UNIWERSYTETY WSB MERITO

Programowanie aplikacji Java

Maciej Gowin

Zjazd 2 - dzień 2

Linki

Opis

https://maciejgowin.github.io/wsb-java/

Kod źródłowy przykładów oraz zadań

https://github.com/MaciejGowin/wsb-programowanie-aplikacji-java

Problem obliczeniowy

Problem obliczeniowy to problem, który może zostać rozwiązany za pomocą maszyny obliczeniowej, w tym komputera.

Definiowany jest on poprzez funkcję zamieniającą zbiór danych wejściowych na zbiór danych wyjściowych.

Problemy obliczeniowe są rozwiązywane za pomocą algorytmów.

Algorytm

Ciąg zdefiniowanych czynności potrzebnych do zrealizowania danego zadania lub rozwiązania danego problemu.

Algorytm nie jest zależny od języka oraz jego faktycznej implementacji. Do jego opisu stosuje się schematy blokowe lub też pseudokod, który przestawia przepis realizacji danego problemu.

Programy są implementacją rozwiązań problemów obliczeniowych opisanych przy pomocy algorytmów.

> Dany problem może zostać rozwiązany przy pomocy więcej niż jednego algorytmu.

Algorytm: suma liczb N

Problem: suma kolejnych N liczb naturalnych

Algorytm

- Do SUM przypisz 0
- Do COUNTER przypisz 0
- Jeżeli COUNTER jest mniejszy lub równy od N
    - dodaj COUNTER do SUM
    - zwiększ COUNTER o 1
    - powtórz operację
- W przeciwnym wypadku przerwij
- Wypisz SUM

Algorytm: suma liczb N

```
int SUM = 0;
int COUNTER = 0;

while (true) {
    if (COUNTER <= N) {
        SUM = SUM + COUNTER;
        COUNTER = COUNTER + 1;
    } else {
        break;
    }
}
```

Algorytm: suma liczb N

Podobne rozwiązanie zaimplementowane przy pomocy języka przy użyciu innej techniki.

```
int SUM = 0;

for (int COUNTER = 0; COUNTER <= N; COUNTER++) {
    SUM = SUM + COUNTER;
}
```

Złożoność obliczeniowa

Złożoność obliczeniowa definiuje ilość zasobów (takich jak: pamięć, czas użycia procesora) potrzebnych do realizacji danego problemu obliczeniowego.

Do określania złożoności obliczeniowej używamy oszacowań.

Złożoność obliczeniowa

Notacja O (dużego O)

Ograniczenie asymptotycznego tempa wzrostu czasu wykonywania algorytmu, gdzie wynik funkcji jest większy lub równy.

```
∀n▫n0:f(n)▫c▫g(n)
```

> Ilość zasobów potrzebnych do wykonania algorytmu przy założeniu "najgorszych" danych.

W praktyce najczęściej spotykane oszacowanie złożoności obliczeniowej, którym będziemy się posługiwać.

Złożoność obliczeniowa

Notacja Ω (omega)

Ograniczenie asymptotycznego tempa wzrostu czasu wykonywania algorytmu, gdzie wynik funkcji jest mniejszy lub równy.

```
∀n▫n0:f(n)▫c▫g(n)
```

> Ilość zasobów potrzebnych do wykonania algorytmu przy założeniu "najlepszych" danych.

Złożoność obliczeniowa

Notacja Θ (theta)

Ograniczenie asymptotycznego tempa wzrostu czasu wykonywania algorytmu, gdzie wynik funkcji jest jednocześnie większy lub równy, oraz mniejszy lub równy.

```
∀n▫n0:c1▫g(n)▫f(n)▫c2▫g(n)
```

> ilość zasobów potrzebnych do wykonania algorytmu przy założeniu "typowych" danych

Złożoność obliczeniowa - przykład

Algorytm liniowego przeszukiwania (szukanie elementu w liście)

- W najgorszym przypadku czas wykonania to **O(n)** - gdy element jest na końcu listy lub go nie ma.
- W najlepszym przypadku czas wykonania to **Ω(1)** - gdy element znajduje się na pierwszej pozycji.
- Jeśli czas wykonania jest zawsze liniowy, można powiedzieć, że ma **Θ(n)** - bo to jednocześnie górna i dolna granica.

Rząd złożoności obliczeniowej

W praktyce złożoność obliczeniowa jest oszacowaniem, w którym pomijamy mniej istotne czynniki.

Załóżmy zatem, że dla danych wejściowych `n` do obliczenia wyniku potrzebujemy: `f(n) = n^4 + 2*n^2 + 5` operacji. W takim przypadku dla rzędu złożoności obliczeniowej kluczowe znaczenie będzie miał czynnik `n^4`. Rząd złożoności zapiszemy jako `O(n^4)`.

Rząd złożoności O(1)

Złożoność stała, niezależna od ilości danych wejściowych.

Przykłady:

Suma liczb w ciągu liczb naturalnych do N.

Instrukcja programu.

Rząd złożoności O(n)

Złożoność liniowa, wprost proporcjonalna do ilości danych wejściowych. Jest specjalnym przypadkiem złożoności wielomianowej `O(n^k)` gdzie `k=1`.

Przykłady

Suma liczb w tablicy o długości N.

Rząd złożoności O(log(n))

Złożoność logarytmiczna, zależna od logarytmu wielkości danych wejściowych.

Przykłady

Wyszukanie liczby `x` w posortowanej tablicy elementów o długości N.

Algorytm Euklidesa.

> Logarytm jest niemal zawsze o podstawie 2.

Rząd złożoności O(nlog(n))

Złożoność liniowo-logarytmiczna, zależna od iloczynu wielkości oraz logarytmu wielkości danych wejściowych.

Przykład

Sortowanie elementów tablicy o długości N.

Rząd złożoności O(n^2)

Złożoność kwadratowa, proporcjonalna do kwadratu ilości danych wejściowych. Jest specjalnym przypadkiem złożoności wielomianowej `O(n^k)` gdzie `k=2`.

Przykład

Sortowanie elementów tablicy o długości N.

Rząd złożoności O(n^k)

Złożoność wielomianowa. Uogólniony przypadek dla złożoności liniowej oraz kwadratowej.

Rząd złożoności O(k^n)

Złożoność wykładnicza, jej przykładem mogą być O(2^n), O(3^n) itp.

Przykład

Dla tablicy N unikalnych liczb znajdź wszystkie możliwe niepuste podzbiory.

> Algorytmy wykładnicze nazywane są nieefektywnymi. Są niezwykle czułe na wielkość danych wejściowych.

Rząd złożoności O(n!)

Złożoność typu silnia n.

Przykłady

Naiwne rozwiązanie problemu komiwojażera: dla N miast oraz odległości pomiędzy nimi znajdź najkrótsze przejście ze startem i metą w mieście A, zakładając, że odwiedzone zostaną wszystkie miasta.

Programowanie: zadanie 14

Dla dwóch liczb naturalnych znajdź NWD (Największy Wspólny Dzielnik) przy pomocy algorytmu Euklidesa.

Algorytm Euklidesa zakłada, że:

- `NWD (a, b) = NWD (b, a mod b)` oraz
- `NWD (a, 0) = a`.

Klasy złożoności

Klasa złożoności definiuje grupę problemów, do których rozwiązania potrzebna jest podobna grupa zasobów.

Problemy P: (ang. deterministic polynomial, deterministycznie wielomianowy)

Problem, dla którego rozwiązanie można znaleźć w czasie wielomianowym (n^k).

Problem NP: (ang. nondeterministic polynomial, niedeterministycznie wielomianowy)

Problem, dla którego rozwiązanie można zweryfikować w czasie wielomianowym (n^k).

Problem komiwojażera

Problem komiwojażera (TSP, Traveling Salesman Problem)

- Zadana jest lista miast oraz odlełości pomiędzy parami miast.
- Komiwojażer musi odwiedzić każde miasto dokładnie raz i wrócić do miasta startowego.
- Celem jest zminimalizowanie całkowitej odległości podróży.

Problem komiwojażera

Wersja decyzyjna

Czy istnieje trasa, która pozwala odwiedzić wszystkie miasta i wrócić do punktu początkowego z łączną odległością nieprzekraczającym zadanej wartości K.

Klasa NP

Jeśli znamy rozwiązanie (np. sekwencję miast), możemy w czasie wielomianowym zweryfikować, czy całkowita odległość jest mniejsza lub równa K.

Algorytmy sortowania

Problem sortowania jest jednym z podstawowych problemów algorytmicznych. Polega na uporządkowaniu elementów zbioru danych pod względem danej cechy.

Przykładowo:

- dla tablicy liczb cechą może być wielkość liczby
- dla tablicy nazwisk cechą może być porządek alfabetyczny słów

Algorytmy sortowania

Istnieje szereg opracowanych algorytmów sortowania. Do najpopularniejszych należą:

- Sortowanie bąbelkowe (ang. bubblesort) - O(n^2)
- Sortowanie przez wstawianie (ang. insertion sort) - O(n^2)
- Sortowanie przez scalanie (ang. merge sort) - O(nlog(n))
- Sortowanie kubełkowe (ang. bucket sort) - O(n^2)
- Sortowanie przez wybieranie (ang. selection sort) - O(n^2)
- Sortowanie szybkie (ang. quicksort) - O(nlog(n))

Sortowanie bąbelkowe

Załóżmy, że do posortowania mamy N elementową tablicę liczb całkowitych. Algorytm polega na N-1 przejściach tablicy, gdzie w każdym przejściu K dokonujemy N-K porównań dwóch kolejnych liczb i zamiany ich kolejności, jeżeli zaburzają one porządek.

Tablica: 3 7 5 6 2

Przejście K = 1

```
1: [3 7] 5 6 2 -> 3 7 5 6 2
2: 3 [7 5] 6 2 -> 3 5 7 6 2
3: 3 5 [7 6] 2 -> 3 5 6 7 2
4: 3 5 6 [7 2] -> 3 5 6 2 7
```

Programowanie: zadanie 15

Zaimplementuj algorytm sortowania bąbelkowego dla tablicy liczb. Przetestuj działanie dla przykładowych tablic wejściowych.

Sortowanie przez wybieranie

Załóżmy, że do posortowania mamy N elementową tablicę liczb całkowitych. Algorytm polega na N-1 przejściach tablicy, gdzie w każdym przejściu K wyszukujemy minimalnej wartości z nieposortowanych elementów oraz podmieniamy ją z pierwszym elementem z tablicy nieposortowanej.

Tablica: 3 7 5 6 2

```
Przejście K = 1: [3 7 5 6 2] -> minumum: 2 -> 2 3 7 5 6
```

```
Przejście K = 2: 2 [3 7 5 6] -> minumum: 3 -> 2 3 7 5 6
```

```
Przejście K = 3: 2 3 [7 5 6] -> minimum: 5 -> 2 3 5 7 6
```

```
Przejście K = 4: 2 3 5 [7 6] -> minimum: 6 -> 2 3 5 6 7
```

Programowanie: przykład 23

Porównanie wydajności naiwnej implementacji algorytmu sortowania bąbelkowego z wbudowaną metodą sortującą.

```
import java.util.Arrays;
import java.util.Random;

class StopWatch {
    private long start = 0L;

    public void start() {
        start = System.currentTimeMillis();
    }

    public long stop() {
        return System.currentTimeMillis() - start;
    }
}

class RandomGenerator {
    public static Random RANDOM = new Random();

    public static int[] randomTable(int length) {
        int[] items = new int[length];
        for (int i = 0; i < length; i++) {
            items[i] = RANDOM.nextInt();
        }
        return items;
    }
}
```

Programowanie: przykład 23

```
public class Main {

    public static void main(String[] args) {
        testPerformance(1000);
        testPerformance(10000);
        testPerformance(100000);
    }

    public static void testPerformance(int length) {
        StopWatch stopWatch = new StopWatch();
        int[] items = RandomGenerator.randomTable(length);

        int[] bubleSortItems = Arrays.copyOf(items, items.length);
        stopWatch.start();
        bubleSort(bubleSortItems);
        System.out.printf("items: %d: bubleSort: %d%n", bubleSortItems.length, stopWatch.stop());

        int[] sortItems = Arrays.copyOf(items, items.length);
        stopWatch.start();
        Arrays.sort(sortItems);
        System.out.printf("items: %d: sort: %d%n", sortItems.length, stopWatch.stop());
    }

    public static void bubleSort(int[] items) {
        int n = items.length;
        for(int i=0; i<n ;i++) {
            for (int j = 1; j < n - i; j++) {
                if (items[j - 1] > items[j]) {
                    // Replace items
                    int temp = items[j - 1];
                    items[j - 1] = items[j];
                    items[j] = temp;
                }
            }
        }
    }
}
```

Algorytmy rekurencyjne

Rekurencja (zwana także rekursją) polega na odwołaniu funkcji do samej siebie.

Przykładem zastosowania rekurencji może być obliczanie sumy kolejnych N liczb naturalnych. Rozwiązaniem tego problemu może być:

```
SUM(N) = SUM(N-1) + N;
SUM(0) = 0;
```

Istotny jest tutaj warunek brzegowy. W przypadku osiągnięcia wartości `N = 0` zwracana jest liczba 0.

Programowanie: przykład 24

Sumowanie kolejnych liczb naturalnych przy użyciu rekurencji.

```
public class Main {

    public static void main(String[] args) {
        System.out.printf("Sum of: %s: %d%n", 3, sum(3));
        System.out.printf("Sum of: %s: %d%n", 5, sum(5));
        System.out.printf("Sum of: %s: %d%n", 100, sum(100));
    }

    public static int sum(int n) {
        if (n == 0) {
            return 0;
        }
        return sum(n - 1) + n;
    }
}
```

Algorytmy rekurencyjne

Warto zauważyć, że wiele algorytmów może zostać rozwiązanych przy użyciu rekurencji, jak i w podejściu iteracyjnym.

Przykładami takich problemów są:

- obliczanie sumy liczb naturalnych
- obliczanie silni
- obliczanie NWD algorytmem Euklidesa

Rozwiązania rekurencyjne są proste w opisie i zrozumieniu. Należy jednak uważać na poprawne zdefiniowanie warunków brzegowych. Ich zły dobór może spowodować niekończące się wywoływanie funkcji prowadzące do błędów w działaniu.

Programowanie: zadanie 16

Zaimplementuj funkcję obliczającą wartość n-tego elementu ciąg Fibonacciego, jeżeli:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)
```

Dziel i zwyciężaj

Strategia "dziel i zwyciężaj" opiera się na rekurencyjnym podziale problemu na większą liczbę mniejszych podproblemów podobnego typu. Podział następuje do momentu, aż podproblem staje się bezpośrednio rozwiązywalny. Otrzymane wyniki podproblemów scala się, aby uzyskać rozwiązanie dla całego problemu.

Przykładem użycia strategi "dziel i zwyciężaj" jest algorytm sortowania szybkiego (quicksort).

Struktury danych

Struktury danych to sposoby przechowywania danych w pamięci, na których operują algorytmy.

Dany problem może zostać uproszczony do problemu algorytmicznego operującego na danej strukturze danych.

Używając przykładu **problemu komiwojażera**, w podejściu algorytmicznym, może on zostać opisany poprzez operacje na grafach.

Struktury danych

Do podstawowych struktur danych należą:

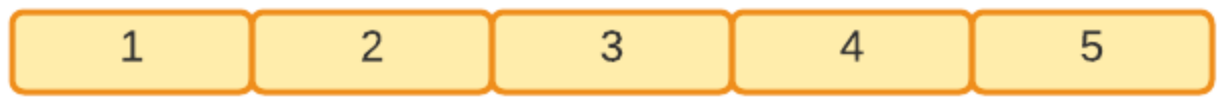
- Tablica
- Lista
- Stos
- Kolejka (LIFO, FIFO)
- Graf
- Drzewa i jego liczne odmiany (np. drzewo binarne)
- Kopiec

Tablica

Struktura uporządkowanych danych, w której każdy z elementów identyfikowany jest przez unikatową liczbę porządkową zwaną indeksem. Indeks jest sekwencyjny i najczęściej przyjmuję wartości liczb naturalnych.

Wyróżniamy dwa główne typy tablic:

- statyczne, ze z góry zdefiniowaną ilościom elementów,
- dynamiczne, ze zmienną ilością elementów.


1
2
3
4
5

Lista

Struktura uporządkowanych danych dla zbiorów dynamicznych, w której elementy są ułożone w porządku liniowym.

Wyróżniamy dwa główne typy list:

- jednokierunkowe, w który każdy element wskazuje na element następny,
- dwukierunkowe, w której każdy element wskazuje na element następny oraz poprzedni.

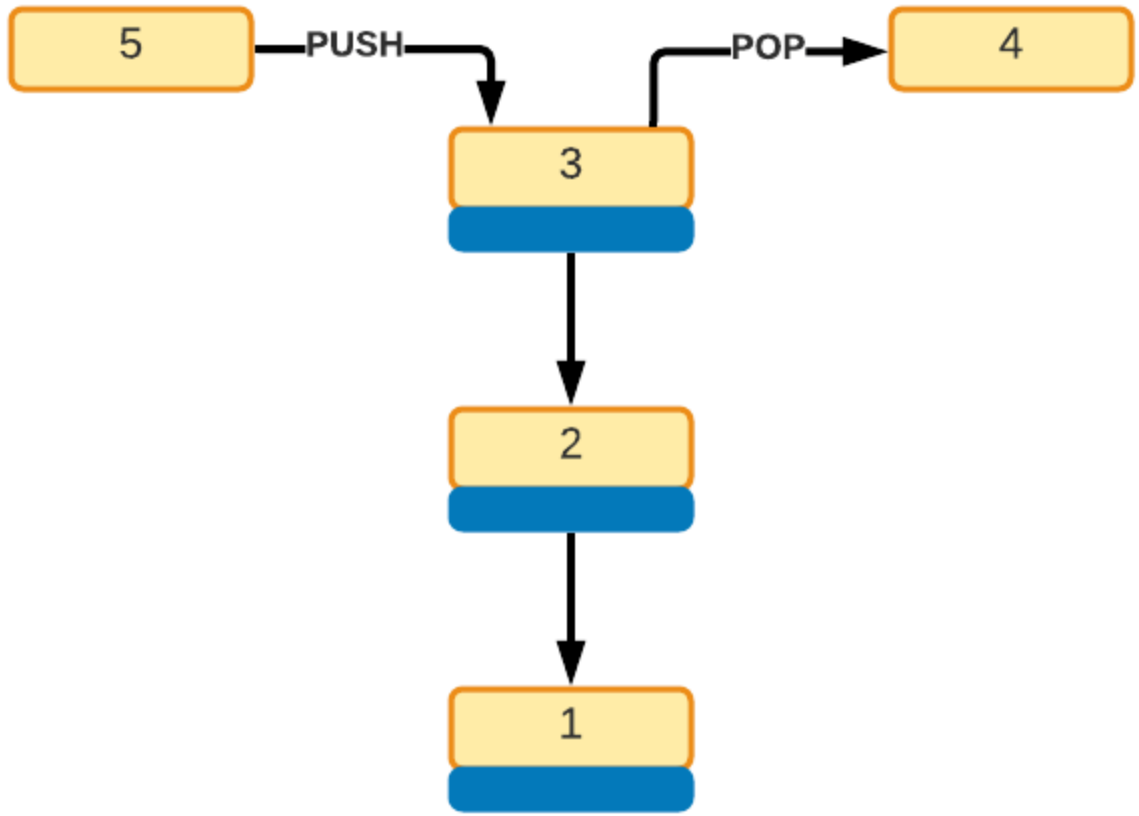

HEAD
1
2
3

Stos

Liniowa struktura uporządkowanych danych realizująca założenie `LIFO (Last In, First Out)`. Elementy dodawane są na jego wierzchołku oraz z tego też wierzchołka są pobierane.

W większości implementacji za dodawanie elementu odpowiedzialna jest operacja `push`, a za pobieranie operacja `pop`. Dodatkowo operacja `peek` ma na celu podgląd ostatniego elementu.

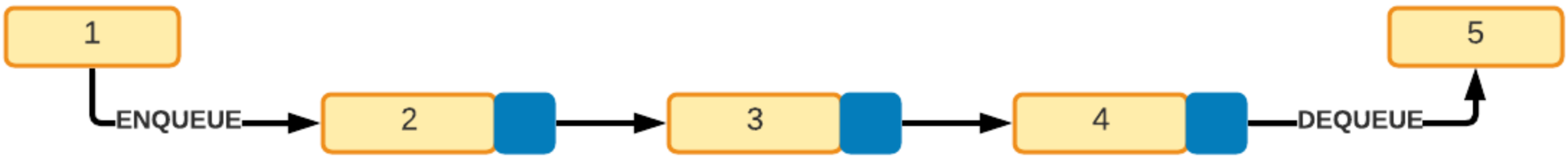
Przykładem stosu mogą być produkty układane na półce sklepowej. Produkt dodany najpóźniej przez obsługę zostanie jako pierwszy ściągnięty przez klienta.


5
PUSH
POP
4
3
2
1

Kolejka

Liniowa struktura uporządkowanych danych realizująca założenie `FIFO (First In, First Out)`. Elementy dodawane są na końcu struktury, ale pobierane z jej początku.

W większości implementacji za dodawanie elementu odpowiedzialna jest operacja `enqueue`, a za pobieranie operacja `dequeue`.


1
ENQUEUE
2
3
4
DEQUEUE
5

Programowanie: zadanie 17

Zaimplementuj klasę `IntegerQueue` przechowującą elementy typu `Integer` oraz realizującą założenia kolejki:

- pobieranie elementu z początku `Integer getFirst()`,
- dodawanie elementu na końcu `void addLast(Integer)`.

Do implementacji kolejki użyj podejścia listy, gdzie każdy element przechowuje referencję do obiektu następnego.

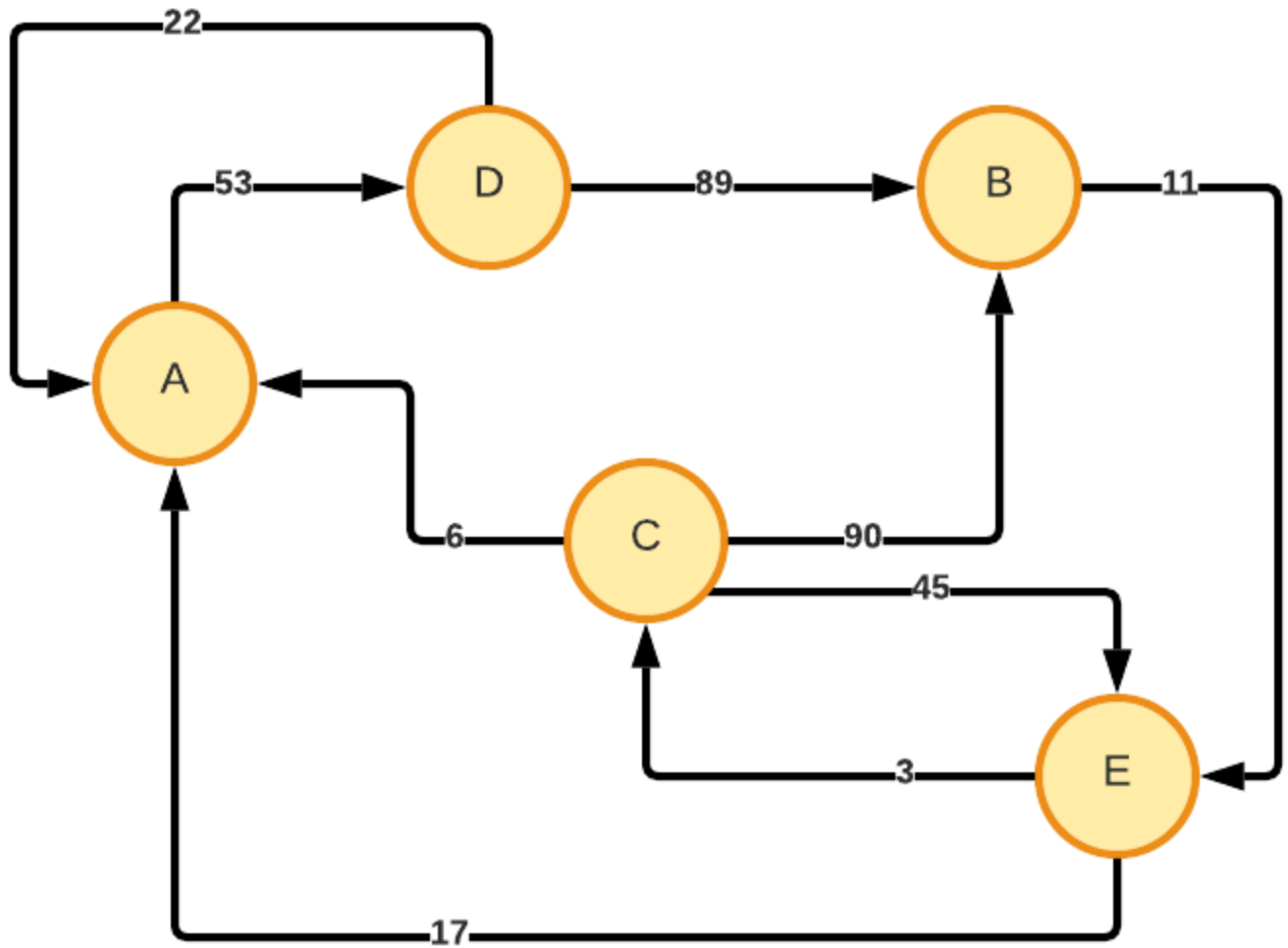
Przetestuj działanie oraz zaimplementuj metodę `String toString()` pozwalającą na wypisanie wszystkich elementów kolejki.

Graf

Struktura matematyczna opierająca się na zbiorze wierzchołków, które mogą być połączone ze sobą krawędziami, gdzie każda z krawędzi ma swój początek i koniec w którymś z wierzchołków.

Istnieje wiele wariancji grafów, w których:

- wierzchołki mogą być numerowane,
- krawędzie mogą byc skierowane,
- krawędzie mogą mieć wagę, gdzie waga może być zależna od kierunku.


22
D
53
89
B
11
A
6
C
90
45
E
3
17

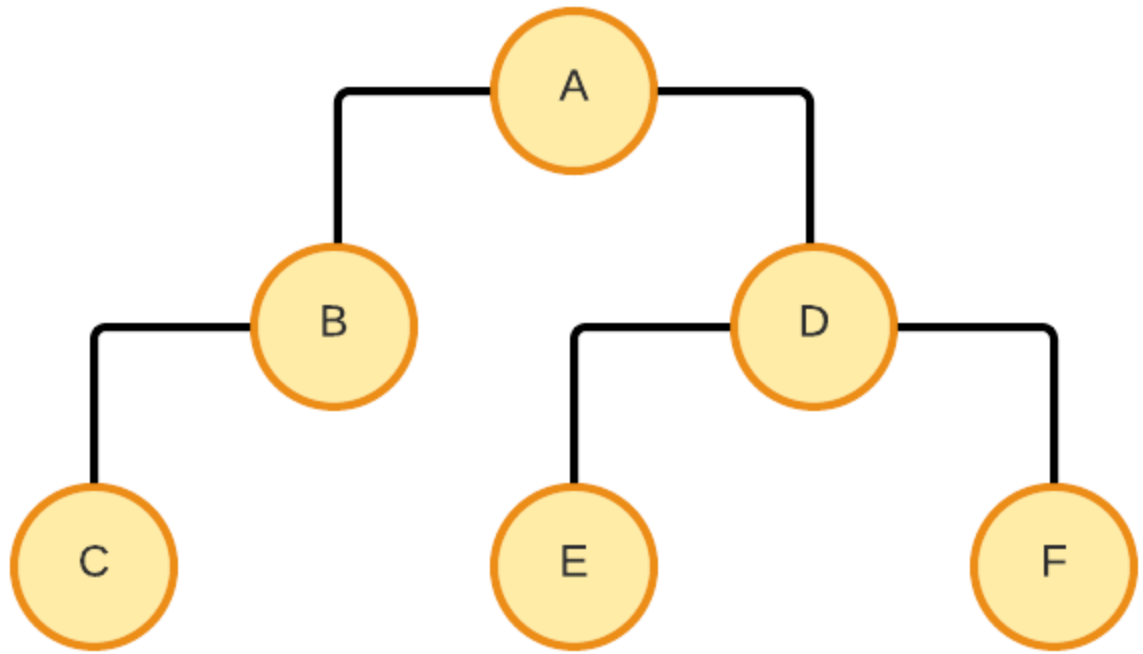
Drzewo

Struktura danych będąca specjalnym przypadkiem grafu, który jest nieskierowany, acykliczny oraz spójny.

Acykliczność to cecha, w której dla każdej pary wierzchołków istnieje dokładnie jedna ścieżka je łącząca.

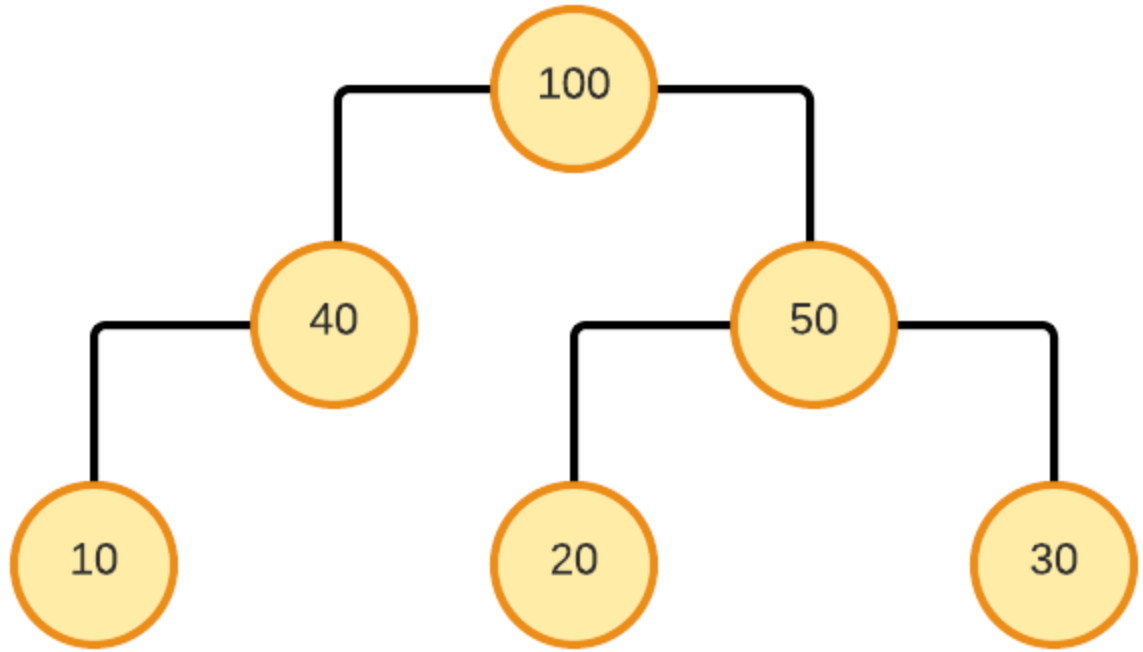
Spójność to cecha, w której dla każdej pary wierzchołków istnieje ścieżka, która je łączy.

Przez swój charakter drzewa dobrze odwzorowują hierarchię danych, przyspieszają wyszukiwanie oraz operacje na posortowanych danych.


A
B
D
C
E
F

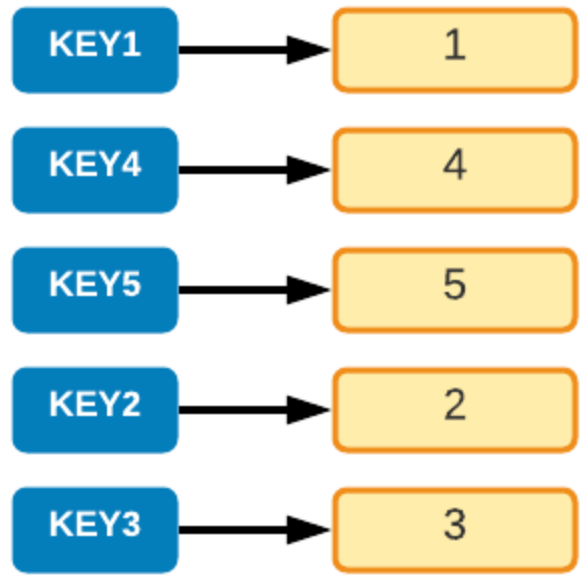
Kopiec

Struktura danych będąca specjalnym przypadkiem drzewa, w którym wartości wierzchołków potomków są w stałej relacji do wartości rodziców.


100
40
50
10
20
30

Mapa

Struktura danych przechowująca pary klucz-wartość, w której dany klucz jest unikatowy i występuje co najwyżej raz.


KEY1
1
KEY4
4
KEY5
5
KEY2
2
KEY3
3

Zbiór

Struktura danych przechowująca unikatowe wartości bez ustalonego porządku.

Do głównych cech należą:

- ponowne dodanie istniejącej wartości do zbioru nie powoduje zwiększania ilości jego elementów
- służy głównie do sprawdzenia, czy dana wartość jest jego częścią


1
4
5
2
3

Programowanie: zadanie 18

Zaimplementuj klasę `IntSet` realizującą założenia zbioru oraz przechowującą elementy typu `int`. Do implementacji użyj struktury kolejki.

Klasa powinna implementować następujące metody:

- `boolean add(int)` dodającą element do zbioru oraz zwracającą `true` w przypadku dodania elementu lub `false` w przeciwnym przypadku
- `boolean remove(int)` usuwającą element ze zbioru oraz zwracającą `true` w przypadku usunięcia elementu lub `false` w przeciwnym przypadku
- `boolean contains(int)` sprawdzającą, czy dany element istnieje w zbiorze
- `int size()` zwracającą ilość elementów w zbiorze
- `String toString()` zwracającą tekstową wersję zbioru z wszystkimi jego elementami