasada działania:

- Push – zwiększenie indeksu *top* i zapisanie elementu w tablicy,
- Pop – odczyt elementu spod indeksu *top* i zmniejszenie jego wartości,
- Peek – odczyt elementu spod indeksu *top* bez jego zmiany.

W implementacji wykorzystującej listę jednokierunkową stos reprezentowany jest jako sekwencja węzłów połączonych wskaźnikami. Referencja *top* odnosi się do pierwszego węzła listy, który jednocześnie stanowi wierzchołek stosu.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zasada działania:

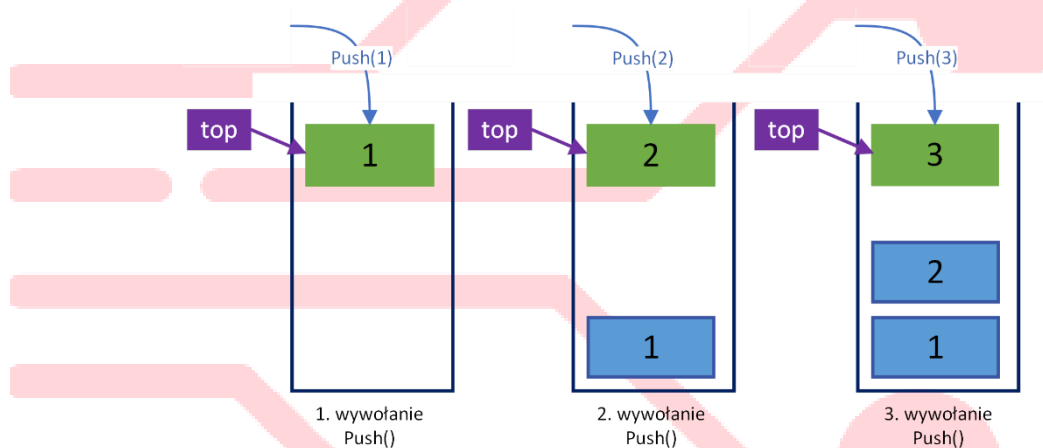
- Push – utworzenie nowego węzła i ustawienie go jako nowego początku listy,
- Pop – usunięcie pierwszego węzła i przesunięcie wskaźnika top,
- Peek – odczyt danych z pierwszego węzła.

Przeanalizujemy trzy podstawowe operacje dla stosu: Push, Peek oraz Pop.

Dodawanie elementów – metoda Push

Operacja Push polega na umieszczeniu nowego elementu na szczycie stosu. Każdy kolejny element trafia na wierzchołek.

Do stosu wstawimy kolejno następujące wartości: 1, 2, 3. Kolejne wywołania metody Push przedstawia Rysunek 12.



Rysunek 12 Kolejne wywołania metody Push() (źródło: opracowanie własne)

Należy zwrócić uwagę, że element 3 znajduje się obecnie na szczycie stosu, ponieważ został dodany jako ostatni.

Podgląd elementu – metoda Peek

Metoda Peek służy do sprawdzenia, jaki element znajduje się na szczycie stosu, bez jego usuwania. Operacja ta nie zmienia stanu struktury danych.

Dla aktualnego stanu stosu metoda Peek() zwraca wartość 3. Po wykonaniu tej operacji stos pozostaje w niezmienionej postaci (patrz Rysunek 13).



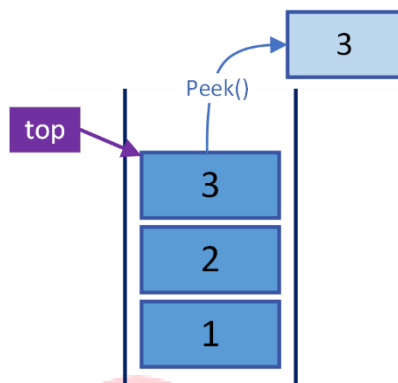
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



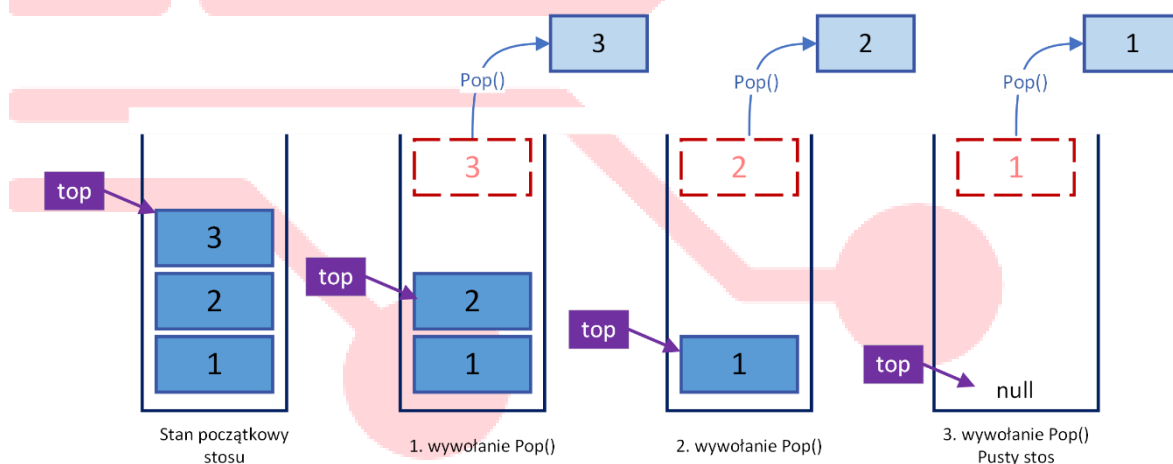


Rysunek 13 Przykład wywołania metody `Peek()`. Metoda zwraca element z wierzchołka stosu, ale nie usuwa go ze struktury (źródło: opracowanie własne)

Usuwanie elementów – metoda `Pop`

Metoda `Pop` zwraca element znajdujący się na szczycie stosu i usuwa go ze struktury. Każde wywołanie powoduje trwałą zmianę stanu stosu.

Kolejne wywołania operacji `Pop` (do momentu opróżnienia stosu) przedstawia Rysunek 14.



Rysunek 14 Kolejne wywołania metody `Pop()` dla stosu zawierającego trzy elementy [3, 2, 1] (źródło: opracowanie własne)

Łatwo zauważyć, że elementy zostały usunięte ze stosu w odwrotnej kolejności niż były dodawane.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykład 5.6. Stos – wbudowana implementacja

W niniejszym przykładzie przedstawiono wykorzystanie wbudowanej struktury stosu [Stack<T>] z przestrzeni nazw System.Collections.Generic.

Struktura Stack<T> udostępnia następujące metody:

- Push(element) – dodaje element na szczyt stosu,
- Pop() – usuwa i zwraca element ze szczytu stosu,
- Peek() – odczytuje wartość z wierzchołka stosu, [bez usuwania]
- Count – liczba elementów na stosie,
- Contains(element) – sprawdzenie, czy element znajduje się na stosie.

Poniższy program demonstruje podstawowe operacje na stosie.

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Utworzenie pustego stosu liczb całkowitych
        Stack<int> stack = new Stack<int>();

        // Dodawanie elementów na stos
        stack.Push(10);
        stack.Push(20);
        stack.Push(30);

        Console.WriteLine("Liczba elementów na stosie: " + stack.Count);
        Console.WriteLine("Element na szczycie stosu: " + stack.Peek());

        Console.WriteLine("Zdejmowanie elementów ze stosu:");

        while (stack.Count > 0)
        {
            int element = stack.Pop();
            Console.WriteLine(element);
        }

        Console.WriteLine("Czy stos jest pusty? " + (stack.Count == 0));
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Dla struktury `Stack<T>` złożoność operacji jest następująca:

- `Push()` – $O(1)$
- `Pop()` – $O(1)$
- `Peek()` – $O(1)$
- `Count` – $O(1)$

Operacje te są bardzo efektywne, ponieważ stos jest wewnętrznie zaimplementowany jako dynamiczna tablica.

Przykład 5.7. Kolejka

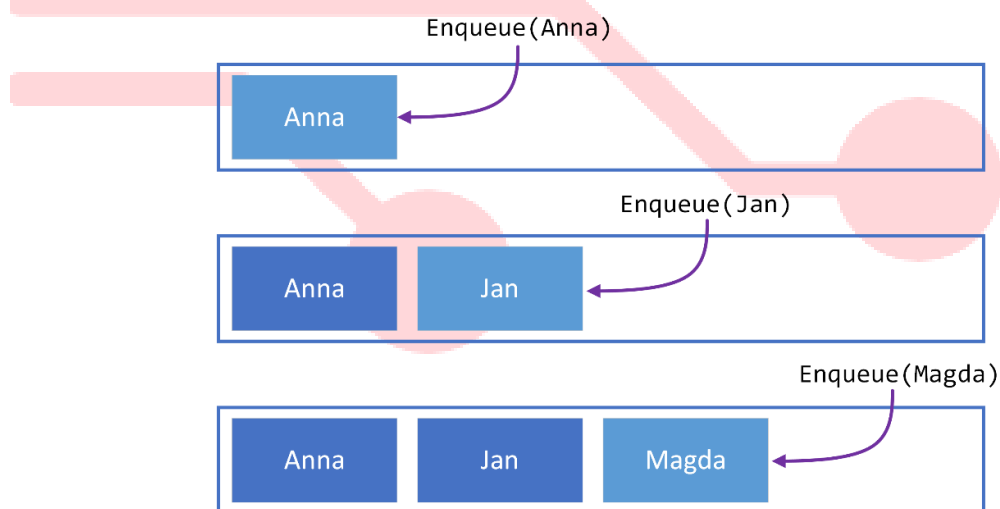
W tym przykładzie rozważymy kolejkę (*queue*). Kolejka działa zgodnie z zasadą FIFO (First In, First Out). Oznacza to, że element, który został dodany jako pierwszy, będzie pierwszym elementem zwróconym podczas usuwania.

Rozważymy trzy podstawowe operacje dla kolejki: `Enqueue`, `Peek` oraz `Dequeue`.

Dodawanie elementów – metoda `Enqueue`

Operacja `Enqueue` polega na dodaniu nowego elementu na koniec kolejki. Elementy ustawiają się jeden za drugim – tak jak osoby stojące w kolejce.

Do kolejki wstawimy następujące elementy: Anna, Jan, Magda. Kolejne wywołania metody `Enqueue` powodują zmiany stanu kolejki (patrz Rysunek 15).



Rysunek 15 Kolejne wywołania metody `Enqueue` (źródło: opracowanie własne)

Należy zwrócić uwagę, że Anna znajduje się na początku kolejki (została dodana jako pierwsza), natomiast Magda znajduje się na jej końcu.



Fundusze Europejskie
dla Rozwoju Społecznego



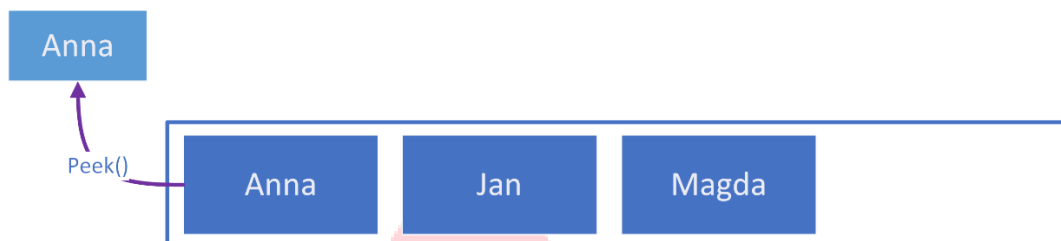
Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Podgląd elementu – metoda Peek

Metoda Peek, podobnie jak w przypadku stosu, służy do sprawdzenia, który element znajduje się na początku kolejki, bez usuwania go ze struktury. Operacja ta nie zmienia stanu kolejki.



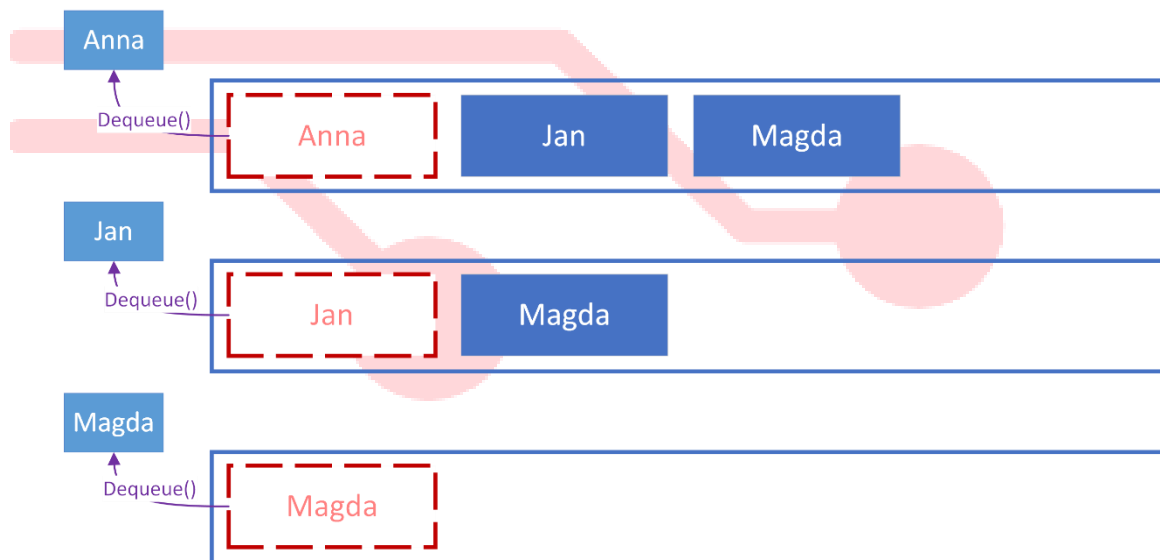
Rysunek 16 Stan kolejki po wywołaniu metody Peek()

Dla aktualnego stanu kolejki metoda Peek() zwraca „Anna”. Stan kolejki po tej operacji pozostaje bez zmian (patrz Rysunek 16).

Usuwanie elementów – metoda Dequeue

Metoda **Dequeue** zwraca element znajdujący się na początku kolejki oraz usuwa go z niej. Każde wywołanie powoduje trwałą zmianę stanu struktury danych.

Kolejne wywołania operacji Dequeue przedstawia Rysunek 17.



Rysunek 17 Kolejne wywołania metody Dequeue aż do momentu opróżnienia kolejki [źródło: opracowanie własne]

Elementy zostały usunięte w tej samej kolejności, w jakiej były dodawane.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykład 5.8. Kolejka – wbudowana implementacja

W niniejszym przykładzie przedstawiono wykorzystanie wbudowanej struktury kolejki [Queue<T>] z przestrzeni nazw System.Collections.Generic.

Struktura Queue<T> udostępnia następujące metody:

- Enqueue(element) – dodanie elementu na koniec kolejki,
- Dequeue() – usunięcie i zwrócenie elementu z początku kolejki,
- Peek() – podejrzenie elementu z początku kolejki (bez usuwania),
- Count – liczba elementów w kolejce,
- Contains(element) – sprawdzenie, czy element znajduje się w kolejce.

Program demonstruje podstawowe operacje na kolejce.

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main()
    {
        Queue<string> queue = new Queue<string>();

        queue.Enqueue("Anna");
        queue.Enqueue("Jan");
        queue.Enqueue("Magda");

        Console.WriteLine("Liczba osób w kolejce: " + queue.Count);

        Console.WriteLine("Pierwsza osoba w kolejce: " + queue.Peek());

        Console.WriteLine("\nRozpoczynamy obsługę klientów:");

        while (queue.Count > 0)
        {
            string klient = queue.Dequeue();
            Console.WriteLine("Obsłużono: " + klient);
        }

        Console.WriteLine("\nCzy kolejka jest pusta? "
            + (queue.Count == 0));
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Dla struktury `Queue<T>` złożoność operacji jest następująca:

- `Enqueue()` – $O(1)$
- `Dequeue()` – $O(1)$
- `Peek()` – $O(1)$
- `Count` – $O(1)$

Przykład 5.9. HashSet

W niniejszym przykładzie przedstawiono wykorzystanie klasy `HashSet<T>` z przestrzeni nazw `System.Collections.Generic`.

`HashSet<T>` reprezentuje zbiór elementów unikalnych, tzn. nie umożliwia przechowywania duplikatów. Struktura ta oparta jest na tablicy haszującej, dzięki czemu operacje wyszukiwania są bardzo szybkie [średnio $O(1)$].

Najczęściej używane metody:

- `Add(element)`
- `Remove(element)`
- `Contains(element)`
- `UnionWith(collection)`
- `IntersectWith(collection)`
- `ExceptWith(collection)`
- `Count`

Przykład – usuwanie duplikatów

Założmy, że mamy listę numerów id pracowników, w której mogą pojawiać się powtórzenia. Chcemy uzyskać zbiór unikalnych indeksów.

```
using System;
using System.Collections.Generic;
class Program
{
    static void Main(string[] args)
    {
        List<int> idPracownikow = new List<int>()
        {
            101, 102, 103, 101, 104, 102, 105
        };
        Console.WriteLine("Lista z duplikatami:");
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
Display(idPracownikow);

// Utworzenie HashSet na podstawie listy
HashSet<int> unikatoweId = new HashSet<int>(idPracownikow);

Console.WriteLine("\nUnikatowe identyfikatory (HashSet):");
Display(unikatoweId);

// Próba dodania istniejącego elementu
bool czyDodano = unikatoweId.Add(103);
Console.WriteLine($"Czy dodano 103 ponownie? " +
    $"{czyDodano}");

// Sprawdzenie przynależności elementu do zbioru
Console.WriteLine($"Czy zbiór zawiera 104? " +
    $"{unikatoweId.Contains(104)}");

// Usunięcie elementu
unikatoweId.Remove(105);
Console.WriteLine("\nPo usunięciu 105:");
Display(unikatoweId);

Console.WriteLine($"Liczba elementów w zbiorze: " +
    $"{unikatoweId.Count}");
}

static void Display(IEnumerable<int> collection)
{
    foreach (var element in collection)
        Console.Write(element + " ");
    Console.WriteLine();
}
}
```

Początkowa lista zawiera duplikaty:

101 102 103 101 104 102 105

Po utworzeniu obiektu `HashSet<int>` na podstawie tej listy w zbiorze pozostają wyłącznie elementy unikalne:

101 102 103 104 105

Należy zauważyć, że kolejność elementów może być inna, ponieważ `HashSet<T>` nie gwarantuje zachowania porządku dodawania elementów.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Próba ponownego dodania wartości 103 powoduje zwrócenie wartości false, ponieważ element ten już znajduje się w zbiorze i `HashSet<T>` nie dopuszcza duplikatów.

W przypadku `HashSet<T>` złożoność operacji jest następująca:

- `Add()` – $O(1)$
- `Remove()` – $O(1)$
- `Contains()` – $O(1)$
- operacje na zbiorach – $O(n)$.

W najgorszym przypadku (kolizje haszowania) operacje mogą mieć złożoność $O(n)$, ale przy dobrej funkcji haszującej taka sytuacja występuje dość rzadko.

Ćwiczenia samodzielne

Zadanie 5.1. Lista jednokierunkowa

Zaimplementuj klasę `SinglyLinkedList`, która będzie reprezentować listę jednokierunkową liczb całkowitych. Lista powinna przechowywać referencję do pierwszego elementu (`head`).

Klasa powinna zawierać metody umożliwiające:

- sprawdzenie, czy lista jest pusta,
- dodawanie elementu na początek listy,
- dodawanie elementu na koniec listy,
- usuwanie pierwszego elementu listy,
- usuwanie elementu o podanej wartości,
- wyszukiwanie elementu o zadanej wartości,
- wyświetlanie zawartości listy.

Dodatkowo zdefiniuj prywatną klasę `Node`, która będzie reprezentować pojedynczy węzeł listy. Klasa ta powinna zawierać dwa pola: `Value` typu `int` oraz `Next` typu `Node`. Przeanalizuj złożoność czasową każdej operacji.

Zadanie 5.2. Lista jednokierunkowa z dwoma wskaźnikami

Zmodyfikuj implementację klasy `SinglyLinkedList` z Zadania 5.1 tak, aby lista przechowywała dwie referencje: `head`, wskazującą na pierwszy element listy, oraz `tail`, wskazującą na ostatni element listy.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Jak zmieni się złożoność czasowa operacji dodawania elementu na koniec listy po wprowadzeniu tail? Czy posiadanie referencji tail w równym stopniu przyspieszy operację usuwania ostatniego elementu listy?

Zadanie 5.3. Lista jednokierunkowa – przechowywanie wydarzeń

Zaimplementuj listę jednokierunkową przechowującą obiekty klasy Wydarzenie. Klasa Wydarzenie powinna reprezentować pojedyncze zdarzenie w kalendarzu i zawierać:

- datę wydarzenia (dzień, miesiąc, rok),
- opis wydarzenia (napis).

Ponadto klasa powinna zawierać:

- metody umożliwiające dostęp do pól,
- metodę Wyświetl(), która wyświetla datę oraz opis wydarzenia,
- metodę DodajOpis(string opis), umożliwiającą dopisanie kolejnego opisu do wydarzenia odbywającego się tego samego dnia,
- metodę PorównajDate(int dzien, int miesiac, int rok), sprawdzającą czy data obiektu wywołującego tę metodę jest równa dacie przekazanej w parametrach.

Lista powinna umożliwiać:

- dodawanie nowego wydarzenia o podanej dacie,
- jeżeli wydarzenie o tej samej dacie już istnieje, dopisanie opisu do istniejącego obiektu,
- usuwanie wydarzenia o wskazanej dacie,
- wyszukiwanie i wyświetlanie pojedynczego wydarzenia,
- wyświetlanie wszystkich wydarzeń zapisanych w liście.

Zadanie 5.4. Lista dwukierunkowa

Zaimplementuj klasę DoublyLinkedList, która będzie reprezentować listę dwukierunkową liczb całkowitych. Lista powinna przechowywać referencje head oraz tail, wskazujące odpowiednio na pierwszy i ostatni element listy.

Zdefiniuj prywatną klasę Node, reprezentującą pojedynczy węzeł listy. Każdy węzeł powinien zawierać trzy pola:

- Value typu int,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- Next typu Node,
- Prev typu Node.

Klasa DoublyLinkedList powinna umożliwiać:

- a) dodawanie elementu na początek listy,
- b) dodawanie elementu na koniec listy,
- c) usuwanie elementu o podanej wartości,
- d) usuwanie pierwszego elementu,
- e) usuwanie ostatniego elementu,
- f) wyszukiwanie elementu o zadanej wartości,
- g) wyświetlanie zawartości listy.

Porównaj działanie oraz złożoność czasową operacji w liście jednokierunkowej i dwukierunkowej.

Zadanie 5.5. Słownik

Korzystając ze słownika, pogrupuj wyrazy z listy ["kot", "tok", "kto", "sok", "kos", "kula", "luka", "wór", "rów", "tara", "rata", "masa", "sama", "grad", "drga"] w taki sposób, aby w jednej grupie znalazły się słowa będące swoimi anagramami. Jako klucz w słowniku zastosuj wyraz zapisany jako ciąg liter ułożonych w porządku alfabetycznym (czyli posortowane znaki danego słowa). Na końcu wypisz wszystkie utworzone grupy anagramów.

Zadanie 5.6. Zbiór

Dana jest lista obiektów klasy Pracownik. Każdy obiekt przechowuje *imię*, *nazwisko* oraz *identyfikator* (ID). Przyjmij, że duplikatem jest pracownik posiadający ten sam identyfikator.

Napisz program, który usuwa duplikaty z listy pracowników z wykorzystaniem zbioru (HashSet).

Jaka jest średnia złożoność takiego rozwiązania? Dlaczego zastosowanie zbioru jest bardziej efektywne niż przeszukiwanie listy?

Zadanie 5.7. Stos

Zaimplementuj stos (LIFO) przy użyciu:

- a) tablicy,
- b) listy jednokierunkowej (z zadania 5.1).



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Struktura powinna zawierać następujące operacje Push, Pop, Peek oraz IsEmpty (sprawdza czy stos jest pusty).

Porównaj obie implementacje pod względem złożoności operacji.

Zadanie 5.8. Kolejka

Zaimplementuj kolejkę FIFO przy użyciu tablicy. Struktura powinna zawierać następujące operacje: Enqueue, Dequeue, Peek, IsEmpty.

Zadanie 5.9. Palindrom

Napisz program, który sprawdzi, czy podany wyraz jest palindromem, wykorzystując stos i kolejkę. Nie używaj gotowych metod odwracania napisu.

Zadanie 5.10. Nawiasy

Napisz program sprawdzający poprawność nawiasów (oraz) w wyrażeniu, np. "(a+b)*(c-d)".

Pytania pomocnicze

- 1) Jakie są podstawowe operacje dla stosu?
- 2) Jakie są podstawowe operacje dla kolejki?
- 3) Czym różni się lista oparta na tablicy od listy wiązanej pod względem organizacji danych w pamięci?
- 4) Dlaczego w liście opartej na tablicy wstawianie „w środku” jest kosztowne?
- 5) Na czym polega zasada LIFO i jakie są jej praktyczne zastosowania?
- 6) Na czym polega zasada FIFO i w jakich problemach jest naturalnym modelem przetwarzania danych?
- 7) Dlaczego operacje na stosie i kolejce mają zazwyczaj złożoność $O(1)$?

Podsumowanie

W ramach niniejszego laboratorium przeanalizowano podstawowe struktury danych: listę, stos oraz kolejkę. Szczególną uwagę poświęcono sposobowi organizacji danych w tych strukturach, charakterystycznym dla nich operacjom oraz analizie złożoności czasowej podstawowych działań, takich jak dodawanie, usuwanie i wyszukiwanie elementów.



Fundusze Europejskie
dla Rozwoju Społecznego



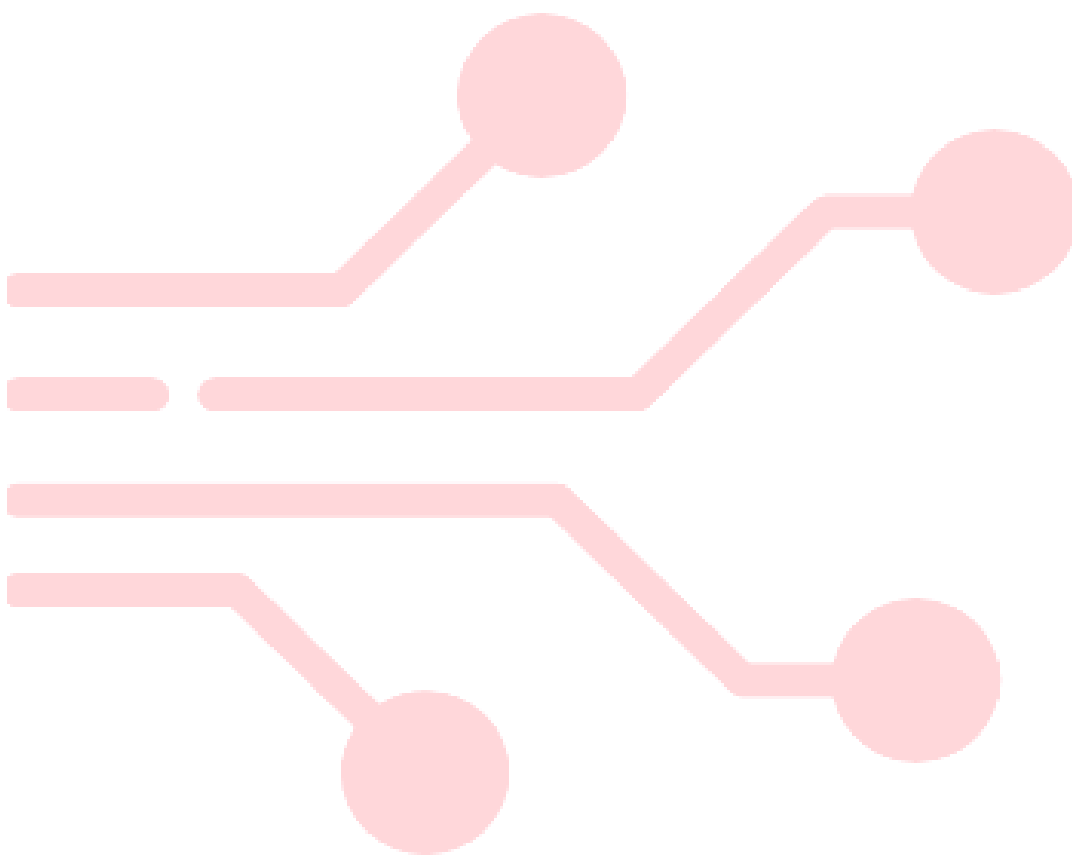
Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Literatura

- [1] Drozdek A., *C++. Algorytmy i struktury danych*, Helion, Gliwice 2004. s. 81-146.
- [2] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C., *Wprowadzenie do algorytmów*, PWN, Warszawa, 2023. s. 238-248.
- [3] Aho A., Hopcroft J., Ullman J., *Algorytmy i struktury danych*, Helion, Gliwice, 2003. s. 49-73.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Materiały do przedmiotu:

Algorytmy i struktury danych

Laboratorium nr 6

Implementacja kolejki priorytetowej

Autor:

dr hab. Adam Kiersztyn
mgr Krystyna Kiersztyn



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Spis treści

Wprowadzenie, tematyka zajęć	108
Cel ćwiczenia/laboratorium	108
Instrukcja laboratoryjna/ćwiczeniowa	109
Ćwiczenia samodzielne	123
Pytania pomocnicze	125
Podsumowanie	125
Literatura	125



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wprowadzenie, tematyka zajęć

Kolejka priorytetowa jest jedną z podstawowych struktur danych. Jej istotą jest przechowywanie elementów wraz z przypisanymi im priorytetami, które decydują o kolejności ich przetwarzania. W odróżnieniu od klasycznej kolejki typu FIFO (First In, First Out), w kolejce priorytetowej elementy są usuwane zgodnie z ich priorytetem, a nie kolejnością dodania.

Znaczenie kolejek priorytetowych jest szczególnie widoczne w aplikacjach, w których konieczne jest szybkie reagowanie na zdarzenia krytyczne lub optymalne zarządzanie zasobami. Dzięki tej strukturze możliwe jest efektywne przetwarzanie zadań o różnym stopniu ważności, co przekłada się na lepszą wydajność oraz przewidywalne zachowanie systemu.

Kolejka priorytetowa może być implementowana na różne sposoby, m.in. z wykorzystaniem tablic, list czy kopców. W praktyce najczęściej stosowanym rozwiązaniem jest kopiec binarny, który zapewnia korzystną złożoność czasową dla podstawowych operacji.

Podstawowe operacje wykonywane na kolejce priorytetowej:

- Enqueue – dodanie elementu do kolejki,
- Dequeue – usunięcie elementu o najwyższym priorytecie,
- Peek – zwrócenie elementu o najwyższym priorytecie bez usuwania,

Złożoność czasowa operacji wykonywanych na kolejce priorytetowej zależy bezpośrednio od zastosowanej struktury implementacyjnej. W przypadku kopca binarnego możliwe jest osiągnięcie logarytmicznej złożoności czasowej dla operacji wstawiania i usuwania elementów.

Kolejki priorytetowe są powszechnie wykorzystywane m.in. w systemach planowania procesów, algorytmach grafowych (takich jak algorytm Dijkstry), grach komputerowych, zarządzaniu ruchem sieciowym oraz w systemach wspomagających podejmowanie decyzji, np. w ochronie zdrowia.

Cel ćwiczenia/laboratorium

Po zakończeniu laboratorium student:

- rozumie pojęcie kolejki priorytetowej oraz zasadę jej działania,



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- zna podstawowe operacje wykonywane na kolejce priorytetowej,
- potrafi wyjaśnić różnice pomiędzy kolejką priorytetową a klasyczną kolejką FIFO,
- rozumie wpływ zastosowanej struktury danych na złożoność czasową operacji,
- zna podstawowe metody implementacji kolejki priorytetowej, ze szczególnym uwzględnieniem kopca binarnego,
- rozumie praktyczne zastosowania kolejki priorytetowej w algorytmach i systemach informatycznych.

Instrukcja laboratoryjna/ćwiczeniowa

Tabela 1 przedstawia różnice między kolejką priorytetową a klasyczną kolejką FIFO.

Tabela 1 Różnice między kolejką priorytetową a FIFO

Cecha / Operacja	Kolejka FIFO	Kolejka priorytetowa
Kryterium obsługi elementów	Kolejność dodania (pierwszy wchodzi, pierwszy wychodzi)	Priorytet elementu (najwyższy/najniższy jako pierwszy)
Enqueue	Dodanie na koniec kolejki	Dodanie elementu wraz z priorytetem, reorganizacja struktury
Dequeue	Usunięcie elementu z początku	Usunięcie elementu o najwyższym/najniższym priorytecie
Peek	Odczyt pierwszego elementu	Odczyt elementu o najwyższym/najniższym priorytecie
Typowe implementacje	Tablica, lista	Kopiec binarny, drzewa
Główna zaleta	Prostota i przewidywalność kolejności	Szybki dostęp do „najważniejszego” elementu

Przykład 6.1. Kolejka priorytetowa z użyciem tablicy/listy

W tym przykładzie rozważymy kolejkę priorytetową zaimplementowaną w oparciu o tablicę (lub listę). Jest to najprostsza możliwa implementacja tej struktury danych. W przeciwieństwie do wersji opartej na kopcu, kolejka nie utrzymuje elementów w uporządkowanej kolejności podczas dodawania.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przyjmujemy, że mniejsza wartość oznacza wyższy priorytet (np. priorytet 0 jest wyższy niż 3).

Do kolejki wstawimy następujące elementy:

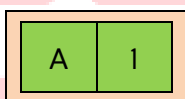
Element	Priorytet
Zadanie A	1
Zadanie B	3
Zadanie C	2
Zadanie D	0

Dodawanie elementów – metoda Enqueue

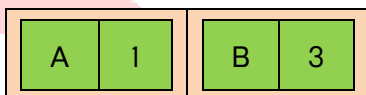
Operacja Enqueue w tej implementacji polega wyłącznie na dopisaniu nowego elementu na koniec tablicy. Podczas dodawania nie jest wykonywane żadne porządkowanie względem priorytetu. Dzięki temu operacja ta ma złożoność czasową $O(1)$.

Kolejne wywołania Enqueue powodują następujące zmiany stanu kolejki:

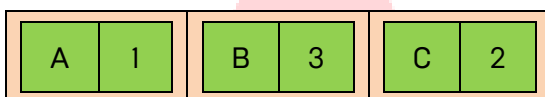
- 1) Po dodaniu elementu **Zadanie A (1)**:



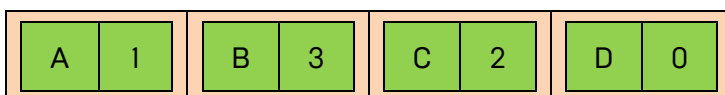
- 2) Po dodaniu elementu **Zadanie B (3)**:



- 3) Po dodaniu elementu **Zadanie C (2)**:



- 4) Po dodaniu elementu **Zadanie D (0)**:



Należy zwrócić uwagę, że elementy w tablicy nie są uporządkowane według priorytetów, lecz znajdują się w niej w takiej kolejności, w jakiej zostały dodane.

Podgląd elementu – metoda Peek

Metoda Peek służy do sprawdzenia, który element ma aktualnie najwyższy priorytet, bez usuwania go z kolejki. W tym celu algorytm przegląda całą tablicę i wyszukuje element o najmniejszej wartości priorytetu.

Dla aktualnego stanu kolejki:

A	1	B	3	C	2	D	0
---	---	---	---	---	---	---	---

najwyższy priorytet ma **Zadanie D (0)**, dlatego metoda Peek zwraca właśnie ten element. Stan kolejki po tej operacji nie ulega zmianie.

Usuwanie elementów – metoda Dequeue

Metoda Dequeue zwraca element o najwyższym priorytecie oraz jednocześnie usuwa go z kolejki. Każde jej wywołanie powoduje trwałą zmianę stanu struktury danych. Usunięcie elementu powoduje przesunięcie pozostałych elementów, co sprawia, że operacja ta ma złożoność $O(n)$.

Kolejne wywołania operacji Dequeue przebiegają następująco:

1) Pierwsze wywołanie

Metoda zwraca element **Zadanie D** o najwyższym priorytecie (0) i usuwa go z kolejki. Stan kolejki po wywołaniu Dequeue:

A	1	B	3	C	2
---	---	---	---	---	---

2) Drugie wywołanie

Zwrócony zostaje element **Zadanie A** o priorytecie 1, który następnie zostaje usunięty z kolejki. Stan kolejki po wywołaniu Dequeue:

B	3	C	2
---	---	---	---

3) Trzecie wywołanie

Metoda zwraca element **Zadanie C** o priorytecie 2 i usuwa go z kolejki. Stan kolejki po wywołaniu Dequeue:

B	3
---	---

4) Czwarte wywołanie

Zwrócony zostaje ostatni element **Zadanie B** o priorytecie 3, po czym kolejka staje się pusta.



Warto zauważyć, że kolejność usuwania elementów zależy wyłącznie od ich priorytetów. Przykładowa implementacja kolejki priorytetowej bazującej na liście wygląda następująco:

```
using System.Collections.Generic;

public class ListPriorityQueue
{
    private class Item
    {
        public string Value;
        public int Priority;

        public Item(string value, int priority)
        {
            Value = value;
            Priority = priority;
        }
    }

    private readonly List<Item> items = new List<Item>();
    public int Count => items.Count;
    public bool IsEmpty => items.Count == 0;

    public void Enqueue(string value, int priority)
    {
        items.Add(new Item(value, priority));
    }

    public string Peek()
    {
        if (IsEmpty)
            throw new InvalidOperationException("Kolejka jest pusta.");

        int bestIndex = FindBestIndex();
        return items[bestIndex].Value;
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
public string Dequeue()
{
    if (IsEmpty)
        throw new InvalidOperationException("Kolejka jest pusta.");
    int bestIndex = FindBestIndex();
    string value = items[bestIndex].Value;
    // RemoveAt usuwa element i przesuwa pozostałe w lewo
    items.RemoveAt(bestIndex);

    return value;
}
// Szuka indeksu elementu o najwyższym priorytecie
private int FindBestIndex()
{
    int best = 0;

    for (int i = 1; i < items.Count; i++)
    {
        if (items[i].Priority < items[best].Priority)
            best = i;
    }
    return best;
}
}

public class Program
{
    static void Main(string[] args)
    {
        ListPriorityQueue pq = new ListPriorityQueue();
        pq.Enqueue("Zadanie A", 1);
        pq.Enqueue("Zadanie B", 3);
        pq.Enqueue("Zadanie C", 2);
        pq.Enqueue("Zadanie D", 0);

        Console.WriteLine("Peek: " + pq.Peek()); // Zadanie D

        while (!pq.IsEmpty)
        {
            Console.WriteLine(pq.Dequeue());
        }
    }
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykład 6.2. Kolejka priorytetowa z użyciem kopca binarnego

W tym przykładzie rozważymy kolejkę priorytetową zaimplementowaną w oparciu o kopiec binarny (typu min), w którym element o najwyższym priorytecie znajduje się zawsze w korzeniu drzewa.

Kopiec binarny spełnia dwa warunki:

- jest **drzewem zupełnym** (wypełnianym poziomami od lewej do prawej),
- każdy węzeł ma priorytet **nie większy** niż jego dzieci (w kopcu typu min).

Dla węzła pod indeksem i :

- lewe dziecko znajduje się pod indeksem $2*i + 1$.
- prawe dziecko znajduje się pod indeksem $2*i + 2$.

Rodzic węzła o indeksie i znajduje się na pozycji $(i-1)/2$.

Wysokość kopca wynosi $\log_2 n$, dlatego operacje mają złożoność logarytmiczną. W odróżnieniu od implementacji tablicowej, kolejka oparta na kopcu utrzymuje własność kopca już w trakcie dodawania i usuwania elementów, co wpływa na złożoności operacji:

- Enqueue – $O(\log n)$,
- Dequeue – $O(\log n)$,
- Peek – $O(1)$.

Do kopca wstawimy elementy w postaci par (element, priorytet):

(A, 4), (B, 2), (C, 3), (D, 1).

Dodawanie elementów do kopca (Enqueue)

W kopcu binarnym typu *min* operacja Enqueue składa się zawsze z dwóch faz:

- Wstawienie elementu na koniec kopca*

Element jest umieszczany w pierwszym wolnym miejscu na najniższym poziomie drzewa. Dzięki temu kopiec pozostaje drzewem zupełnym.

- Przywrócenie własności kopca (przesiewanie w górę / heapify-up)*

Po wstawieniu element jest porównywany z jego rodzicem:

- Jeżeli priorytet elementu jest wyższy niż priorytet rodzica, następuje ich zamiana (mniejsza wartość → wyższy priorytet),



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

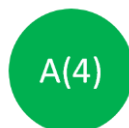
Dofinansowane przez
Unię Europejską



- o Porównanie jest powtarzane „w górę drzewa”, aż element znajdzie poprawną pozycję lub osiągnie korzeń.

I. Wstawienie elementu A(4)

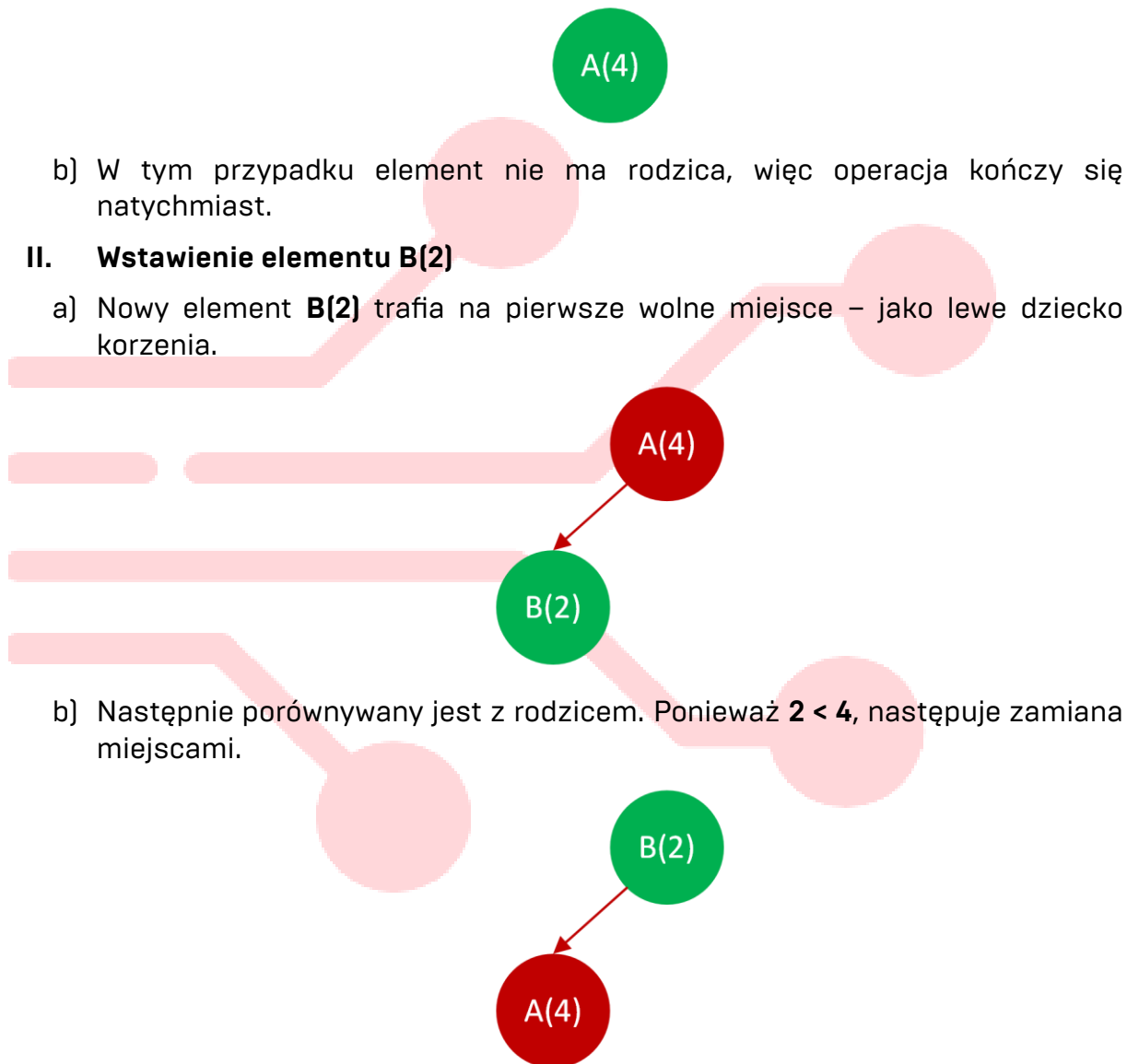
- Na początku kopiec jest pusty, zatem element **A(4)** zostaje umieszczony jako korzeń kopca.



- W tym przypadku element nie ma rodzica, więc operacja kończy się natychmiast.

II. Wstawienie elementu B(2)

- Nowy element **B(2)** trafia na pierwsze wolne miejsce – jako lewe dziecko korzenia.



- Następnie porównywany jest z rodzicem. Ponieważ $2 < 4$, następuje zamiana miejscami.

Operacja kończy się, bo **B(2)** jest już w korzeniu.



Fundusze Europejskie
dla Rozwoju Społecznego



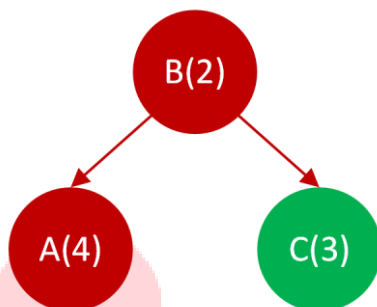
Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



III. Wstawienie elementu C(3)

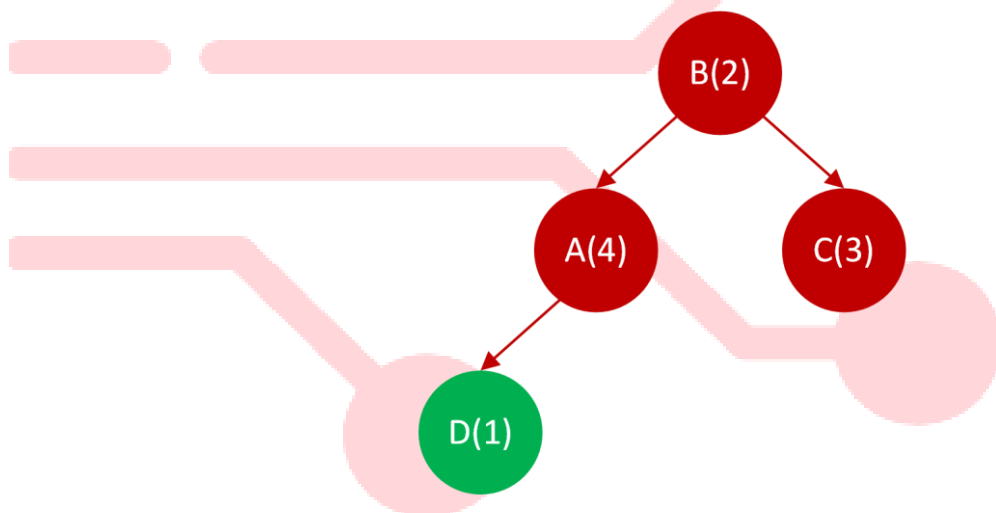
- a) Nowy element **C(3)** trafia na kolejne wolne miejsce – jako prawe dziecko korzenia.



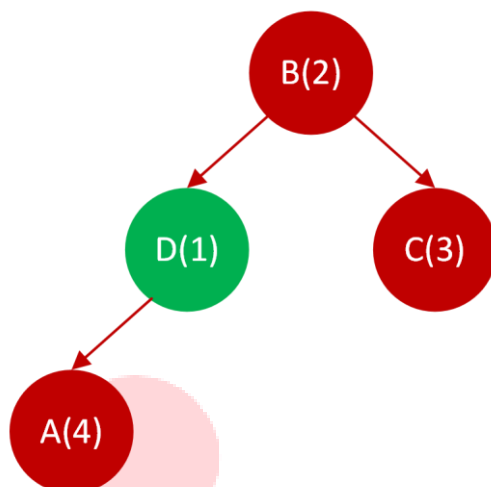
- b) Ponieważ $3 > 2$, struktura kopca pozostaje bez zmian (spełniona jest własność kopca).

IV. Wstawienie elementu D(1)

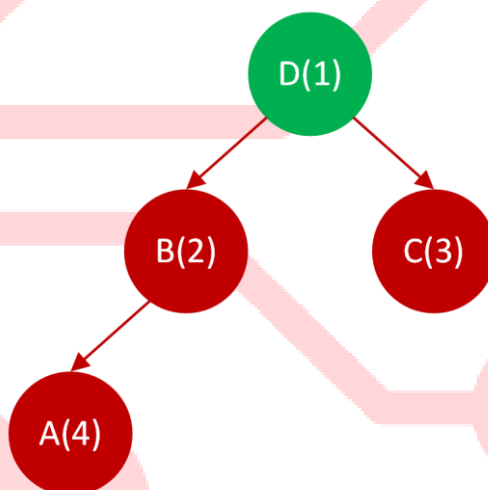
- a) Nowy element **D(1)** trafia na kolejne wolne miejsce – jako lewe dziecko **A(4)**.



- b) Porównanie **D(1)** z rodzicem **A(4)**: $1 < 4 \rightarrow$ zamiana.



W kolejnym kroku wykonywane jest porównanie **D(1)** z nowym rodzicem **B(2)**:
 $1 < 2 \rightarrow$ **zamiana**.



D(1) osiąga korzeń, więc przesiewanie w górę kończy się.

Po zakończeniu operacji Enqueue w korzeniu kopca znajduje się element o najwyższym priorytecie.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



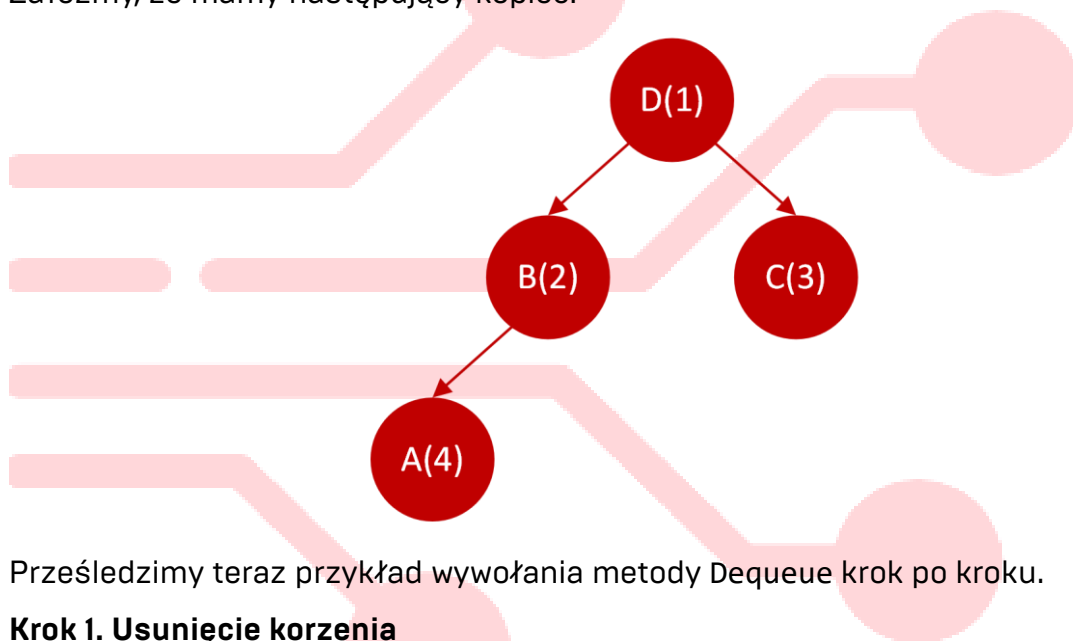
Usuwanie elementu z kopca (Dequeue)

W kopcu binarnym typu *min* operacja Dequeue zwraca element o najwyższym priorytecie i usuwa go ze struktury. Następnie struktura jest reorganizowana, aby ponownie spełniała własność kopca.

Operacja Dequeue składa się zawsze z trzech faz:

- 1) Odczyt i usunięcie korzenia (to element o najwyższym priorytecie).
- 2) Przeniesienie ostatniego elementu do korzenia (aby zachować drzewo zupełne).
- 3) Przywrócenie własności kopca (przesiewanie w dół / heapify-down).

Założmy, że mamy następujący kopiec.



Prześledzimy teraz przykład wywołania metody Dequeue krok po kroku.

Krok 1. Usunięcie korzenia

Korzeń kopca to element o najwyższym priorytecie, więc:

- metoda Dequeue zwraca element D[1],
- element D[1] jest usuwany z kopca.

Krok 2. Przeniesienie ostatniego elementu do korzenia

Aby zachować własność drzewa zupełnego, element znajdujący się na ostatniej pozycji kopca (najbardziej na prawo, na najniższym poziomie) zostaje przeniesiony na miejsce usuniętego korzenia. W tym przypadku to **A(4)**. Jego poprzednia pozycja zostaje usunięta. Po tych operacjach kopiec wygląda następująco:



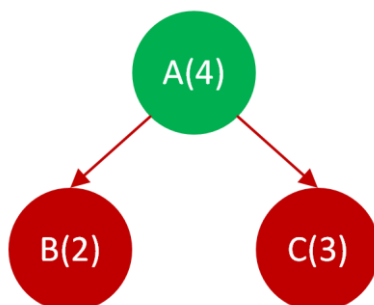
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską

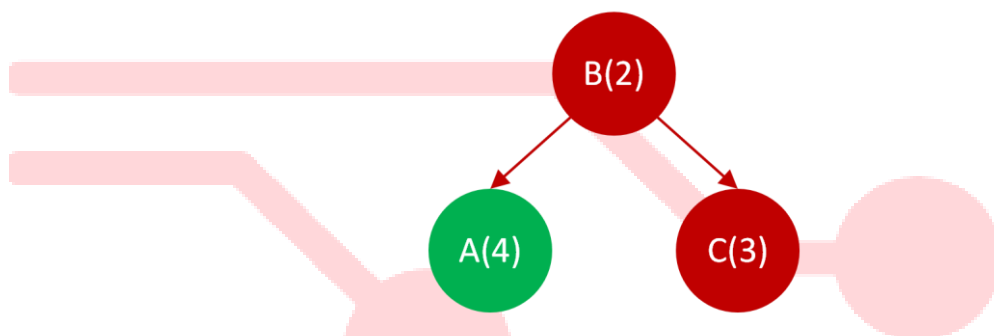




Krok 3. Przesiewanie w dół (heapify-down)

W tym momencie własność kopca jest naruszona, ponieważ **A(4)** ma mniejszy priorytet niż jego dzieci. W celu jej przywrócenia element ten jest porównywany z jego dziećmi i zamieniany z tym dzieckiem, które posiada wyższy priorytet. Operacja ta jest powtarzana, aż element znajdzie się na poprawnej pozycji lub osiągnie poziom liści.

W tym przypadku dziećmi korzenia są elementy **B(2)** oraz **C(3)**. Spośród nich wybierany jest element o wyższym priorytecie, czyli **B(2)**. Następnie wykonywane jest porównanie elementu **A(4)** z **B(2)**. Ponieważ $2 < 4$, następuje zamiana tych elementów miejscami.



Po zamianie element **A(4)** znajduje się na niższym poziomie kopca. Należy sprawdzić, czy posiada on dzieci. W tym przypadku element **A(4)** nie ma dzieci, dlatego proces przesiewania zostaje zakończony.

W wyniku wykonania operacji Dequeue zwrócony i usunięty zostaje element **D(1)**. Kopiec po zakończeniu reorganizacji ma postać:



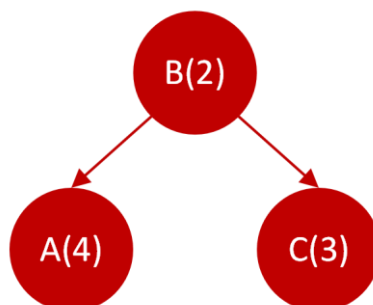
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską





Po zakończeniu przesiewania w dół nasza struktura ponownie spełnia własność kopca, a w korzeniu znajduje się element o najwyższym priorytecie spośród pozostałych.

Każde kolejne wywołanie operacji Dequeue przebiega analogicznie i prowadzi do stopniowego opróżniania kopca, przy jednoczesnym zachowaniu poprawnej struktury danych po każdej operacji.

Przykład 6.3. Wbudowana implementacja kolejki priorytetowej

W niniejszym przykładzie omówimy klasę `PriorityQueue` dostępną w przestrzeni nazw `System.Collections.Generic` (dostępna od .NET 6), która jest to wbudowaną implementacją kolejki priorytetowej. Każdy element kolejki priorytetowej ma postać pary (`TElement`, `TPriority`):

- `TElement` – typ elementu (np. `string`, `int`, klasa),
- `TPriority` – typ priorytetu (najczęściej `int`).

Implementacja ta oparta jest na kopcu (`min-heap`), czyli element o najmniejszej wartości priorytetu zostanie zwrócony jako pierwszy. Oznacza to, że mniejsza liczba oznacza wyższy priorytet.

Tworzenie obiektu kolejki

Kolejkę tworzymy poprzez wywołanie konstruktora:

```
PriorityQueue<string, int> pq = new PriorityQueue<string, int>();
```

Pierwszy parametr generyczny (`string`) określa typ elementu, drugi – typ priorytetu (`int`).

Dodawanie elementu — `Enqueue(element, priority)`

```
pq.Enqueue("Zadanie A", 2);
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Metoda dodaje element wraz z jego priorytetem i umieszcza go w odpowiednim miejscu struktury.

Pobranie i usunięcie elementu — Dequeue() oraz TryDequeue()

```
string element = pq.Dequeue();
```

Metoda Dequeue zwraca element o najwyższym priorytecie i usuwa go z kolejki. Jeżeli kolejka jest pusta, zgłaszany jest wyjątek `System.InvalidOperationException`.

Bezpieczniejszą wersją jest:

```
pq.TryDequeue(out string element, out int priority);
```

Metoda TryDequeue zwraca `false`, jeżeli kolejka jest pusta, i nie generuje wyjątku. Ponadto pozwala jednocześnie odczytać element i jego priorytet.

Podgląd elementu — Peek() oraz TryPeek()

```
var element = pq.Peek();
```

Zwraca element o najwyższym priorytecie bez jego usuwania. Jeżeli kolejka jest pusta, zgłaszany jest wyjątek `System.InvalidOperationException`.

Bezpieczniejsza wersja:

```
pq.TryPeek(out string element, out int priority);
```

Podobnie do TryDequeue pozwala jednocześnie odczytać element i jego priorytet (oczywiście bez usuwania go z kolejki).

Operacje pomocnicze

- Właściwość `Count` zwraca liczbę elementów w kolejce.

```
int n = pq.Count;
```

- Metoda `Clear()` usuwa wszystkie elementy z kolejki. Po jej wywołaniu właściwość `Count` ma wartość 0.

```
pq.Clear();
```

- Właściwość `UnorderedItems` pozwala przejrzeć zawartość kolejki.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```
foreach (var item in pq.UnorderedItems)
    Console.WriteLine($"{item.Element} ({item.Priority})");
```

Należy jednak podkreślić, że elementy nie są zwracane w kolejności priorytetów. Właściwość ta służy jedynie do wglądu w aktualny stan struktury.

Iterowanie po kolejce przy użyciu pętli while

Najczęstszym sposobem przetwarzania wszystkich elementów kolejki priorytetowej jest stopniowe jej opróżnianie za pomocą metody Dequeue lub TryDequeue w pętli while.

a) Wersja z Dequeue:

```
while (pq.Count > 0)
{
    string element = pq.Dequeue();
    Console.WriteLine($"Przetwarzanie: {element}");
}
```

b) Wersja z TryDequeue:

```
while (pq.TryDequeue(out string element, out int priority))
{
    Console.Write($"Przetwarzanie: ");
    Console.WriteLine($"{element} (priorytet: {priority})");
}
```

Modyfikacja sposobu porównywania elementów

Domyślnie wykorzystywany jest `Comparer<TPriority>.Default`. Warto zaznaczyć, że możliwe jest przekazanie własnego komparatora w konstruktorze kolejki:

```
var pq = new PriorityQueue<string, int>(
    Comparer<int>.Create((a, b) => b.CompareTo(a))
);
```

Pozwala to odwrócić porządek porównywania elementów i uzyskać strukturę typu max-heap, gdzie większa liczba oznacza wyższy priorytet.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Ćwiczenia samodzielne

Zadanie 6.1. Harmonogram zadań

Napisz program, który wykorzystuje wbudowaną kolejkę priorytetową `PriorityQueue<TElement, TPriority>` dostępną w platformie .NET. Program powinien symulować prosty harmonogram zadań.

W programie należy utworzyć kolejkę priorytetową, dodać do niej kilka zadań wraz z przypisanymi im priorytetami, a następnie pobierać i usuwać zadania z kolejki przy użyciu odpowiednich operacji.

Zadanie 6.2. Kolejka priorytetowa typu max

Zaimplementuj własną kolejkę priorytetową typu *max*, wykorzystując strukturę `SortedDictionary<TKey, Queue<TValue>>`. Kolejka powinna przechowywać elementy wraz z przypisanymi im priorytetami.

Utwórz klasę reprezentującą kolejkę priorytetową oraz zaimplementuj następujące operacje:

- `Enqueue` – dodawanie elementu do kolejki wraz z jego priorytetem,
- `Dequeue` – zwracanie i usuwanie elementu o najwyższym priorytecie,
- `Peek` – zwracanie elementu o najwyższym priorytecie bez jego usuwania.

Zadbaj o poprawną obsługę sytuacji, w której wykonywana jest próba operacji `Dequeue` lub `Peek` na pustej kolejce.

Po zaimplementowaniu klasy przygotuj prosty program testowy, który demonstruje poprawność działania kolejki poprzez dodanie kilku elementów o różnych priorytetach oraz ich pobieranie w odpowiedniej kolejności.

Zadanie 6.3. Harmonogram zadań z priorytetem i czasem dodania

Napisz program, który realizuje harmonogram zadań z wykorzystaniem kolejki priorytetowej.

- a) Zdefiniuj klasę `Task` zawierającą właściwości: `Description` (string), `Priority` (int), `Timestamp` (DateTime).
- b) Zaimplementuj komparator `TaskPriorityComparer`, który:
 - o w pierwszej kolejności porównuje zadania po `Priority` (wyższy priorytet ma być obsłużony wcześniej),



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- w przypadku równego priorytetu porównuje po Timestamp (wcześniej dodane zadanie ma pierwszeństwo).
- c) Utwórz kolejkę `PriorityQueue<Task, Task>` z użyciem tego komparatora.
- d) Dodaj do kolejki co najmniej 5 różnych zadań.
- e) Pobieraj zadania metodą `Dequeue` aż do opróżnienia kolejki i wypisz na ekranie kolejność przetwarzania.

Zadanie 6.4. Obsługa zgłoszeń

Napisz program, który symuluje obsługę zgłoszeń z wykorzystaniem kolejki priorytetowej.

- a) Zdefiniuj klasę `Ticket`, która zawiera:
 - `Id` – identyfikator zgłoszenia (`int`),
 - `Description` – krótki opis zgłoszenia (`string`),
 - `Priority` – priorytet zgłoszenia (`int`).
- b) Utwórz kolejkę priorytetową, w której zgłoszenia są przetwarzane w kolejności wynikającej z wartości priorytetu (zgłoszenie o wyższym priorytecie jest obsługiwane jako pierwsze).
- c) Dodaj do kolejki co najmniej kilka zgłoszeń o różnych priorytetach.
- d) Pobieraj zgłoszenia z kolejki przy użyciu metody `Dequeue` i wyświetlaj informacje o obsługiwanym zgłoszeniu.

Zadanie 6.5. Scalanie k posortowanych tablic

Danych jest k tablic liczb całkowitych, z których każda jest posortowana niemalejąco. Tablice mogą mieć różne długości.

Napisz program, który połączy wszystkie te tablice w jedną tablicę wynikową posortowaną niemalejąco. Nie wolno sortować danych wynikowych ani łączyć wszystkich elementów w jedną tablicę przed przetwarzaniem.

Rozwiązanie musi wykorzystywać kolejkę priorytetową. W kolejce należy przechowywać bieżące najmniejsze elementy z tablic wejściowych. Po pobraniu elementu z tablicy T do kolejki należy dodać kolejny element z tej samej tablicy (jeśli istnieje).

Zadanie 6.6. Najmniejszy przedział dla k tablic

Danych jest k tablic liczb całkowitych, z których każda jest posortowana niemalejąco.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Należy wyznaczyć najmniejszy przedział domknięty $[L, R]$, taki że w każdej tablicy znajduje się co najmniej jeden element należący do tego przedziału.

Rozwiązanie należy oprzeć na kolejce priorytetowej, która umożliwia wybór najmniejszego z aktualnie rozważanych elementów.

Przykład: dla tablic $[1, 4, 7]$, $[2, 5]$, $[3, 6, 8]$ wynik to **$[1, 3]$** .

Pytania pomocnicze

- 1) Czym różni się kolejka priorytetowa od klasycznej kolejki typu FIFO?
- 2) Na czym polega zasada działania kolejki priorytetowej i w jaki sposób ustalana jest kolejność przetwarzania elementów?
- 3) Jakie podstawowe operacje realizuje kolejka priorytetowa?
- 4) Jaka jest różnica pomiędzy operacjami Peek oraz Dequeue?
- 5) Na czym polega operacja Enqueue w kolejce priorytetowej opartej na kopcu binarnym?
- 6) W jaki sposób zapewniona jest własność kopca po dodaniu lub usunięciu elementu?

Podsumowanie

W ramach laboratorium przedstawiono zasadę działania kolejki priorytetowej oraz omówiono jej implementację w języku C#. Zaprezentowano podstawowe operacje wykonywane na tej strukturze danych, takie jak dodawanie elementów (Enqueue), usuwanie elementu o najwyższym priorytecie (Dequeue) oraz podgląd elementu (Peek). Szczególną uwagę poświęcono implementacji kolejki priorytetowej z wykorzystaniem kopca binarnego, wraz z dokładnym omówieniem mechanizmów przesiewania w górę i w dół.

Literatura

- [1] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C., *Wprowadzenie do algorytmów*, PWN, Warszawa, 2023. s. 163-171.
- [2] Drozdek A., *C++. Algorytmy i struktury danych*, Helion, Gliwice 2004. s. 147.
- [3] Aho A., Hopcroft J., Ullman J., *Algorytmy i struktury danych*, Helion, Gliwice, 2003. s. 148-155.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Materiały do przedmiotu:

Algorytmy i struktury danych

Laboratorium nr 7

**Implementacja podstawowych algorytmów
grafowych**

Autor:

dr hab. Adam Kiersztyn
mgr Krystyna Kiersztyn



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Spis treści

Wprowadzenie, tematyka zajęć	128
Cel ćwiczenia/laboratorium	129
Instrukcja laboratoryjna/ćwiczeniowa	129
Ćwiczenia samodzielne	159
Pytania pomocnicze	160
Podsumowanie	161
Literatura	161



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wprowadzenie, tematyka zajęć

Graf jest strukturą danych składającą się z **wierzchołków** (ang. vertices) oraz **krawędzi** (ang. edges), które łączą te wierzchołki. W zależności od charakteru relacji pomiędzy elementami graf może być **skierowany** lub **nieskierowany**. W grafie skierowanym krawędzie mają określony kierunek, natomiast w grafie nieskierowanym relacja pomiędzy wierzchołkami jest symetryczna.

Dodatkowo grafy mogą być **ważone** lub **nieważone**. W grafach ważonych każdej krawędzi przypisana jest wartość liczbową określającą koszt przejścia, długość połączenia lub inną miarę zależną od modelowanego problemu.

W implementacjach grafy reprezentuje się najczęściej przy użyciu dwóch struktur:

- **macierzy sąsiedztwa**, która przechowuje informacje o istnieniu krawędzi pomiędzy parami wierzchołków w postaci tablicy dwuwymiarowej,
- **listy sąsiedztwa**, w której dla każdego wierzchołka przechowywana jest lista jego bezpośrednich sąsiadów.

W przypadku grafów rzadkich lista sąsiedztwa jest rozwiązaniem bardziej efektywnym pamięciowo, dlatego jest często wykorzystywana w implementacjach algorytmów grafowych.

Do podstawowych algorytmów operujących na grafach należą algorytmy przeszukiwania grafu, takie jak **DFS** (Depth First Search) oraz **BFS** (Breadth First Search). Pozwalają one na systematyczne odwiedzanie wierzchołków grafu i stanowią podstawę wielu bardziej złożonych algorytmów.

Algorytm DFS polega na przeszukiwaniu grafu w głąb – od aktualnego wierzchołka przechodzi się do jednego z jego nieodwiedzonych sąsiadów i kontynuuje eksplorację tak długo, jak to możliwe. Dopiero po wyczerpaniu wszystkich możliwości następuje powrót do poprzedniego wierzchołka. Do implementacji tego podejścia używany jest stos.

Z kolei BFS realizuje przeszukiwanie grafu wszerz. Algorytm odwiedza najpierw wszystkie wierzchołki znajdujące się na tym samym poziomie od wierzchołka startowego, a dopiero następnie przechodzi do kolejnych poziomów grafu. Do realizacji tej strategii wykorzystywana jest kolejka.

W przypadku grafów ważonych jednym z najczęściej stosowanych algorytmów jest **algorytm Dijkstry**, który umożliwia wyznaczenie najkrótszych ścieżek od wybranego wierzchołka początkowego do wszystkich pozostałych wierzchołków grafu, pod warunkiem że wagi krawędzi są nieujemne.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Innym ważnym zagadnieniem związanym z grafami ważonymi nieskierowanymi jest wyznaczanie **minimalnego drzewa rozpinającego** (Minimum Spanning Tree – MST). Drzewo rozpinające jest podgrafem grafu, który zawiera wszystkie jego wierzchołki oraz minimalną liczbę krawędzi potrzebną do zachowania spójności, przy czym nie zawiera cykli. Minimalne drzewo rozpinające to takie drzewo, dla którego suma wag wszystkich krawędzi jest najmniejsza spośród wszystkich możliwych drzew rozpinających w danym grafie.

Do najczęściej stosowanych algorytmów wyznaczania minimalnego drzewa rozpinającego należą **algorytm Kruskala** oraz **algorytm Prima**, które różnią się strategią wyboru kolejnych krawędzi, jednak prowadzą do tego samego celu – znalezienia drzewa o minimalnym łącznym koszcie.

Cel ćwiczenia/laboratorium

Po zakończeniu laboratorium student:

- rozumie pojęcie grafu,
- rozróżnia grafy skierowane i nieskierowane oraz grafy ważne i nieważne,
- zna podstawowe sposoby reprezentacji grafów,
- rozumie zasadę działania algorytmów przeszukiwania grafu DFS oraz BFS,
- rozumie działanie algorytmu Dijkstry,
- rozumie pojęcie minimalnego drzewa rozpinającego (MST),
- potrafi zaimplementować podstawowe algorytmy grafowe w wybranym języku programowania.

Instrukcja laboratoryjna/ćwiczeniowa

Przykład 7.1. Przeszukiwanie grafu wszerz (BFS)

Algorytm BFS polega na odwiedzaniu wierzchołków grafu według rosnącej odległości od wierzchołka startowego. Najpierw analizowane są wszystkie wierzchołki bezpośrednio połączone z wierzchołkiem startowym, następnie wierzchołki oddalone o dwa kroki, a potem kolejne poziomy grafu.

Działanie algorytmu można opisać następująco:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



1. Wstawiamy wierzchołek startowy do kolejki.
2. Oznaczamy go jako odwiedzoney.
3. Dopóki kolejka nie jest pusta:
 - usuwamy pierwszy element z kolejki,
 - dopisujemy go do wyniku,
 - dla każdego jego sąsiada sprawdzamy, czy został już odwiedzony; jeżeli nie – oznaczamy go jako odwiedzonego i dodajemy na koniec kolejki.

Algorytm P.71. BFS

Dane:

$G = (V, E)$ – graf

$s \in V$ – wierzchołek początkowy

Wynik: kolejność odwiedzania wierzchołków grafu

BFS(G, s)

1. dla każdego wierzchołka $v \in V$ wykonaj
2. $visited[v] = false$
3. utwórz pustą kolejkę Q
4. $visited[s] = true$
5. **ENQUEUE**(Q, s)
6. dopóki $Q \neq \emptyset$ wykonuj
7. $v = \text{DEQUEUE}(Q)$
8. wypisz v
9. dla każdego wierzchołka u sąsiadującego z v wykonaj
10. jeżeli $visited[u] == false$ wtedy
11. $visited[u] = true$
12. **ENQUEUE**(Q, u)

Rozważmy graf nieskierowany, nieważony o wierzchołkach $V = \{A, B, C, D, E, F\}$ i krawędziach $E = \{(A, B), (A, E), (A, D), (B, C), (B, E), (D, E), (E, F)\}$.

Na tej podstawie można określić listę sąsiedztwa:

- A: B,E,D
- B: A,C,E
- E: A,B,D,F
- D: A,E



Fundusze Europejskie
dla Rozwoju Społecznego



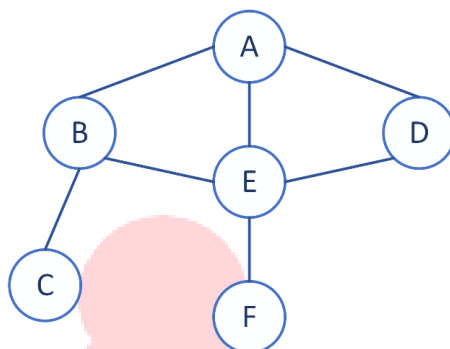
Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- C: B
- F: E

Wizualizację opisanego grafu przedstawia Rysunek 18.



Rysunek 18 Graf nieskierowany (źródło: opracowanie własne)

Objaśnienia do oznaczeń na grafie:



– wierzchołek nieodwiedzony,



– wierzchołek odkryty, ale jeszcze nie w pełni przetworzony,



– wierzchołek w pełni przetworzony.

Rozpoczynamy przeszukiwanie od wierzchołka **A**. Do kolejki dodajemy wierzchołek startowy **A**.



Oznaczamy **A** jako odwiedzony.



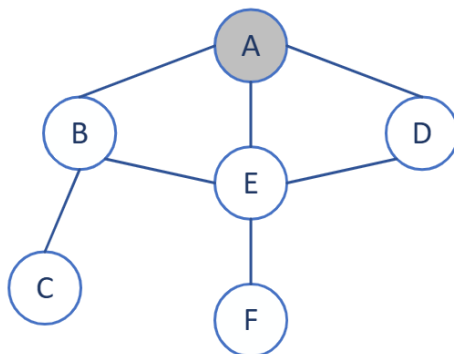
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

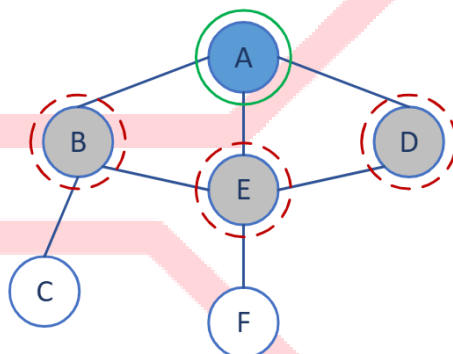
Dofinansowane przez
Unię Europejską





Krok 1

- Usuujemy pierwszy (w tym momencie jedyny) element z kolejki, czyli **A**.
- Dodajemy **A** do wyniku: **[A]**
- Sąsiedzi wierzchołka **A** to **[B, E, D]** – wszyscy są nieodwiedzani, zatem dodajemy ich do kolejki.



- Kolejka zawiera następujące wierzchołki:



- Odwiedzone wierzchołki: A, B, E, D

Krok 2

- Usuujemy pierwszy element z kolejki, czyli **B**
- Dodajemy wierzchołek B do wyniku: **[A, B]**
- Sąsiedzi wierzchołka **B** to **[A, C, E]**:
 - A już odwiedzony,
 - E w kolejce (oznaczony jako odwiedzony),



Fundusze Europejskie
dla Rozwoju Społecznego

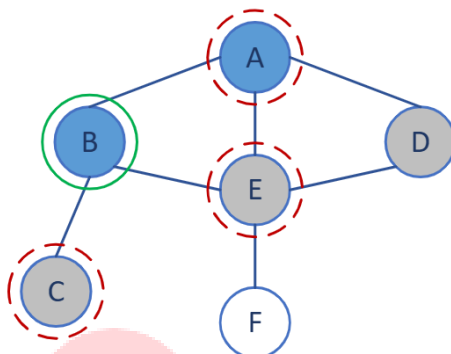


Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- C jest nowy zatem dodajemy do kolejki.



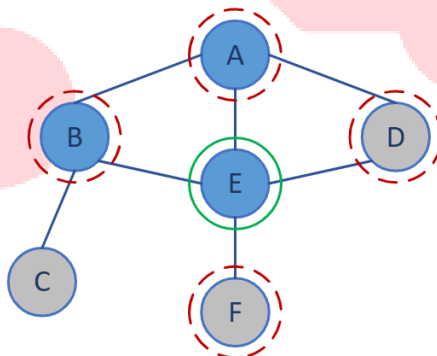
- Kolejka:



- Odwiedzone wierzchołki: A, B, E, D, C

Krok 3

- Usuwamy pierwszy element z kolejki, czyli **E**
- Dodajemy wierzchołek **E** do wyniku: **[A, B, E]**
- Sąsiedzi wierzchołka **E** to **[A, B, D, F]**:
 - A, B – już odwiedzone,
 - D – już w kolejce,
 - F – nowy, więc dodajemy do kolejki.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



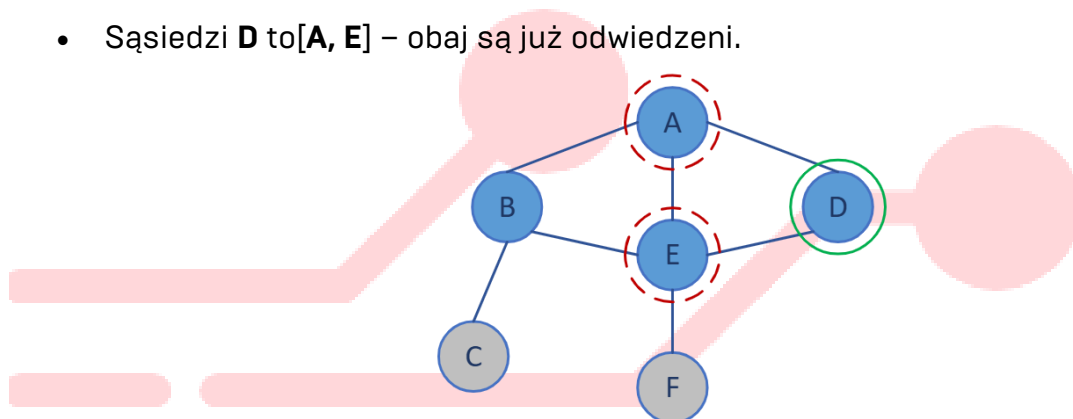
- Kolejka:



- Odwiedzone wierzchołki: A, B, E, D, C, F

Krok 4

- Usuwamy pierwszy element z kolejki, czyli **D**
- Dodajemy wierzchołek **E** do wyniku: **[A, B, E, D]**
- Sąsiedzi **D** to **[A, E]** – obaj są już odwiedzeni.

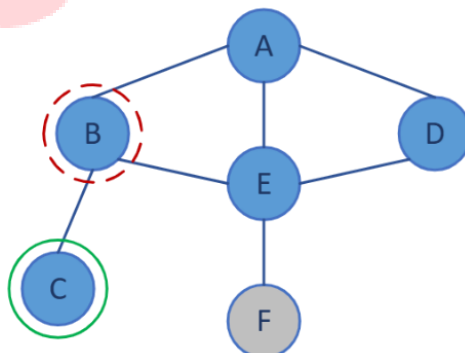


- Kolejka:



Krok 5

- Usuwamy pierwszy element z kolejki, czyli **C**
- Dodajemy wierzchołek **C** do wyniku: **[A, B, E, D, C]**
- Sąsiad wierzchołka **C** to **[B]** – już odwiedzony.

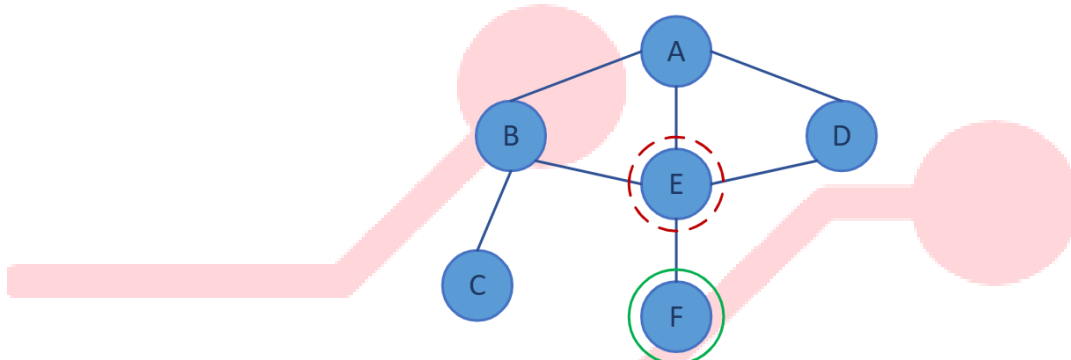


- Kolejka:



Krok 6

- Usuwamy pierwszy (i ostatni już) element z kolejki, czyli **F**
- Dodajemy wierzchołek **F** do wyniku: **[A, B, E, D, C, F]**
- Sąsiad wierzchołka **F** to **[E]** – już odwiedzony.



- Kolejka jest pusta, więc kończymy algorytm.

Wynik BFS: $A \rightarrow B \rightarrow E \rightarrow D \rightarrow C \rightarrow F$.

Przykład 7.2. Przeszukiwanie grafu w głąb (DFS)

Algorytm DFS polega na przeszukiwaniu grafu w głąb, zaczynając od wierzchołka startowego. W każdym kroku algorytm przechodzi do pierwszego nieodwiedzonego sąsiada aktualnego wierzchołka i kontynuuje eksplorację tak długo, jak jest to możliwe. Gdy dla danego wierzchołka nie ma już nieodwiedzonych sąsiadów, następuje powrót do poprzedniego wierzchołka i dalsze przeszukiwanie pozostałych gałęzi grafu. Dzięki temu algorytm eksploruje jedną ścieżkę do maksymalnej możliwej głębokości, zanim zacznie analizować inne możliwe kierunki. Algorytm DFS może być implementowany zarówno w sposób rekurencyjny, jak i iteracyjny.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Algorytm P.7.2.A. DFS – rekurencyjnie**Dane:** $G = (V, E)$ – graf $s \in V$ – wierzchołek początkowy**Wynik:** kolejność odwiedzania wierzchołków grafuDFS(G, s)

1. dla każdego wierzchołka $v \in V$ wykonaj
2. $visited[v] = false$
3. DFS_VISIT(s)

DFS_VISIT(v)

4. $visited[v] = true$
5. wypisz v
6. dla każdego wierzchołka u sąsiadującego z v wykonaj
7. jeżeli $visited[u] = false$ wtedy
8. DFS_VISIT(u)

Algorytm P.7.2.B. DFS – iteracyjnie**Dane:** $G = (V, E)$ – graf $s \in V$ – wierzchołek początkowy**Wynik:** kolejność odwiedzania wierzchołków grafuDFS(G, s)

1. dla każdego wierzchołka v w G
2. $visited[v] = false$
3. utwórz pusty stos S
4. $visited[s] = true$
5. push(S, s)
6. dopóki S nie jest pusty
7. $v = pop(S)$
8. wypisz v
9. dla każdego sąsiada u w Adj[v]
10. jeżeli $visited[u] = false$
11. $visited[u] = true$
12. push(S, u)

Po omówieniu zasady działania algorytmu DFS można prześledzić jego wykonanie na konkretnym przykładzie. W przedstawionym przykładzie przeszukiwanie realizowane jest w sposób iteracyjny, z wykorzystaniem stosu.

Fundusze Europejskie
dla Rozwoju SpołecznegoRzeczpospolita
PolskaDofinansowane przez
Unię Europejską

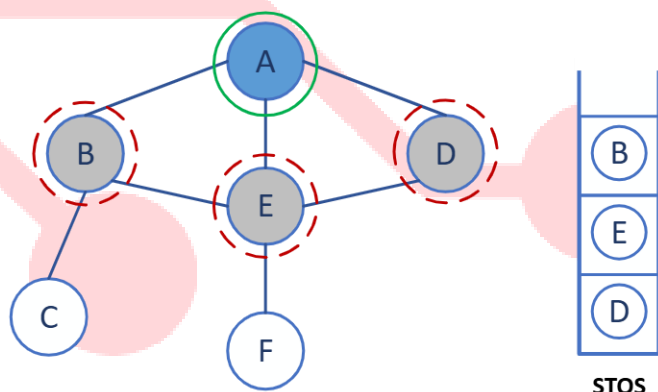
1. Na stos dodawany jest wierzchołek startowy.
2. Dopóki stos nie jest pusty:
 - zdejmujemy wierzchołek znajdujący się na szczycie stosu,
 - dodajemy go do wyniku,
 - przeglądamy jego sąsiadów,
 - jeśli dany sąsiad nie został jeszcze odwiedzony:
 - oznaczamy go jako odwiedzonego,
 - dodajemy go na stos (zwykle w odwróconej kolejności).

Dalsze rozważenia przeprowadzimy na grafie z **Przykładu 7.1**.

Dodajemy wierzchołek startowy **A** na stos.

Krok 1

- Zdejmujemy wierzchołek ze stosu: **A**
- Oznaczamy A jako „odwiedzony”
- Wynik: [**A**]
- Sąsiedzi wierzchołka **A** to [**B, E, D**] – wszyscy nieodwiedzieni, więc dodajemy na stos (od końca)



Krok 2

- Zdejmujemy wierzchołek ze stosu: **B**
- Wynik: [**A, B**]
- Sąsiedzi wierzchołka **B** to [**A, C, E**]:



Fundusze Europejskie
dla Rozwoju Społecznego

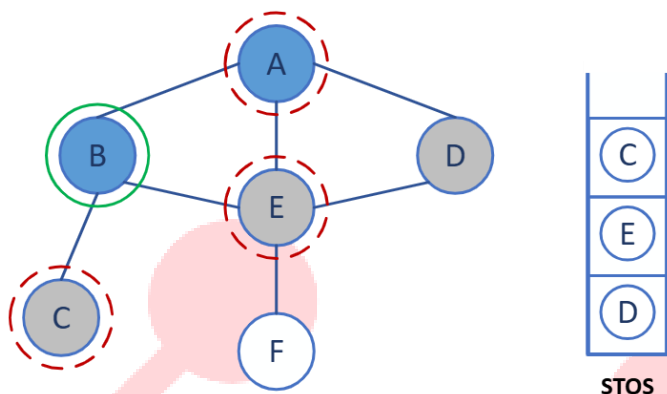


Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską

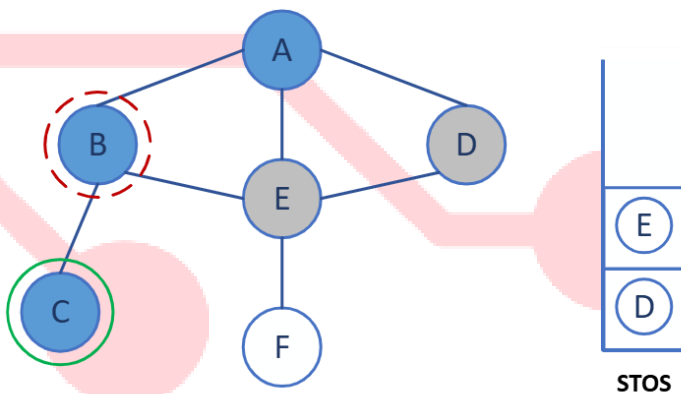


- A – już odwiedzony
- C – nowy, więc dodajemy na stos
- E – już odwiedzony



Krok 3

- Zdejmujemy wierzchołek ze stosu: **C**
- Wynik: **[A, B, C]**
- Sąsiad wierzchołka **C** to **[B]** – już odwiedzony



Krok 4

- Zdejmujemy wierzchołek ze stosu: **E**
- Wynik: **[A, B, C, E]**
- Sąsiedzi wierzchołka **E** to **[A, B, D, F]**:
 - A – już odwiedzony



Fundusze Europejskie
dla Rozwoju Społecznego

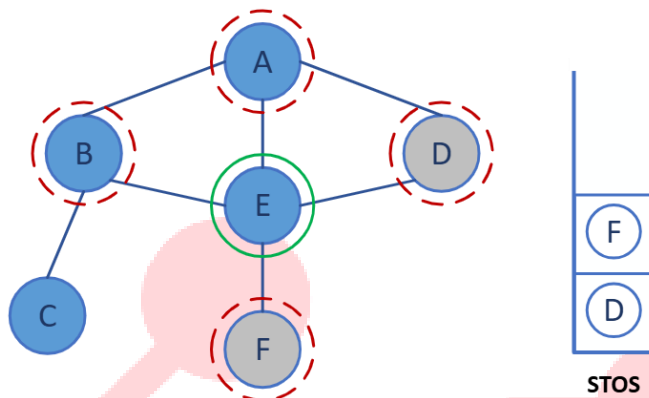


Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską

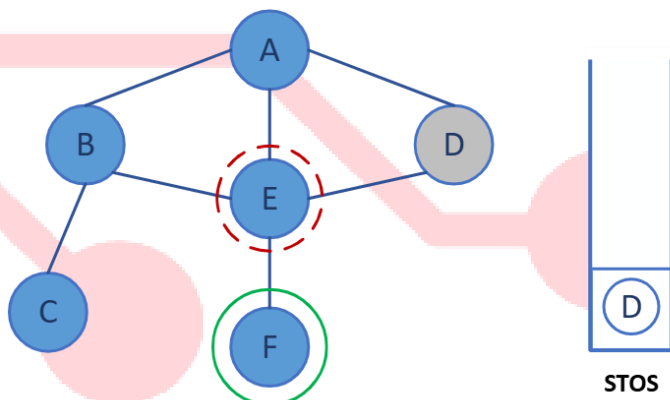


- B – już odwiedzony
- D – już odwiedzony
- F – nowy, więc dodajemy na stos



Krok 5

- Zdejmujemy wierzchołek ze stosu: **F**
- Wynik: **[A, B, C, E, F]**
- Sąsiad wierzchołka **F** to **[E]** – już odwiedzony



Krok 6

- Zdejmujemy wierzchołek ze stosu: **D**
- Wynik: **[A, B, C, E, F, D]**
- Sąsiedzi wierzchołka **D** to **[A, E]** – oba wierzchołki są już odwiedzane



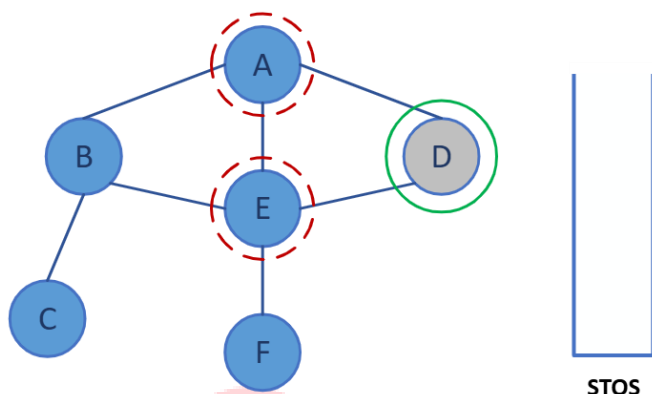
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską





- Stos jest pusty. Algorytm kończy działanie.

Wynik DFS: Kolejność odwiedzania wierzchołków: $A \rightarrow B \rightarrow C \rightarrow E \rightarrow F \rightarrow D$.

Przykład 7.3. Przeszukiwanie grafów skierowanych

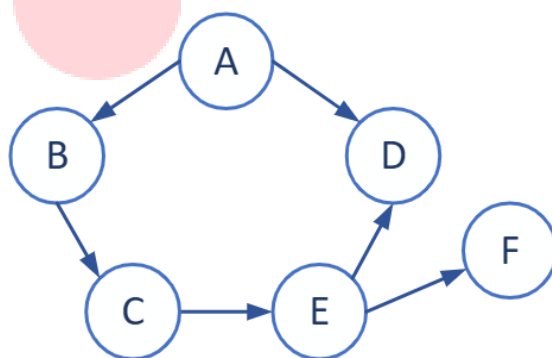
W przypadku grafów skierowanych każda krawędź posiada określony kierunek. Oznacza to, że przejście między wierzchołkami jest możliwe tylko w kierunku wskazanym przez krawędź. Podczas przeszukiwania grafu (np. algorytmami BFS lub DFS) analizowane są wyłącznie krawędzie wychodzące z aktualnego wierzchołka. W praktyce oznacza to, że w reprezentacji grafu, np. w postaci listy sąsiedztwa, zapisuje się jedynie wierzchołki osiągalne z danego wierzchołka, a samo przeszukiwanie przebiega analogicznie jak w grafie nieskierowanym.

Rozważmy graf skierowany $G = (V, E)$, którego wierzchołki oraz krawędzie są zdefiniowane następująco:

$$V = \{A, B, C, D, E, F\}$$

$$E = \{(A, B), (A, D), (B, C), (C, E), (E, D), (E, F)\}$$

Rysunek 19 przedstawia wizualizację opisanego grafu.



Rysunek 19 Graf skierowany [źródło: opracowanie własne]



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Analogicznie jak w przypadku grafu nieskierowanego, graf skierowany można przedstawić w postaci listy sąsiedztwa, przy czym uwzględnia się wyłącznie krawędzie wychodzące z danego wierzchołka.

Lista sąsiedztwa rozważanego grafu ma postać:

A: B, D
B: C
C: E
D: –
E: D, F
F: –

Przeszukiwanie grafu skierowanego za pomocą BFS

Tabela 2 przedstawia przebieg przeszukiwania grafu metodą BFS, rozpoczynając od wierzchołka A. W kolejnych krokach pokazano stan kolejki oraz aktualną kolejność odwiedzonych wierzchołków. Algorytm odwiedza wierzchołki poziomami, przechodząc najpierw do wszystkich sąsiadów danego wierzchołka.

Tabela 2 Przeszukiwanie grafu za pomocą BFS

Krok	Kolejka	Wynik
Start	A	[]
1	B, D	A
2	D, C	A, B
3	C	A, B, D
4	E	A, B, D, C
5	F	A, B, D, C, E
6	[]	A, B, D, C, E, F

Przeszukiwanie grafu skierowanego za pomocą DFS

Tabela 3 przedstawia przebieg przeszukiwania grafu metodą DFS, rozpoczynając od wierzchołka A. W tabeli pokazano stan stosu oraz kolejność odwiedzonych wierzchołków. Algorytm zagłębia się w graf, odwiedzając kolejne wierzchołki wzdłuż jednej ścieżki, aż do wyczerpania możliwości dalszego przejścia.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Tabela 3 Przeszukiwanie grafu za pomocą DFS

Krok	Stos	Wynik
Start	A	[]
1	D, B	A
2	D, C	A, B
3	D, E	A, B, C
4	D, F	A, B, C, E
5	D	A, B, C, E, F
6	[]	A, B, C, E, F, D

Przykład 7.4. Algorytm Dijkstry

Algorytm Dijkstry służy do wyznaczania najkrótszych ścieżek od wybranego wierzchołka startowego do wszystkich pozostałych wierzchołków w grafie ważonym, pod warunkiem że wagi krawędzi są nieujemne. W każdej iteracji algorytm wybiera wierzchołek o najmniejszej znanej odległości od wierzchołka startowego, a następnie próbuje poprawić odległości do jego sąsiadów. Proces ten nazywany jest **relaksacją krawędzi**.

Działanie algorytmu można przedstawić następująco:

1. Dla wszystkich wierzchołków ustawiamy początkową odległość równą nieskończoności.
2. Wierzchołkowi startowemu przypisujemy odległość równą 0.
3. Dopóki istnieją nieprzetworzone wierzchołki:
 - o wybieramy wierzchołek o najmniejszej aktualnej odległości,
 - o dla każdego jego sąsiada sprawdzamy, czy możliwe jest znalezienie krótszej ścieżki przez ten wierzchołek,
 - o jeżeli tak, aktualizujemy odległość.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Algorytm P.7.3. Algorytm Dijkstry

Dane:

$G = (V, E)$ – graf ważony

$s \in V$ – wierzchołek początkowy

Wynik: najkrótsze odległości od s do wszystkich wierzchołków

DIJKSTRA(G, s)

1. dla każdego $v \in V$:
2. $\text{dist}[v] = \infty$
3. $\text{visited}[v] = \text{false}$
4. $\text{dist}[s] = 0$
5. powtarzaj $|V|$ razy:
6. u = wierzchołek o najmniejszym $\text{dist}[u]$, dla którego $\text{visited}[u] = \text{false}$
7. $\text{visited}[u] = \text{true}$
8. dla każdego sąsiada v wierzchołka u :
9. jeżeli $\text{visited}[v] = \text{false}$ oraz $\text{dist}[u] + w(u,v) < \text{dist}[v]$:
10. $\text{dist}[v] = \text{dist}[u] + w(u,v)$

Objaśnienia

- $\text{dist}[v]$ – aktualna najkrótsza odległość od wierzchołka startowego do v ,
- $w(u, v)$ – waga krawędzi między u i v .

Rozważmy graf ważony skierowany $G = (V, E)$ o wierzchołkach $V = \{A, B, C, D, E\}$. Wagi krawędzi oznaczają koszt przejścia między wierzchołkami.



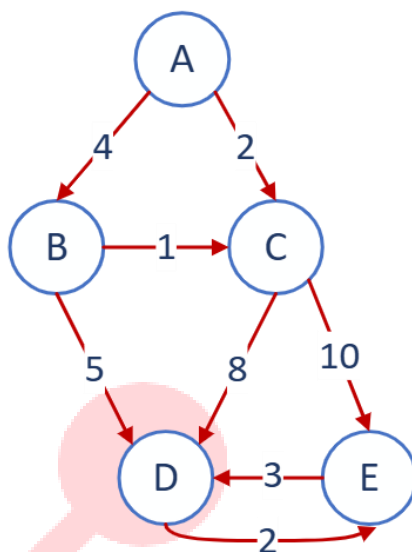
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską

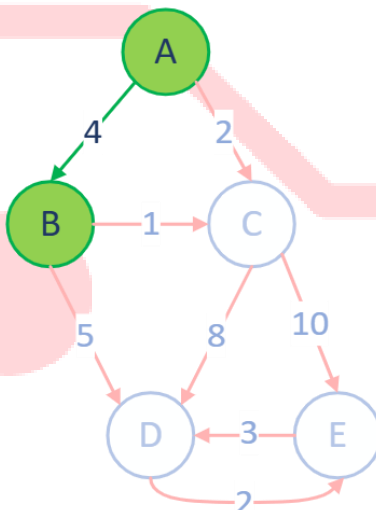




Algorytm rozpoczynamy od wierzchołka **A**. Przeanalizujemy sąsiadów **A**.

Krok 1 – aktualny wierzchołek **A**

Z wierzchołka **A** do **B** jest jedna ścieżka z wagą **4**. Zatem odległość do **B** wynosi **4**, $dist(A, B) = 4$.



Fundusze Europejskie
dla Rozwoju Społecznego

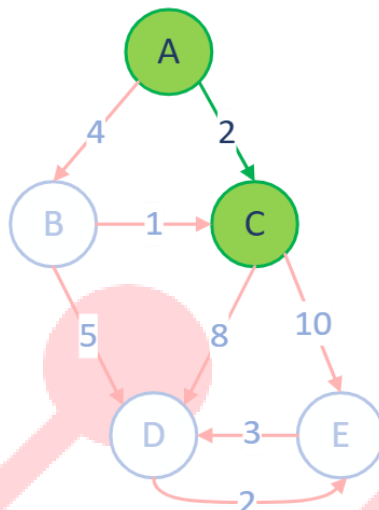


Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Z wierzchołka **A** do **C** ścieżka ma wagę **2**. Stąd odległość do **C** wynosi **2**, $dist(A, C) = 2$.

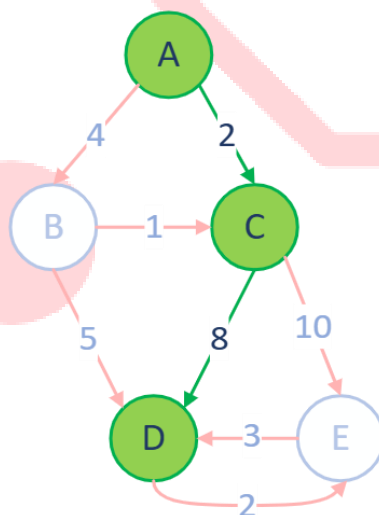


Krok 2 – aktualny wierzchołek C

Spośród nieprzetworzonych wierzchołków najmniejszą odległość ma teraz wierzchołek **C** [2]. Analizujemy sąsiadów C.

Z wierzchołka **C** do **D** ścieżka ma wagę **8**. Obliczamy odległość do **D** przez **C**:

$$dist(A, D) = dist(A, C) + dist(C, D) = 2 + 8 = 10$$



Z wierzchołka **C** do **E** ścieżka ma wagę **10**. Nowa odległość do **E** przez **C** wynosi:

$$dist(A, E) = dist(A, C) + dist(C, E) = 2 + 10 = 12$$



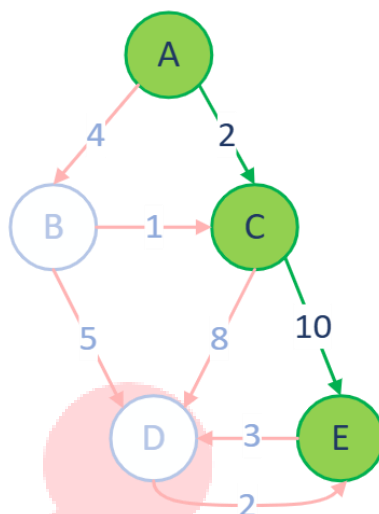
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską





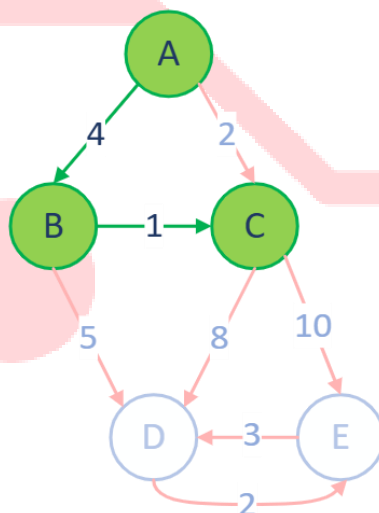
Krok 3 – aktualny wierzchołek B

Teraz najmniejszą odległość spośród nieprzetworzonych wierzchołków ma **B** [4]. Przeanalizujemy sąsiadów **B**.

Z wierzchołka **B** do **C** ścieżka ma wagę 1. Nowa odległość do **C** przez **B** wynosi:

$$\text{dist}(A, C) = \text{dist}(A, B) + \text{dist}(B, C) = 4 + 1 = 5$$

Obecna odległość do **C** wynosi 2, więc **nie aktualizujemy wartości**.



Z wierzchołka **B** do **D** ścieżka ma wagę 5. Nowa odległość do **D** przez **B** wynosi:

$$\text{dist}(A, D) = \text{dist}(A, B) + \text{dist}(B, D) = 4 + 5 = 9$$



Fundusze Europejskie
dla Rozwoju Społecznego

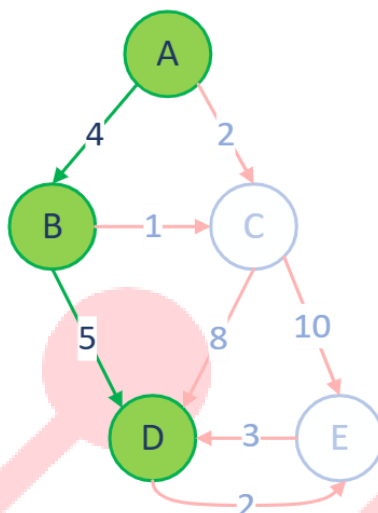


Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Obecna odległość do **D** wynosi **10**, a ponieważ **9 < 10**, aktualizujemy odległość do **D** na **9**.



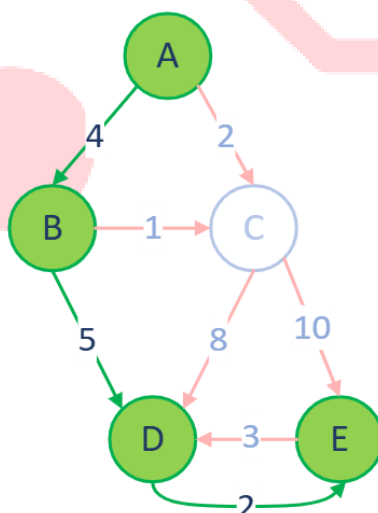
Krok 4 – aktualny wierzchołek **D**

Najmniejszą odległość spośród pozostałych wierzchołków ma teraz **D** [9].
Przeanalizujemy sąsiada **D**.

Z wierzchołka **D** do **E** ścieżka ma wagę **2**. Nowa odległość do **E** przez **D** wynosi:

$$\text{dist}(A, E) = \text{dist}(A, D) + \text{dist}(D, E) = 9 + 2 = 11$$

Obecna odległość do **E** wynosi **12**, a ponieważ **11 < 12**, aktualizujemy odległość do **E** na **11**.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Krok 5 – aktualny wierzchołek E

Pozostał wierzchołek **E** z odległością **11**. Analizujemy jego sąsiada.

Z wierzchołka **E** do **D** ścieżka ma wagę **3**. Nowa odległość do **D** przez **E** wynosi:

$$\text{dist}(A, D) = \text{dist}(A, E) + \text{dist}(D, E) = 11 + 3 = 14$$

Obecna odległość do **D** wynosi **9**, więc nie dokonujemy aktualizacji.

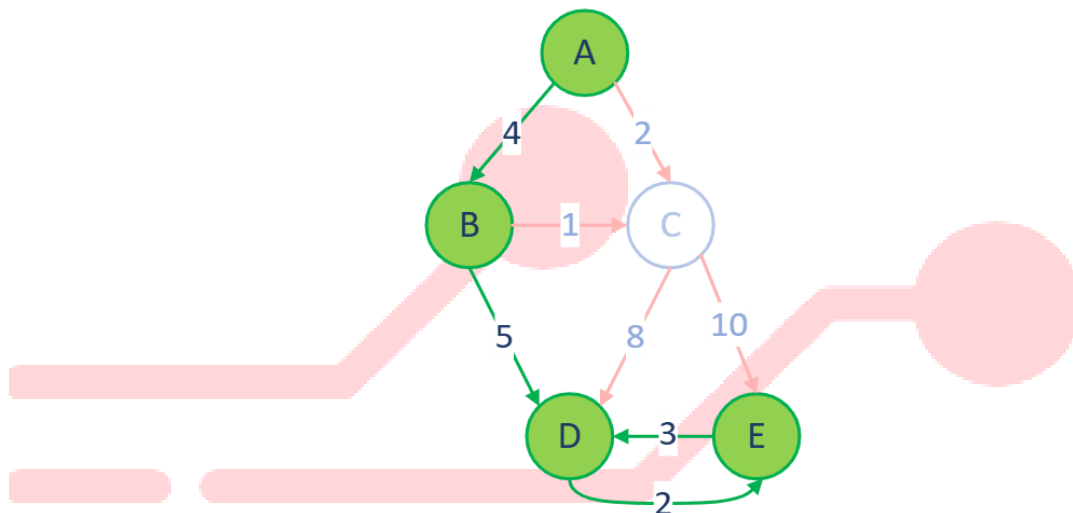


Tabela 4 przedstawia ostateczne najkrótsze odległości od wierzchołka A do pozostałych wierzchołków grafu wraz z odpowiadającymi im ścieżkami.

Tabela 4 Ostateczne najkrótsze odległości od wierzchołka A

Wierzchołek	Najkrótsza odległość	Ścieżka
A	0	-
B	4	A → B
C	2	A → C
D	9	A → B → D
E	11	A → B → D → E



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Przykład 7.5. Algorytm Kruskala

Algorytm **Kruskala** służy do wyznaczania **minimalnego drzewa rozpinającego (MST)** w grafie nieskierowanym ważonym. Jego celem jest wybranie takiego podzbioru krawędzi, który:

- łączy wszystkie wierzchołki,
- nie zawiera cykli,
- ma minimalną sumę wag.

Algorytm działa w sposób zachłanny (greedy) – w każdej iteracji wybiera „najtańszą” możliwą krawędź, która nie tworzy cyklu z już wybranymi krawędziami.

Kroki algorytmu

1. Posortuj wszystkie krawędzie grafu rosnąco według wag.
2. Utwórz zbiór (las), w którym każdy wierzchołek stanowi osobne drzewo.
3. Przechodź po kolejnych krawędziach (od najmniejszej wagi):
 - jeżeli krawędź łączy dwa różne drzewa (nie tworzy cyklu) to dodaj ją do MST,
 - w przeciwnym przypadku pomini ją.
4. Zatrzymaj się, gdy liczba krawędzi w MST wynosi $|V| - 1$.

Wykrywanie cyklu

Aby sprawdzić, czy dodanie krawędzi tworzy cykl, wykorzystuje się strukturę danych **Union-Find (Disjoint Set Union)**.

Dla każdej krawędzi(u, v):

- sprawdzamy, czy u i v należą do tego samego zbioru,
 - jeżeli tak – powstałby cykl, pomijamy krawędź,
 - jeżeli nie – łączymy zbiory i dodajemy krawędź do MST.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Algorytm P.7.4. Algorytm Kruskala**Dane:** $G = (V, E)$ – graf nieskierowany ważony**Wynik:** zbiór krawędzi tworzących minimalne drzewo rozpinające (MST)**KRUSKAL(G)**

1. utwórz pusty zbiór MST
2. dla każdego wierzchołka $v \in V$ wykonaj
3. MAKE-SET(v)
4. posortuj krawędzie E rosnąco według wag
5. dla każdej krawędzi $(u, v) \in E$ wykonuj
6. jeżeli FIND-SET(u) $\neq$ FIND-SET(v) wtedy
7. dodaj krawędź (u, v) do MST
8. UNION(u, v)
9. zwróć MST

Objaśnienia operacji

- MAKE-SET(v) – tworzy zbiór zawierający jeden element v ,
- FIND-SET(v) – zwraca reprezentanta zbioru, do którego należy v ,
- UNION(u, v) – łączy dwa zbiory zawierające u i v ,
- MST – zbiór krawędzi tworzących minimalne drzewo rozpinające.

Po omówieniu zasady działania algorytmu Kruskala można prześledzić jego wykonanie na konkretnym przykładzie. Rozważmy graf nieskierowany ważony, w którym każdej krawędzi przypisana jest określona waga. Celem algorytmu jest wyznaczenie minimalnego drzewa rozpinającego, czyli takiego zbioru krawędzi, który łączy wszystkie wierzchołki grafu przy minimalnym łącznym koszcie.

Rozważmy graf nieskierowany ważony (patrz Rysunek 20).



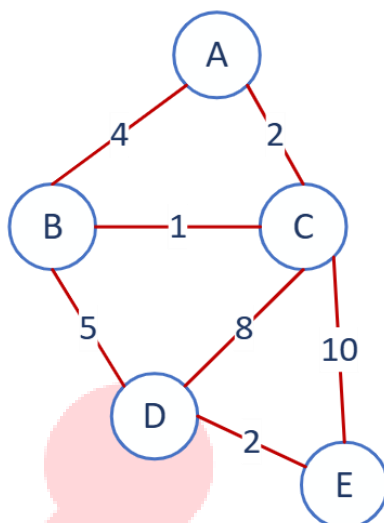
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską





Rysunek 20 Graf nieskierowany ważony

Krok 1 – sortowanie krawędzi

Krawędź	Waga
BC	1
AC	2
DE	2
AB	4
BD	5
CD	8
CE	10

Krok 2 – budowa drzewa

Po posortowaniu krawędzi według rosnących wag algorytm Kruskala analizuje je kolejno, zaczynając od najtańszych. Do budowanego drzewa dodawane są tylko te krawędzie, które nie tworzą cyklu.

Krok 2.1. Dodanie krawędzi BC

Najmniejszą wagę ma krawędź **BC** o koszcie 1. Na początku żaden z wierzchołków nie jest jeszcze połączony z innymi, dlatego dodanie tej krawędzi nie powoduje powstania cyklu.

Po dodaniu krawędzi otrzymujemy pierwsze poddrzewo: {B, C}.

Aktualny koszt drzewa: 1.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Krok 2.2. Dodanie krawędzi AC

Kolejną krawędzią jest **AC** o koszcie **2**. Wierzchołek **A** nie należy jeszcze do żadnego poddrzewa, natomiast **C** należy do poddrzewa {B,C}.. Dodanie tej krawędzi łączy wierzchołek **A** z istniejącym poddrzewem i nadal nie tworzy cyklu.

Po dodaniu krawędzi otrzymujemy poddrzewo: {A, B, C}.

Aktualny koszt drzewa: $1 + 2 = 3$

Krok 2.3. Dodanie krawędzi DE

Następna krawędź to **D-E** o koszcie **2**. Wierzchołki **D** i **E** nie są jeszcze połączone z żadnym z wcześniej utworzonych poddrzew, dlatego dodanie tej krawędzi nie powoduje powstania cyklu.

Powstaje drugie, niezależne poddrzewo: {D, E}.

Aktualny koszt drzewa: $1 + 2 + 2 = 5$.

Krok 2.4. Pominięcie krawędzi AB

Kolejna krawędź to **AB** o koszcie **4**. Oba wierzchołki należą już do tego samego poddrzewa {A, B, C}.

Dodanie krawędzi **AB** spowodowałoby powstanie cyklu:

$$A \rightarrow C \rightarrow B \rightarrow A$$

Dlatego ta krawędź zostaje **pominięta**.

Krok 2.5. Dodanie krawędzi BD

Następna analizowana krawędź to **BD** o koszcie **5**. Wierzchołek **B** należy do poddrzewa {A, B, C}, natomiast **D** należy do poddrzewa {D, E}.

Dodanie tej krawędzi nie tworzy cyklu, ponieważ łączy dwa różne poddrzewa. W efekcie wszystkie wierzchołki zostają połączone w jedno drzewo.

Po dodaniu krawędzi otrzymujemy {A, B, C, D, E}.

Aktualny koszt drzewa: $1 + 2 + 2 + 5 = 10$

Po dodaniu $|V| - 1 = 4$ krawędzi otrzymujemy MST.

Wynik

- Krawędzie drzewa: BC, AC, DE, BD.
- Łączny koszt MST wynosi: 10



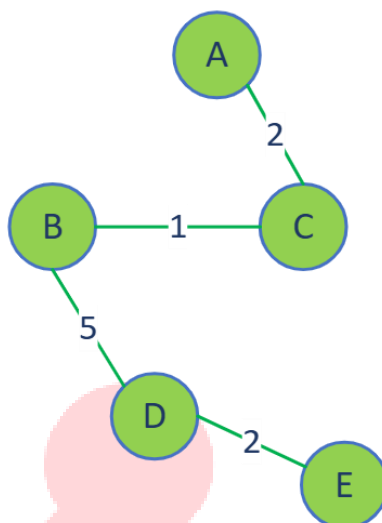
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską





Rysunek 21 Minimalne drzewo rozpinające uzyskane za pomocą algorytmu Kruskala [źródło: opracowanie własne]

Przykład 7.6. Algorytm Prima

Algorytm **Prima** służy do wyznaczania **minimalnego drzewa rozpinającego (MST)** w grafie **nieskierowanym ważonym**. Jego celem jest znalezienie takiego zbioru krawędzi, który łączy wszystkie wierzchołki grafu, nie tworzy cykli i ma możliwie najmniejszą sumę wag.

W odróżnieniu od algorytmu Kruskala, który analizuje krawędzie globalnie w kolejności rosnących wag, algorytm Prima buduje drzewo **stopniowo**, rozpoczynając od wybranego wierzchołka startowego. W każdej iteracji do aktualnie budowanego drzewa dołączany jest nowy wierzchołek za pomocą krawędzi o najmniejszej wadze spośród tych, które łączą drzewo z jeszcze nieodwiedzoną częścią grafu.

Na początku drzewo zawiera tylko jeden wierzchołek. Następnie wybierana jest najtańsza krawędź wychodząca z tego drzewa do wierzchołka, który jeszcze do niego nie należy. Po dodaniu tej krawędzi nowy wierzchołek staje się częścią drzewa, a proces jest powtarzany aż do chwili, gdy wszystkie wierzchołki grafu zostaną do niego włączone.

Ważną cechą algorytmu Prima jest to, że w każdej iteracji rozważane są wyłącznie te krawędzie, które mają jeden koniec w już zbudowanym drzewie, a drugi poza nim. Dzięki temu algorytm nie tworzy cykli, ponieważ do drzewa dołączany jest zawsze dokładnie jeden nowy wierzchołek.

Algorytm Prima jest algorytmem zachłannym, ponieważ w każdym kroku dokonuje lokalnie najlepszego wyboru, czyli wybiera najtańszą możliwą krawędź rozszerzającą



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



aktualne drzewo. Mimo tej lokalnej strategii prowadzi on do otrzymania poprawnego minimalnego drzewa rozpinającego. W praktyce algorytm Prima implementuje się z użyciem kolejki priorytetowej, co pozwala efektywnie wybierać najtańszą krawędź w każdej iteracji.

Algorytm P.7.5. Algorytm Prima

Dane:

$G = (V, E)$ – graf nieskierowany ważony

$s \in V$ – wierzchołek startowy

Wynik: zbiór krawędzi tworzących minimalne drzewo rozpinające (MST)

PRIM(G, s)

1. dla każdego wierzchołka $v \in V$ wykonaj
2. $\text{key}[v] = \infty$
3. $\text{parent}[v] = \text{null}$
4. $\text{key}[s] = 0$
5. utwórz kolejkę priorytetową Q zawierającą wszystkie wierzchołki V
6. dopóki $Q \neq \emptyset$ wykonuj
7. $u = \text{EXTRACT-MIN}(Q)$
8. dla każdego wierzchołka v sąsiadującego z u wykonaj
9. jeżeli $v \in Q$ oraz $w(u, v) < \text{key}[v]$ wtedy
10. $\text{parent}[v] = u$
11. $\text{key}[v] = w(u, v)$
12. zwróć zbiór krawędzi $(v, \text{parent}[v])$ dla wszystkich $v \neq s$

Objaśnienia

- $\text{key}[v]$ – najmniejsza znana waga krawędzi łączącej v z budowanym drzewem,
- $\text{parent}[v]$ – poprzednik w drzewie MST,
- $\text{EXTRACT-MIN}(Q)$ – zwraca wierzchołek o najmniejszej wartości key ,
- $w(u, v)$ – waga krawędzi między u i v ,

W celu porównania działania algorytmu Prima z algorytmem Kruskala, rozważymy ten sam graf nieskierowany ważony, który został wykorzystany w poprzednim przykładzie.



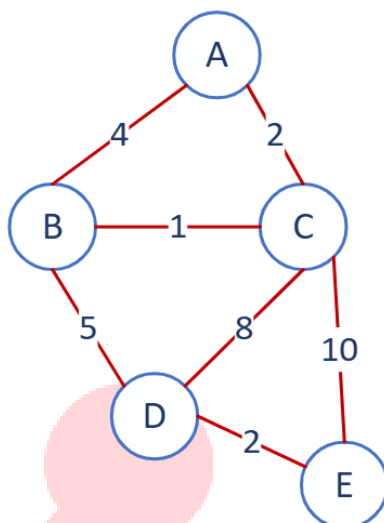
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską





Dzięki temu możliwe będzie zaobserwowanie różnic w sposobie budowy minimalnego drzewa rozpinającego.

Założmy startujemy z **A**. Na początku drzewo zawiera tylko wierzchołek startowy:

$$T = \{A\}$$

Krok 1

Rozpatrujemy wszystkie krawędzie wychodzące z wierzchołka **A**:

- AC ma wagę 2
- AB ma wagę 4

Wybieramy krawędź o najmniejszej wadze, czyli AC [2], dodajemy ją do drzewa razem z wierzchołkiem **C**.

Aktualne drzewo:

$$T = \{A, C\}$$



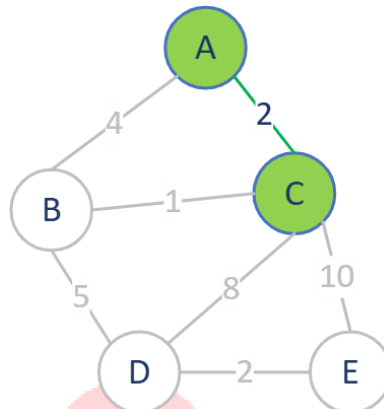
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską





Wybrane krawędzie: AC.

Łączny koszt: 2.

Krok 2

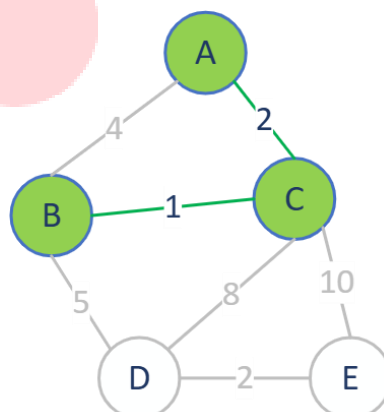
Rozpatrujemy wszystkie krawędzie, które łączą aktualne drzewo $\{A, C\}$ z pozostałymi wierzchołkami:

- AB – 4
- CB – 1
- CD – 8
- CE – 10

Najmniejszą wagę ma krawędź **CB** (1), dodajemy ją do drzewa razem z wierzchołkiem **B**.

Aktualne drzewo:

$$T = \{A, B, C\}$$



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wybrane krawędzie: AC, CB.

Łączny koszt: $2 + 1 = 3$.

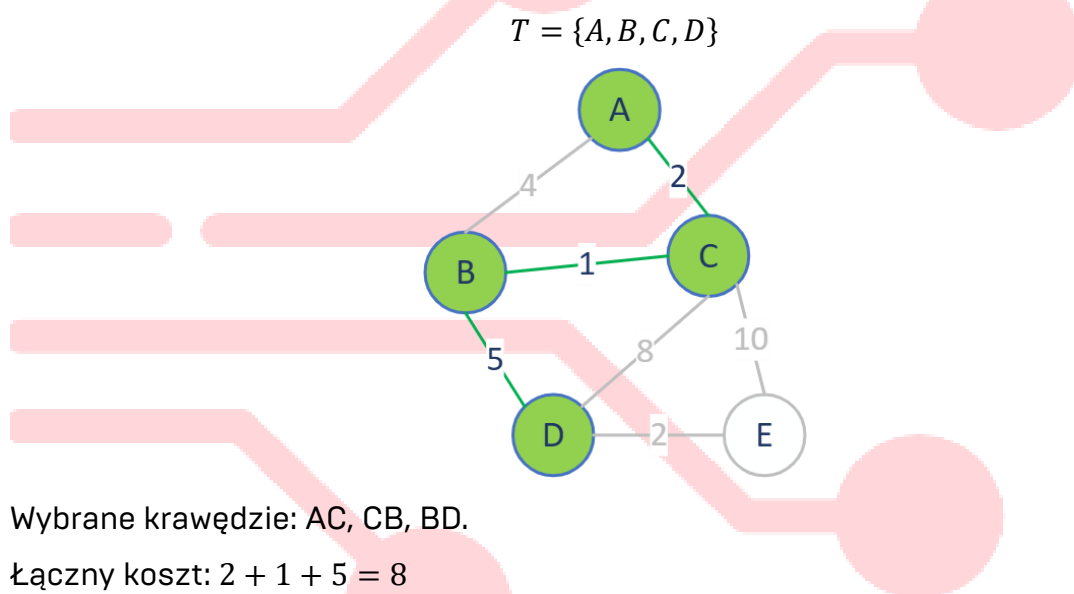
Krok 3

Rozpatrujemy krawędzie łączące drzewo $\{A, B, C\}$ z pozostałymi wierzchołkami:

- BD – 5
- CD – 8
- CE – 10

Najmniejszą wagę ma krawędź BD [5], dodajemy ją do drzewa razem z wierzchołkiem **D**.

Aktualne drzewo:



Krok 4

Rozpatrujemy krawędzie łączące drzewo $\{A, B, C, D\}$ z pozostałymi wierzchołkami:

- DE – 2
- CE – 10

Najmniejszą wagę ma krawędź DE [2]. Dodajemy ją do drzewa razem z wierzchołkiem **E**.

Aktualne drzewo:

$$T = \{A, B, C, D, E\}$$



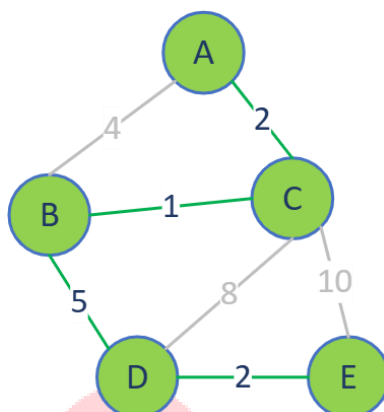
Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



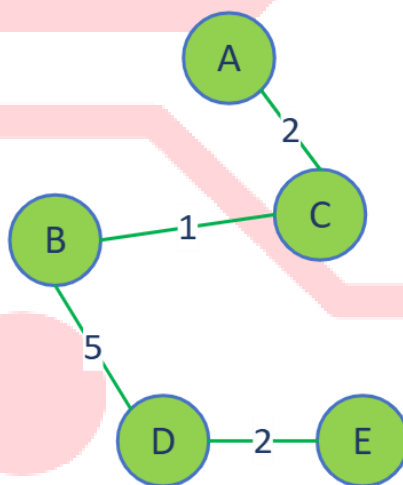


Wybrane krawędzie: AC, CB, BD, DE.

Łączny koszt: $2 + 1 + 5 + 2 = 10$

Wynik

- Minimalne drzewo rozpinające składa się z następujących krawędzi: AC, CB, BD, DE.
- Łączny koszt minimalnego drzewa rozpinającego wynosi: 10.



Rysunek 22 Minimalne drzewo rozpinające uzyskane za pomocą algorytmu Prima (źródło: opracowanie własne)



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Ćwiczenia samodzielne

Zadanie 7.1 Przeglądanie grafu wszerz

Napisz program implementujący algorytm BFS dla grafu reprezentowanego za pomocą listy sąsiedztwa. Program powinien:

- rozpoczynać przeszukiwanie od wskazanego wierzchołka,
- wypisywać kolejność odwiedzania wierzchołków.

Przetestuj działanie programu dla przykładowego grafu.

Zadanie 7.2 Przeglądanie grafu w głąb

Napisz program implementujący algorytm DFS dla grafu reprezentowanego za pomocą listy sąsiedztwa. Wykonaj dwie wersje:

- rekurencyjną,
- iteracyjną z wykorzystaniem stosu.

Porównaj otrzymane wyniki dla tego samego grafu.

Zadanie 7.3. BFS i DFS dla grafu skierowanego

Zmodyfikuj programy z poprzednich zadań tak, aby działały również dla grafu skierowanego.

Zadanie 7.4. Sprawdzanie spójności grafu

Napisz program, który dla zadanego grafu nieskierowanego sprawdzi, czy graf jest spójny. Wykorzystaj algorytm BFS lub DFS.

Wskazówka: Jeżeli wszystkie wierzchołki zostaną odwiedzone, graf jest spójny, w przeciwnym przypadku graf jest niespójny.

Zadanie 7.5. Implementacja algorytmu Dijkstry

Zaimplementuj algorytm Dijkstry dla grafu ważonego. Program powinien:

- obliczać najkrótsze odległości od wierzchołka startowego do wszystkich pozostałych,
- wypisywać wartości najkrótszych odległości,
- wypisywać przebieg najkrótszej ścieżki do wybranego wierzchołka.

Rozszerz implementację algorytmu Dijkstry o tablicę poprzedników, tak aby możliwe było wypisanie kolejnych wierzchołków najkrótszej ścieżki.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Zadanie 7.6. Najkrótsza droga

Rozważ graf ważony reprezentujący sieć dróg pomiędzy miastami. Wierzchołki grafu odpowiadają miastom, natomiast krawędzie oznaczają drogi pomiędzy nimi. Waga krawędzi określa długość drogi (np. w kilometrach).

Napisz program, który reprezentuje sieć dróg jako graf ważony, a dla wskazanego miasta początkowego wyznacza najkrótsze drogi do wszystkich pozostałych miast. Ponadto program powinien wypisać długość najkrótszej drogi i przebieg trasy (kolejne miasta na trasie).

Zadanie 7.7. Sieć połączeń pomiędzy miastami

Rozważ graf ważony nieskierowany reprezentujący sieć połączeń pomiędzy miastami. Wierzchołki grafu odpowiadają miastom, natomiast krawędzie oznaczają możliwe połączenia między nimi. Waga krawędzi określa koszt budowy połączenia.

Celem jest zbudowanie sieci połączeń, która łączy wszystkie miasta oraz ma minimalny całkowity koszt budowy.

Napisz program, który reprezentuje sieć połączeń jako graf ważony i wyznacza minimalne drzewo rozpinające grafu. Program powinien wypisać krawędzie należące do minimalnego drzewa rozpinającego oraz łączny koszt budowy sieci.

Pytania pomocnicze

- 1) Na czym polega różnica między algorytmami BFS i DFS?
- 2) Jakie struktury danych są wykorzystywane w algorytmach BFS i DFS?
- 3) W jaki sposób kolejność sąsiadów w liście sąsiedztwa wpływa na wynik działania algorytmu DFS?
- 4) Czy algorytm BFS zawsze znajduje najkrótszą ścieżkę w grafie? W jakim przypadku jest to prawdziwe?
- 5) Jak zmienia się działanie algorytmów BFS i DFS w przypadku grafu skierowanego?
- 6) Na czym polega algorytm Dijkstry i jakie jest jego zastosowanie?
- 7) Co oznacza pojęcie relaksacji krawędzi w algorytmie Dijkstry?
- 8) Dlaczego algorytm Dijkstry działa poprawnie tylko dla grafów z nieujemnymi wagami krawędzi?



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



9) Czym jest minimalne drzewo rozpinające (MST)?

Podsumowanie

W ramach laboratorium przedstawiono podstawowe algorytmy grafowe oraz ich implementacje. Algorytmy DFS i BFS służą do przeszukiwania grafu, jednak różnią się strategią odwiedzania wierzchołków. DFS eksploruje graf w głąb, natomiast BFS odwiedza wierzchołki poziomami.

Algorytm Dijkstry umożliwia wyznaczenie najkrótszych ścieżek w grafie ważonym, co ma szerokie zastosowanie w problemach optymalizacyjnych, takich jak wyznaczanie tras w systemach nawigacyjnych.

Literatura

- [1] Aho A., Hopcroft J., Ullman J., *Algorytmy i struktury danych*, Helion, Gliwice, 2003. s. 211-258.
- [2] Drozdek A., *C++. Algorytmy i struktury danych*, Helion, Gliwice 2004. s. 337-365.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Materiały do przedmiotu:
Algorytmy i struktury danych

Laboratorium nr 8

**Implementacja wybranych algorytmów
tekstowych**

Autor:

dr hab. Adam Kiersztyn
mgr Krystyna Kiersztyn



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Spis treści

Wprowadzenie, tematyka zajęć	164
Cel ćwiczenia/laboratorium	164
Instrukcja laboratoryjna/ćwiczeniowa	165
Ćwiczenia samodzielne	178
Pytania pomocnicze	179
Podsumowanie	179
Literatura	179



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Projekt współfinansowany ze środków Unii Europejskiej w ramach: FUNDUSZE EUROPEJSKIE DLA ROZWOJU SPOŁECZNEGO 2021-2027 (FERS) "POLLUB z nami nowoczesne technologie" FERS.01.05-IP.08-0319/23-00

Wprowadzenie, tematyka zajęć

Przetwarzanie tekstu stanowi jeden z ważniejszych obszarów informatyki, znajdujący zastosowanie w wielu dziedzinach, takich jak wyszukiwarki internetowe, analiza dokumentów, systemy wykrywania plagiatów czy przetwarzanie języka naturalnego. Wraz ze wzrostem ilości danych tekstowych rośnie zapotrzebowanie na efektywne metody ich analizy oraz wyszukiwania informacji. Jednym z podstawowych problemów w tej dziedzinie jest **problem wyszukiwania wzorca w tekście**, polegający na odnalezieniu wszystkich wystąpień określonego ciągu znaków (wzorca) w dłuższym ciągu znaków (tekście).

Najprostszym rozwiązaniem tego problemu jest porównywanie wzorca z kolejnymi fragmentami tekstu znak po znaku. Metoda ta jest łatwa do implementacji, jednak w przypadku dużych danych tekstowych może okazać się nieefektywna pod względem czasowym. Z tego powodu opracowano bardziej zaawansowane algorytmy, które pozwalają znacząco ograniczyć liczbę wykonywanych porównań oraz przyspieszyć proces wyszukiwania. Algorytmy te wykorzystują różne techniki, takie jak analiza struktury wzorca, funkcje skrótu czy sprytne przesuwanie wzorca względem tekstu.

Do najbardziej znanych algorytmów wyszukiwania wzorca w tekście należą między innymi algorytm **Knutha-Morrisa-Pratta (KMP)**, algorytm **Rabina-Karpa** oraz algorytm **Boyer-Moore**. Każdy z nich opiera się na innym podejściu do rozwiązania problemu i charakteryzuje się różnymi właściwościami wydajnościowymi. W praktyce ich zastosowanie pozwala na znaczne przyspieszenie operacji przeszukiwania tekstu, szczególnie w przypadku dużych zbiorów danych.

Cel ćwiczenia/laboratorium

Efektem realizacji laboratorium jest:

- zrozumienie zasad działania algorytmów tekstowych
- poznanie metod wyszukiwania wzorca w tekście
- umiejętność implementacji wybranych algorytmów
- analiza i porównanie ich złożoności obliczeniowej



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Instrukcja laboratoryjna/ćwiczeniowa

Przykład 8.1. Algorytm naiwny

Algorytm naiwny (ang. *naive string matching*) jest najprostszą metodą rozwiązywania problemu wyszukiwania wzorca w tekście. Jego działanie polega na sprawdzaniu wszystkich możliwych pozycji, w których wzorec może wystąpić w tekście, bez wykorzystywania dodatkowych optymalizacji.

Niech:

- T – tekst o długości n
- P – wzorec o długości m

Algorytm przesuwając wzorec względem tekstu od lewej do prawej i dla każdej pozycji sprawdza, czy wzorec pasuje do fragmentu tekstu.

Dla każdej pozycji $i = 0, 1, \dots, n - m$:

1. Porównywane są kolejne znaki:

- $T[i]$ z $P[0]$
- $T[i + 1]$ z $P[1]$
- ...
- $T[i + m - 1]$ z $P[m - 1]$

2. Jeżeli wszystkie znaki są zgodne – znaleziono wystąpienie wzorca.

3. Jeżeli wystąpi różnica – algorytm przesuwając wzorec o jedną pozycję w prawo i zaczyna porównania od początku wzorca.

Algorytm P.8.1. Naive string matching

```
NaiveStringMatcher(T,P)
{
    n = length(T)
    m = length(P)
    for i from 0 to n - m:
        j = 0
        while j < m and T[i + j] == P[j]:
            j = j + 1

        if j == m:
            print "Wzorec występuje z przesunięciem " i
    }
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Rozważmy tekst **T = „ABABDABABCAB”** oraz wzorzec **P = „ABABC”**.

Krok 1 (i = 0)

A	B	A	B	D	A	B	A	B	C	A	B
A	B	A	B	C							

Krok 2 (i = 1)

A	B	A	B	D	A	B	A	B	C	A	B
	A	B	A	B	C						

Już przy pierwszym znaku mamy brak dopasowania. Przechodzimy do kolejnej iteracji.

Krok 3 (i = 2)

A	B	A	B	D	A	B	A	B	C	A	B
		A	B	A	B	C					

Dla trzeciego znaku mamy brak dopasowania. Przechodzimy do kolejnej iteracji.

Krok 4 (i = 3)

A	B	A	B	D	A	B	A	B	C	A	B
			A	B	A	B	C				

Przy pierwszym znaku mamy brak dopasowania. Przechodzimy do kolejnej iteracji.

Krok 5 (i = 4)

A	B	A	B	D	A	B	A	B	C	A	B
				A	B	A	B	C			

Przy pierwszym znaku mamy brak dopasowania. Przechodzimy do kolejnej iteracji.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Krok 6 (i = 5)

A	B	A	B	D	A	B	A	B	C	A	B
					A	B	A	B	C		

W tej iteracji mamy pełne dopasowanie. Algorytm znalazł wyszukiwany wzorzec P w tekście T na pozycji 5.

Krok 7 (i = 6)

A	B	A	B	D	A	B	A	B	C	A	B
						A	B	A	B	C	

Krok 8 (i = 7)

A	B	A	B	D	A	B	A	B	C	A	B
							A	B	A	B	C

Algorytm kończy działanie. Wzorzec występuje w tekście na pozycji 5.

Złożoność czasowa

- pesymistyczna: $O(n \cdot m)$
- optymistyczna: $O(n)$ (gdy pierwszy znak często się nie zgadza)

Przykład 8.2. Algorytm Knutha-Morrisa-Pratta (KMP)

Algorytm Knutha-Morrisa-Pratta (KMP) jest efektywną metodą wyszukiwania wzorca w tekście. Jego główną zaletą jest unikanie zbędnych porównań znaków poprzez wykorzystanie informacji o wcześniej dopasowanych fragmentach wzorca.

W przeciwieństwie do algorytmu naiwnego, który po każdej niezgodności rozpoczyna dopasowanie od początku wzorca, algorytm KMP wykorzystuje specjalną tablicę pomocniczą **LPS (Longest Prefix Suffix)**. Tablica ta określa, jaka część wzorca może nadal pasować po wystąpieniu błędu, dzięki czemu nie ma potrzeby ponownego sprawdzania znaków, które już wcześniej zostały dopasowane.

Działanie algorytmu można podzielić na dwa główne etapy. W pierwszym etapie budowana jest tablica LPS na podstawie wzorca. W drugim etapie właściwego przeszukiwania tekstu, algorytm porównuje znaki tekstu i wzorca. W przypadku zgodności indeksy są zwiększane, natomiast w przypadku niezgodności



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



wykorzystywana jest wartość z tablicy LPS, która pozwala określić nowe przesunięcie wzorca bez cofania się w tekście.

Dzięki takiemu podejściu każdy znak tekstu jest analizowany co najwyżej jeden raz, co sprawia, że złożoność czasowa algorytmu wynosi $O(n + m)$, gdzie n oznacza długość tekstu, a m długość wzorca.

Algorytm P.8.2. LPS

```
ComputeLPS(P)
{
    m = length(P)
    LPS[0] = 0
    length = 0
    i = 1

    while i < m:
        if P[i] == P[length]:
            length = length + 1
            LPS[i] = length
            i = i + 1
        else:
            if length != 0:
                length = LPS[length - 1]
            else:
                LPS[i] = 0
                i = i + 1

    return LPS
}
```

Algorytm P.8.3. Algorytm Knutha-Morrisa-Pratta

```
KMP(T, P)
{
    n = length(T)
    m = length(P)
    LPS = ComputeLPS(P)
    i = 0
    j = 0

    while i < n:
        if T[i] == P[j]:
            i = i + 1
            j = j + 1
        else:
            if j != 0:
                j = LPS[j - 1]
            else:
                i = i + 1
}
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```

        j = j + 1

    if j == m:
        print "Wzorzec występuje z przesunięciem " i - j
        j = LPS[j - 1]

    else if i < n and T[i] != P[j]:
        if j != 0:
            j = LPS[j - 1]
        else:
            i = i + 1
    }

```

Aby lepiej zrozumieć działanie algorytmu KMP, rozważmy konkretny przykład wyszukiwania wzorca w tekście. Celem algorytmu jest odnalezienie wszystkich wystąpień wzorca **P** w tekście **T**.

Niech dany będzie tekst **T** = „AABAACAADAABAABA” oraz wzorzec **P** = „AABAA”, długość wzorca $m = 4$.

Krok 1 – Definiowanie tablicy LPS

W pierwszym etapie wyznaczamy tablicę **LPS** (*Longest Proper Prefix which is also Suffix*) dla wzorca **P**.

i	P[i]	LPS[i]
0	A	0
1	A	1
2	B	0
3	A	1
4	A	2

Interpretacja tablicy LPS:

- dla „A” – brak wspólnego prefiksu i sufiksu,
- dla „AA” – najdłuższy wspólny prefiks i sufiks ma długość 1 („A”),
- dla „AAB” – brak wspólnego prefiksu i sufiksu,
- dla „AABA” – najdłuższy wspólny prefiks i sufiks to „A”,
- dla „AABAA” – najdłuższy wspólny prefiks i sufiks to „AA”.

Krok 2 – Wyszukiwanie wzorca w tekście

Wprowadzamy dwa indeksy:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- i – indeks tekstu T ,
- j – indeks wzorca P .

Algorytm działa następująco:

- przy zgodności znaków zwiększamy i oraz j ,
- przy niezgodności wykorzystujemy tablicę LPS:

$$j = \text{LPS}[j - 1],$$

bez cofania indeksu i .

Pierwsze dopasowanie

Początek: $i = 0, j = 0$.

Porównania:

i	j	$T[i]$	$P[j]$
0	0	A	A
1	1	A	A
2	2	B	B
3	3	A	A
4	4	A	A

Uzyskano pełne dopasowanie dla $j = m = 5$. Pozycja wystąpienia dopasowania:

$$i - m = 5 - 5 = 0.$$

Po znalezieniu dopasowania algorytm nie zaczyna od początku wzorca, lecz korzysta z tablicy LPS:

$$j = \text{LPS}[4] = 2.$$

Oznacza to, że pierwsze dwa znaki wzorca („AA”) mogą być ponownie wykorzystane w dalszym dopasowaniu, bo są jednocześnie sufiksem dopasowania.

Kontynuacja przeszukiwania

Stan: $i = 5, j = 2$

Porównujemy:

- $T[5] = C$
- $P[2] = B$

Mamy brak dopasowania. Zamiast wracać do początku wzorca korzystamy z LPS:

$$j = \text{LPS}[1] = 1$$



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Stan: $i = 5, j = 1$

Porównujemy:

- $T[5] = C$
- $P[1] = A$

Ponownie mamy brak dopasowania, $j = LPS[0] = 0$. Ponieważ $j = 0$, algorytm nie ma już informacji do wykorzystania, dlatego zwiększamy indeks tekstu $i = 6$.

Kolejne dopasowanie

Stan: $i = 6, j = 0$

i	j	T[i]	P[j]
6	0	A	A
7	1	A	A
8	2	D	B

Występuje częściowe dopasowanie („AA”). Zamiast zaczynać od początku, korzystamy z LPS:

$$j = LPS[1] = 1$$

Stan: $i = 8, j = 1$

Porównujemy:

- $T[8] = D$
- $P[1] = A$

Ponownie mamy brak dopasowania, $j = 0$, zwiększamy indeks tekst $i = 9$.

Kolejne dopasowanie

Stan: $i = 9, j = 0$

i	j	T[i]	P[j]
9	0	A	A
10	1	A	A
11	2	B	B
12	3	A	A
13	4	A	A

Kolejne porównania prowadzą do pełnego dopasowania. Pozycja wystąpienia:

$$i - m = 14 - 5 = 9.$$



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Wynik

Wzorzec występuje w tekście na pozycjach: 0 oraz 9.

Złożoność czasowa:

$O(n + m)$, gdzie n oznacza długość tekstu, a m długość wzorca.

Przykład 8.3. Algorytm Rabina-Karpa

Algorytm Rabina-Karpa opiera się na wykorzystaniu funkcji skrótu (ang. *hash function*), która przekształca ciąg znaków w wartość liczbową. Zamiast porównywać bezpośrednio wszystkie znaki wzorca z kolejnymi fragmentami tekstu, algorytm porównuje ich wartości hash.

Działanie algorytmu przebiega w następujących krokach:

1. Obliczany jest hash wzorca P .
2. Obliczany jest hash pierwszego fragmentu tekstu T o długości m .
3. Następnie algorytm przesuwając „okno” długości m po tekście i aktualizuje wartość hash w sposób efektywny (tzw. *rolling hash*).
4. Jeżeli hash fragmentu tekstu jest równy hashowi wzorca, wykonywane jest dokładne porównanie znaków w celu potwierdzenia dopasowania.

W celu zobrazowania działania algorytmu Rabina-Karpa rozważono następujące dane wejściowe: tekst $T = „AABAACAADAABAABA”$ oraz wzorzec $P = „AABAA”$. Celem algorytmu jest odnalezienie wszystkich wystąpień wzorca P w tekście T .

Długość wzorca wynosi $m = 5$. Algorytm analizuje kolejne fragmenty tekstu o długości 5.

Algorytm P.8.4. Algorytm Rabina-Karpa

```
RabinKarpMatcher(T, P)
{
    n = length(T)
    m = length(P)
    d = 256
    q = 101

    p = 0      // hash wzorca
    t = 0      // hash tekstu
    h = 1

    // obliczenie h = d^(m-1) mod q
    for i from 0 to m - 2:
```



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



```

h = (h * d) mod q

// obliczenie hash wzorca i pierwszego okna tekstu
for i from 0 to m - 1:
    p = (d * p + code(P[i])) mod q
    t = (d * t + code(T[i])) mod q

// przesuwanie okna po tekście
for i from 0 to n - m:

    if p == t:
        j = 0
        while j < m and T[i + j] == P[j]:
            j = j + 1

        if j == m:
            print "Wzór występuje z przesunięciem " i

    if i < n - m:
        t = (d * (t - code(T[i]) * h) + code(T[i + m])) mod q

    if t < 0:
        t = t + q
}

```

Pierwsze okno (i = 0)

T: A A B A A

P: A A B A A

Hash fragmentu tekstu jest równy hashowi wzorca.

Następuje dokładne porównanie znaków:

- A = A ✓
- A = A ✓
- B = B ✓
- A = A ✓
- A = A ✓

Wzorzec znaleziono na pozycji 0.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Kolejne przesunięcia

Algorytm przesuwa okno o jedną pozycję i aktualizuje hash:

i = 1

T: A B A A C

Hash $\neq$ hash wzorca

Brak dopasowania (bez porównań znaków).

i = 2

T: B A A C A

Hash $\neq$ hash wzorca

Brak dopasowania

i = 3

T: A A C A A

Hash $\neq$ hash wzorca

Brak dopasowania

i = 4

T: A C A A D

Hash $\neq$ hash wzorca

Brak dopasowania

i = 5

T: C A A D A

Hash $\neq$ hash wzorca

Brak dopasowania

Kluczowy moment (i = 9)

T: A A B A A

P: A A B A A

Hash fragmentu tekstu jest równy hashowi wzorca.

Wykonywane jest dokładne porównanie:

- A = A ✓



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- $A = A \checkmark$
- $B = B \checkmark$
- $A = A \checkmark$
- $A = A \checkmark$

Wzorzec znaleziono na pozycji 9

Wynik końcowy

Wzorzec występuje w tekście na pozycjach 0 oraz 9.

Przykład 8.4. Algorytm Boyera-Moore'a

Algorytm Boyera-Moore'a jest jednym z najbardziej efektywnych algorytmów wyszukiwania wzorca w tekście stosowanych w praktyce. W odróżnieniu od algorytmu naiwnego oraz algorytmu KMP, porównuje on znaki **od końca wzorca**, co pozwala na wykonywanie większych przesunięć wzorca względem tekstu.

Podstawowa idea algorytmu polega na tym, że w przypadku wystąpienia niezgodności nie przesuwamy wzorca o jedną pozycję, lecz wykorzystujemy dodatkowe informacje o strukturze wzorca. Dzięki temu możliwe jest pominięcie fragmentów tekstu, które na pewno nie zawierają dopasowania.

Algorytm Boyera-Moore'a wykorzystuje dwie główne heurystyki:

- regułę złego znaku (ang. *bad character rule*),
- regułę dobrego sufiksu (ang. *good suffix rule*).

W niniejszym laboratorium zastosowano uproszczoną wersję algorytmu opartą na regule złego znaku.

Zasada działania (bad character rule)

Dla każdego znaku wzorca zapamiętywana jest jego ostatnia pozycja wystąpienia.

Podczas porównywania:

- jeżeli znaki są zgodne – kontynuujemy porównanie w lewo,
- jeśli wystąpi niezgodność – wzorzec przesuwany jest tak, aby dopasować ostatnie wystąpienie błędnego znaku we wzorcu.

Jeżeli znak nie występuje we wzorcu, możliwe jest przesunięcie wzorca o całą jego długość.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Algorytm P.8.5. Tablica ostatnich wystąpień

```
BuildLast(P)
{
    m = length(P)
    for każdy znak c:
        last[c] = -1

    for i from 0 to m - 1:
        last[P[i]] = i

    return last
}
```

Algorytm P.8.6. Algorytm Boyera-Moore'a

```
BoyerMooreMatcher(T, P)
{
    n = length(T)
    m = length(P)

    last = BuildLast(P)

    i = 0

    while i <= n - m:
        j = m - 1

        while j >= 0 and P[j] == T[i + j]:
            j = j - 1

        if j < 0:
            print "Wzorzec występuje z przesunięciem " i
            i = i + 1

        else:
            i = i + max(1, j - last[T[i + j]])

    }
```

W celu zobrazowania działania algorytmu Boyera-Moore'a rozważono następujące dane: tekst **T** = „**ABAAABCD**” oraz wzorzec **P** = „**ABC**”. Celem jest znalezienie wszystkich wystąpień wzorca w tekście.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



Krok 1 – Tablica ostatnich wystąpień (bad character)

Dla wzorca „ABC” tworzymy tablicę ostatnich wystąpień znaków:

Znak	Pozycja
A	0
B	1
C	2

Dla znaków nieobecnych we wzorcu przyjmujemy wartość -1.

Krok 2 – Wyszukiwanie wzorca w tekście

Algorytm porównuje znaki **od końca wzorca**.

Początek (i = 0)

T: A B A

P: A B C

Porównania (od końca):

• A ≠ C ✗

Błąd na znaku „A”. Sprawdzamy ostatnie wystąpienie „A” we wzorcu – 0. Obliczamy przesunięcie:

$$shift = \max(1, j - last[A]) = 2 - 0 = 2$$

Przesuwamy wzorec o 2 pozycje.

Kolejne ustawienie (i = 2)

T: A A A

P: A B C

Porównania:

• A ≠ C ✗

Znak „A” – ostatnie wystąpienie = 0

$$shift = 2 - 0 = 2$$

Przesuwamy o 2.

Kolejne ustawienie (i = 4)

T: A B C



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



P: A B C

Porównania:

- C = C ✓
- B = B ✓
- A = A ✓

Mamy pełne dopasowanie na pozycji 4.

Wynik

Wzorzec występuje w tekście na pozycji 4.

Ćwiczenia samodzielne

Zadanie 8.1. Algorytm naiwny

Zaimplementuj algorytm naiwny wyszukiwania wzorca P w tekście T. Program powinien zwracać wszystkie pozycje wystąpień wzorca w tekście.

Zadanie 8.2. Algorytm KMP

Zaimplementuj algorytm Knutha-Morrisa-Pratta, w tym:

- funkcję obliczającą tablicę LPS,
- właściwy algorytm wyszukiwania wzorca.

Zadanie 8.3. Algorytm Rabina-Karpa

Zaimplementuj algorytm Rabina-Karpa wykorzystujący funkcję skrótu oraz mechanizm przesuwania okna (rolling hash).

Zadanie 8.4. Algorytm Boyera-Moore'a

Zaimplementować algorytm Boyera-Moore'a z wykorzystaniem reguły złego znaku (bad character rule).

Zadanie 8.5. Najdłuższy wspólny podciąg (LCS)

Zaimplementuj algorytm znajdowania najdłuższego wspólnego podciągu (LCS).

Zadanie 8.6. Wyrażenia regularne

Napisz program, który sprawdza poprawność danych wejściowych za pomocą wyrażeń regularnych.

Sprawdź:



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską



- hasło – co najmniej 8 znaków, zawiera przynajmniej jedną dużą literę i jedną cyfrę
- kod pocztowy (np. 12-345)

Program powinien wczytać dane od użytkownika i wypisać, czy są poprawne.

Pytania pomocnicze

- 1) Na czym polega problem wyszukiwania wzorca w tekście?
- 2) Dlaczego algorytm naiwny jest nieefektywny dla dużych danych?
- 3) Na czym polega idea algorytmu Knutha-Morrisa-Pratta (KMP)?
- 4) Czym jest tablica LPS i jaką pełni rolę w algorytmie KMP?
- 5) Co to jest kolizja hash i jaki ma wpływ na działanie algorytmu Rabina-Karpa?
- 6) Jak działa algorytm Boyera-Moore'a?
- 7) Czym jest reguła złego znaku (bad character rule)?

Podsumowanie

W ramach laboratorium dokonano analizy problemu wyszukiwania wzorca w tekście oraz zapoznano się z wybranymi algorytmami służącymi do jego rozwiązania. Zaimplementowano i przeanalizowano działanie algorytmu naiwnego, algorytmu Knutha-Morrisa-Pratta (KMP), algorytmu Rabina-Karpa oraz algorytmu Boyera-Moore'a.

Literatura

- [1] Banachowski L., Diks K., Rytter W., *Algorytmy i struktury danych*, PWN, Warszawa 2018. s. 164-192.
- [2] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C., *Wprowadzenie do algorytmów*, PWN, Warszawa, 2023. s. 901-926.



Fundusze Europejskie
dla Rozwoju Społecznego



Rzeczpospolita
Polska

Dofinansowane przez
Unię Europejską

