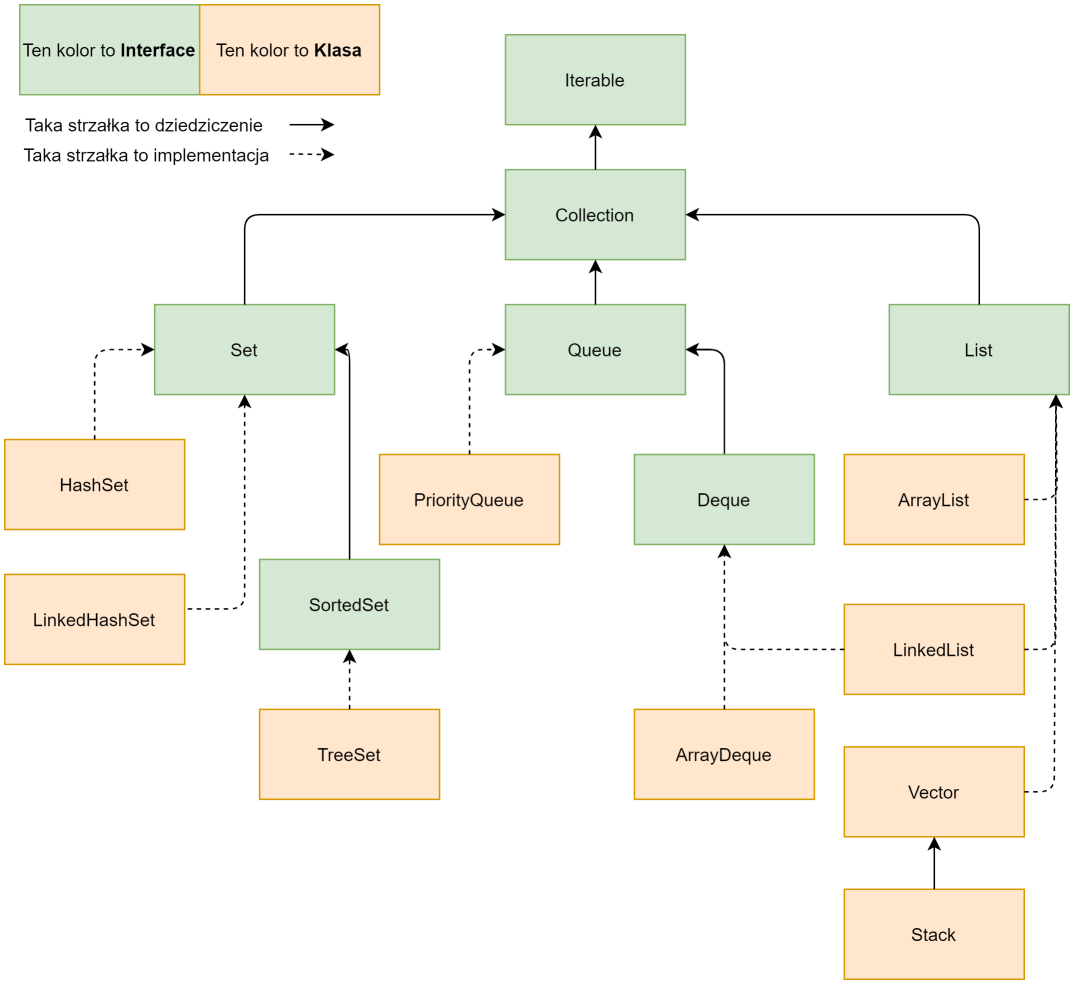
Notatki - Kolekcje

Spis treści

Kolekcje

Kolekcje to określenie na grupę obiektów, które możemy przechowywać w pojedynczym obiekcie (kolekcji). **Java Collections**, to zbiór klas w paczce `java.util`, są tam zdefiniowane m.in. 4 główne interfejsy: `List`, `Set`, `Map` i `Queue`.

W taki sposób wygląda diagram hierarchii klas i interfejsów, które dziedziczą z interfejsu `Collection`, o których będziemy rozmawiać.



Obraz 1. Hierarchia Collections

Tak jak wspomniałem, `List`, `Set`, `Map` i `Queue` to są interfejsy, każdy z nich ma swoje konkretne implementacje, które różnią się pewnymi niuansami. Dlatego przejdźmy po kolei przez 4 omówione w materiałach filmowych rodzaje kolekcji i ich implementacje.

List

Lista to kolekcja elementów zachowująca kolejność. Pozwala na przechowywanie duplikatów. Możliwy jest dostęp do elementów po podanym indeksie, który oznacza miejsce, na którym element się znajduje w kolekcji.

Listy używamy, gdy chcemy przechowywać elementy w określonej kolejności z możliwością posiadania zduplikowanych elementów. Elementy mogą być pobierane i wstawiane w określonych pozycjach na liście w oparciu o indeks - jak w tablicach. Listy są bardzo często używaną kolekcją, gdyż wiele razy przychodzi nam przechowywanie więcej niż jednego elementu w zbiorze. Myślę, że nie będzie to kłamstwo jak napiszę, że jest to jedna z częściej stosowanych kolekcji. Najważniejszą cechą wspólną dla wszystkich implementacji `List` jest to, że mają uporządkowaną kolejność i zezwalają na duplikaty. Poza tym każda implementacja różni się pewnymi cechami.

ArrayList

`ArrayList` jest jak tablica z automatyczną możliwością zmieniania rozmiaru w trakcie działania. `ArrayList` ma de facto pod spodem zaimplementowaną tablicę, którą automatycznie zarządza. Po

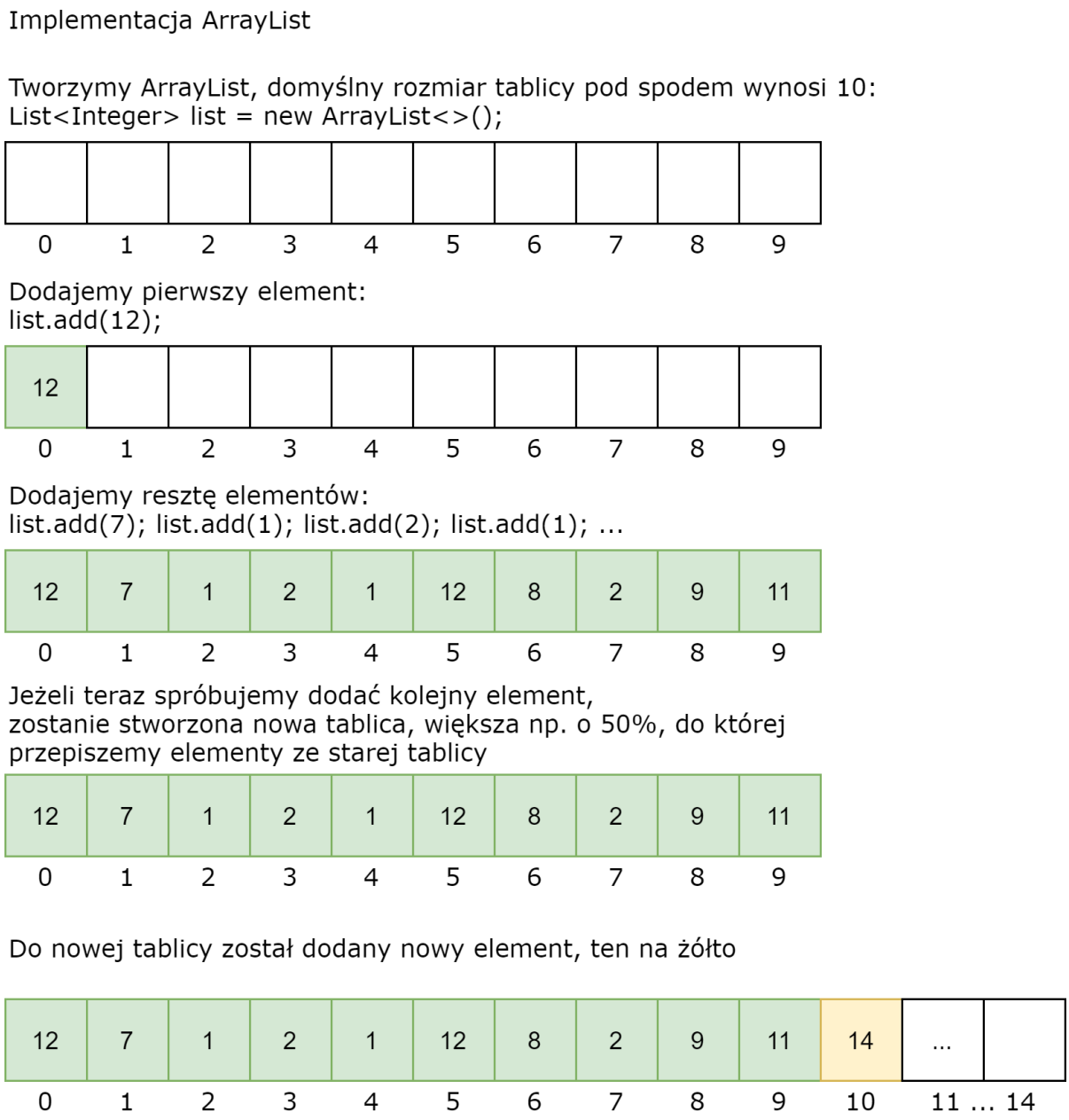
dodaniu elementów do `ArrayList` jest ona w stanie automatycznie zwiększać swój rozmiar.

Główną zaletą `ArrayList` jest to, że możliwe jest wyszukanie dowolnego elementu w stałym czasie, gdyż pod spodem mamy tablicę elementów, która daje nam możliwość dostępu do tych elementów w stałym czasie. Dodawanie lub usuwanie elementu jest natomiast wolniejsze niż uzyskiwanie dostępu do elementu, szczególnie gdy chcemy taki element dodać w środku listy, np. na pozycji 3, wtedy wszystkie elementy za pozycją 3 muszą zostać przesunięte o 1. Tablica pod spodem musi wtedy wymienić indeksy pozostałych elementów, co komplikuje nam taką operację i zwiększa czas.

W najgorszym przypadku, przy dodawaniu elementu do listy, jeżeli skończy nam się miejsce w tablicy pod spodem, `ArrayList` musi stworzyć nową tablicę i przepisać wszystkie elementy ze starej do nowej. Stąd też jeżeli będziemy potrzebowali dodawać bardzo dużo elementów do listy, `ArrayList` może nie być najlepszym wyborem. Jednocześnie `ArrayList` jest dobrym wyborem, gdy więcej czytamy z listy niż do niej zapisujemy (albo mamy tyle samo zapisów co odczytów). Taka operacja została rozrysowana na obrazku poniżej.

Różnica między `ArrayList` a tablicą jest taka, że tablica ma raz określony rozmiar i jak chcemy do niej (tablicy) dodać więcej elementów, niż tablica jest w stanie zmieścić, to musimy zdefiniować nową tablicę z nowym rozmiarem i przepisać do niej elementy ze starej tablicy. `ArrayList` robi to za nas, dlatego też w praktyce tablic używa się bardzo rzadko, najczęściej stosowane są kolekcje.

Często się mówi, że jeśli nie masz pewności, której kolekcji użyć, używaj `ArrayList`.



Obraz 2. ArrayList działanie

Na obrazku zostało wspomniane, że nowo-utworzona tablica może wzrosnąć np. o 50%. Dlatego napisałem np., bo dokumentacja `ArrayList` nie określa jak bardzo ta tablica rośnie gdy wystąpi taka potrzeba, ale często w internecie można przeczytać, że 50%.

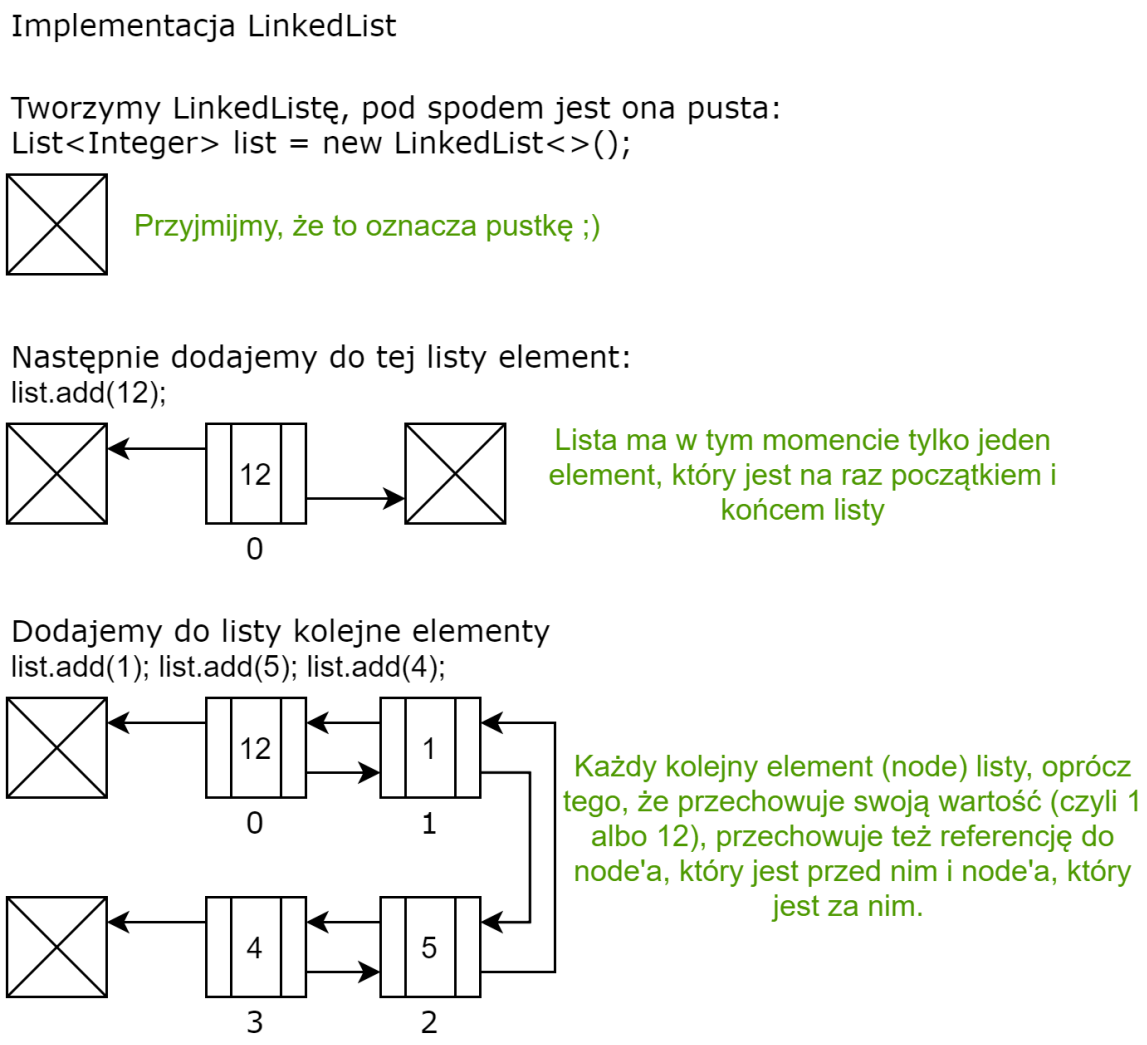
LinkedList

LinkedList jest o tyle specyficzną implementacją, że implementuje jednocześnie interfejsy `List` i `Queue`. Skoro implementuje interface `List`, oznacza to, że ma dostępne wszystkie metody `List`. Posiada również dodatkowe metody ułatwiające dodawanie lub usuwanie z początku i / lub końca listy, gdyż implementuje interface `Queue`.

Główne zalety `LinkedList` to możliwość uzyskiwania dostępu, dodawania i usuwania plików z początku i końca listy w stałym czasie, wynika to z tego w jaki sposób `LinkedList` jest zaimplementowana pod spodem. Jednocześnie kompromis polega na tym, że znalezienie elementu na dowolnym indeksie wymaga czasu liniowego, czyli im dalej jest element od początku lub końca listy tym dłużej go szukamy.

Efekt ten staje się coraz bardziej widoczny im więcej elementów przechowujemy w liście. Spójrz na obrazek poniżej aby zobaczyć jak elementy w liście są ze sobą połączone, wtedy się wyjaśni, czemu aby dostać się do elementu np. na pozycji 3, trzeba zacząć, od któregoś końca i iść przed siebie.

W przypadku `ArrayList` czas odczytu elementu był stały, gdyż tablica pod spodem wiedziała gdzie w pamięci jest każdy z elementów, który był zapisany pod danym indeksem. W przypadku `LinkedList`, szukając elementu na pozycji np. 7, musimy zacząć albo od początku, albo od końca listy i przechodzić przez każdy z elementów na liście, gdyż każdy element wie co następuje przed nim i co nastąpi po nim. W przypadku `LinkedList` nie da się przejść bezpośrednio do elementu na pozycji 7. Musimy zrobić taki sznureczek od początku albo od końca, odpytując każdy z elementów, kto jest następny, aż znajdziemy element, którego szukamy. To zajmuje czas i trwa tym dłużej im więcej elementów na tej liście się znajduje. To wszystko sprawia, że `LinkedList` jest często dobrym wyborem, jeżeli używamy jej jak kolejki, czyli dodajemy i usuwamy elementy na początku lub na końcu. Natomiast nie jest dobrym wyborem, jak mamy dużo elementów w liście i chcemy dostać się do elementu na pozycji np 38.



Obraz 3. LinkedList działanie

To co widać na obrazku powyżej to podwójnie linkowana (*doubly linked*) Lista. Taka implementacja jest stosowana w Javie, czyli każdy element może wskazać namiar na element następny i element poprzedzający. Istnieją również implementacje pojedynczo linkowanych (*singly linked*) list, wtedy możemy tylko zacząć od początku listy i iść przed siebie aż do końca. `LinkedList` jest *doubly linked*.

Przykład wykorzystania `ArrayList` i `LinkedList` różni się tylko konkretną klasą implementującą interface

List, dlatego przykłady pokażę na ArrayList.

```
List<String> listWithCities = new ArrayList<>();
listWithCities.add("Warszawa");
listWithCities.add("Gdańsk");
listWithCities.add("Łódź");
listWithCities.add("Wrocław");

// Tablica wymagała użycia metody Arrays.toString(),
// lista drukuje się normalnie bez tego
// Cities: [Warszawa, Gdańsk, Łódź, Wrocław]
System.out.println("Cities: " + listWithCities);

// IndexOutOfBoundsException
System.out.println("Get 120: " + listWithCities.get(120));
// Get 0: Warszawa
System.out.println("Get 0: " + listWithCities.get(0));
// isEmpty: false
System.out.println("isEmpty: " + listWithCities.isEmpty());
// size: 4
System.out.println("size: " + listWithCities.size());
// contains: false
System.out.println("contains: " + listWithCities.contains("zajavka"));
```

Istnieją również dwie stare implementacje interfejsu List.

Vector

Dawno temu jak chcieliśmy mieć zachowanie analogiczne do List, używało się klasy Vector. W Javie 1.2 zostało wprowadzone ArrayList i od tego momentu należy tej klasy używać zamiast Vector. Vector robi to samo, co ArrayList, tylko wolniej, ze względu na sposób w jaki został zaimplementowany. Implementacja ta powoduje, że Vector jest bezpieczniejszy w przypadku programowania wielowątkowego, ale my cały czas rozmawiamy o aplikacjach jednowątkowych, dlatego nie zagłębiam się w ten temat. Pisząc jednowątkowych mam na myśli, że programy, które piszemy wykonują się w jednym wątku. Nic nie dzieje się równolegle, do tego przejdziemy później. Jedyny powód, dla którego musisz wiedzieć o istnieniu klasy Vector, jest to, że może odnosić się do niej naprawdę stary kod.

Stack

Stack to struktura danych, w której dodajesz i usuwasz elementy ze szczytu stosu. Stos możemy sobie wyobrazić jako stos faktur na stole. Kolejna faktura jest kładziona na górze stosu, pierwszą faktura jaką zdejmujemy jest ostatnia jaka została na tym stosie położona. Podobnie jak klasa Vector, klasa Stack nie jest już dawno używany do tworzenia nowego kodu. W rzeczywistości klasa Stack rozszerza klasę Vector. Jeśli potrzebujesz zachowania stosu, zamiast klasy Stack należy używać klasy ArrayDeque.

Nie należy mylić tej klasy ze strukturą Stosu, którą omawialiśmy w modelu pamięci Javy. Tutaj mówimy o klasie, dzięki której możemy zaimplementować zachowanie stosu.

Set

Kolekcja, która nie pozwala na przechowywanie duplikatów i jednocześnie nie daje nam możliwości posiadania indeksów, czyli oznaczania miejsc w których znajdują się elementy. Nie możemy zatem w

przypadku interfejsu `Set` odwołać się do elementu na konkretnej pozycji, elementy mogą być w zbiorze w losowej kolejności. Wiemy natomiast, że dodając te elementy do zbioru nie będziemy mogli w nim mieć elementów zduplikowanych.

I znowu, każda konkretna implementacja oferuje inną funkcjonalność. Najczęściej używane to `HashSet`, `TreeSet` i `LinkedHashSet`.

Zanim natomiast przejdziemy do implementacji `HashSet` należy wspomnieć o strukturze `HashTable`, bo implementacja `HashSet` go wykorzystuje.

Struktura HashTable

`HashTable` to struktura danych, która będzie nam potrzebna do omówienia klas `HashSet` i `HashMap`.

Przechodzimy do praktycznego wykorzystania metody `hashCode()`. Lubię tłumaczyć tę strukturę na drużynach piłkarskich i tak też zrobimy ☺. Działanie metody `hashCode()` wyjaśnialiśmy w sekcji o programowaniu obiektowym.

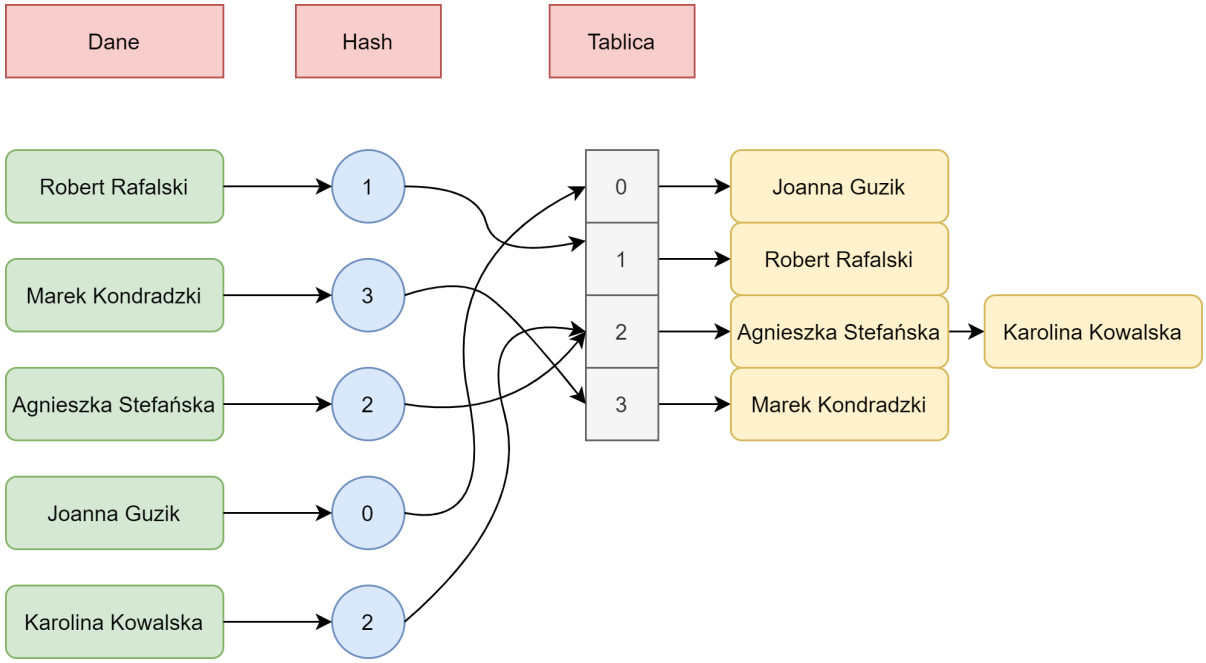
Wyobraź sobie, że masz przed sobą 1000 osób i potrzebujesz wśród nich znaleźć Karolinę Kowalską (patrz obrazek niżej). Ja wiem, że w życiu codziennym można krzyknąć "Hej, szukam Karoliny Kowalskiej" i mogłaby ona podnieść rękę, ale programując to tak nie działa.

Gdybyśmy chcieli w programowaniu znaleźć Panią Karolinę, trzeba by było zapytać każdą z 1000 osób po kolei, czy przypadkiem nie nazywa się "Karolina Kowalska", czyli zrobić `"Karolina Kowalska".equals(osoba.getName())` i albo byśmy trafili od razu, albo znaleźli Panią Karolinę po przepytaniu 998 osób.

Żeby zrobić to szybciej, można by najpierw podzielić wszystkie te 1000 osób na 10 drużyn piłkarskich, na podstawie np. metody `hashCode()`. Czyli liczymy `hashCode()` dla każdej osoby, jak wychodzi 3 to trafia do drużyny 3, jak 9 to do 9. Później napiszę co można zrobić jak `hashCode()` zwróci wynik większy niż mamy ilość drużyn.

Skoro każda z osób ma już policzoną wartość `hashCode()` i na tej podstawie trafiła do jakiejś drużyny, a wiemy też, że szukamy "Karolina Kowalska", to możemy policzyć `hashCode()` z "Karolina Kowalska" i na tej podstawie będziemy wiedzieli, w której drużynie tej Pani szukać. Jeżeli mamy napisaną metodę `hashCode()` w taki sposób, że każda drużyna ma mniej więcej tyle samo osób, to z przeszukiwania 1000 osób, zmniejszyliśmy właśnie ilość porównań `"Karolina Kowalska".equals(osoba.getName()) do 100.

Takie właśnie jest założenie struktur, które wykorzystują `HashTable`, stworzyć tablicę (tablica ta często jest nazywana *hash table*) z jakąś cechą grupującą (w tym przypadku wartość `hashCode()`) i na tej podstawie przyspieszyć wyszukiwanie, bo wiemy w której grupie możemy szukać.



Obraz 4. Hashtable działanie

Na obrazku, dla każdej z osób, możemy policzyć `hashCode()` i na tej podstawie umiejscowić tę osobę w konkretnej grupie. Grupa ma konkretne miejsce w tablicy. Jeżeli w danej grupie pojawia się więcej niż jedna osoba, możemy je dodawać do Listy, którą poznaliśmy wcześniej. Sytuacja taka nazywa się w praktyce *hash collision* - kolizją hashy. Dla kilku osób wyliczyliśmy tę samą wartość `hashCode()` i muszą one trafić do tego samego indeksu w *hash table*, zatem naturalnym jest dodanie pod takim indeksem listy, aby pod tym samym indeksem mogło znaleźć się kilka osób.

I znowu, jeżeli szukalibyśmy takiej osoby "Karolina Kowalska", należałoby policzyć dla kogoś takiego `hashCode()`, znaleźć na tej podstawie odpowiednie miejsce w tablicy *hash table*, zostanie mam wtedy wykonanie `"Karolina Kowalska".equals(osoba.getName())` dla ludzi, którzy znajdują się na liście osób pod danym indeksem tablicy *hash table*, nie będziemy tego musieli robić dla wszystkich ludzi znajdujących się w zbiorze.

Co zrobić jeżeli `hashCode()` wyjdzie większy niż rozmiar tablicy? W praktyce stosuje się modulo. Czyli `hashCode() % rozmiarTablicy`, wtedy otrzymamy wartości w zakresie `0 - rozmiarTablicy` i problem załatwiony.

Dodam tutaj też, że często pojawiają się stwierdzenia o dobrej metodzie `hashCode()`. Patrząc na przykład powyżej, dobra metoda `hashCode()` to taka, która zapewnia bardzo równomierne rozłożenie danych w konkretnych grupach (miejscach w tablicy). Innymi słowy, najgorszy `hashCode()`, to taki, który dla każdego obiektu zwróciłby 1 (a w zasadzie to każdą liczbę, która byłaby stała), bo wtedy nie osiągnęlibyśmy żadnego grupowania, wszyscy byliby w tej samej grupie.

HashSet

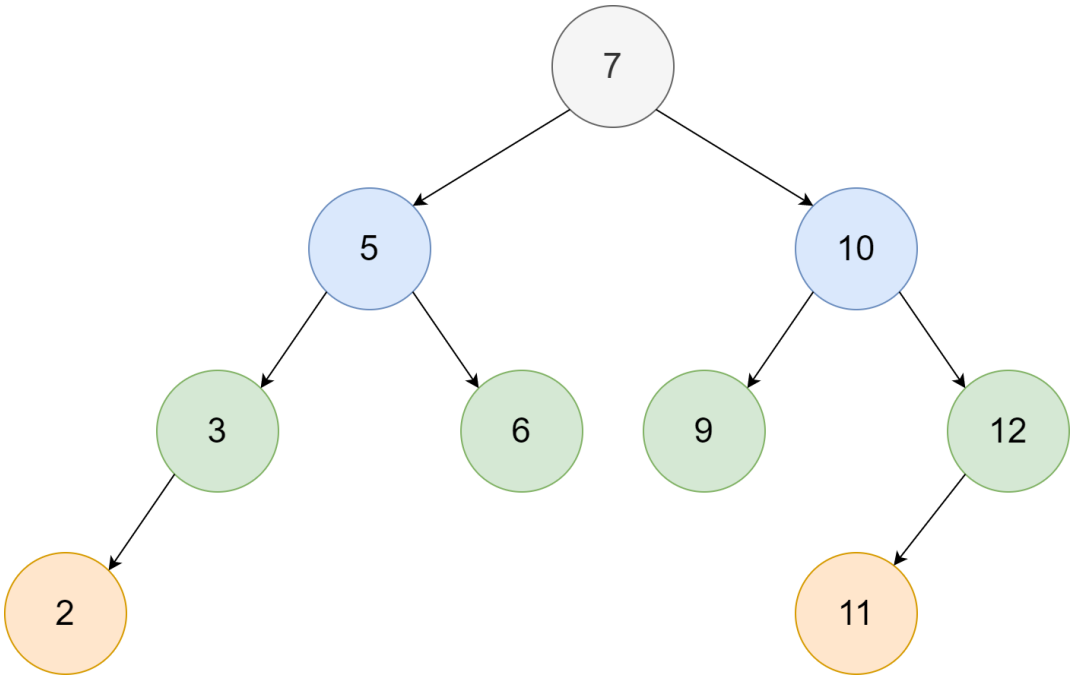
`HashSet` przechowuje swoje elementy zgodnie ze strukturą `HashTable`. Oznacza to, że używa metody `hashCode()` dla przetrzymywanych obiektów, aby wydajniej je pobierać. Należy pamiętać o tym, że przez to, że stosujemy tę strukturę, tracimy informację o kolejności elementów w jakiej były one dodawane. W większości przypadków używania `Set` i tak okazuje się, że nie przyjmujemy określonej kolejności elementów, co sprawia, że `HashSet` jest najpopularniejszą implementacją.

```
Set<String> setWithCities = new HashSet<>();
setWithCities.add("Warszawa");
setWithCities.add("Gdańsk");
setWithCities.add("Łódź");
setWithCities.add("Wrocław");
setWithCities.add("Warszawa");
setWithCities.add("Wrocław");

// Tablica wymagała użycia metody Arrays.toString(),
// set drukuje się normalnie bez tego
// Cities size: [Gdańsk, Warszawa, Łódź, Wrocław]
// Set pozbywa się duplikatów, dlatego mamy tylko unikalne miasta
System.out.println("Cities size: " + setWithCities);

// nie kompiluje się gdyż Set nie ma metody get
// System.out.println("Get 120: " + setWithCities.get(120));

// isEmpty: false
System.out.println("isEmpty: " + setWithCities.isEmpty());
// size: 4
System.out.println("size: " + setWithCities.size());
// contains: false
System.out.println("contains: " + setWithCities.contains("zajavka"));
```

TreeSet

Czym jest drzewo

Zanim przejdziemy do omówienia TreeSet (czyli jakiś zbiór z drzewem), omówmy sobie czym jest drzewo. Jest to struktura, która w naturalny sposób jest w stanie reprezentować hierarchię danych i często jest stosowane do tego celu. Drzewa są stosowane gdy potrzebujemy ułatwić wyszukiwanie danych, ze względu na to, że możemy w prosty sposób przedstawić zależność między danymi jeżeli są one posortowane.



Obraz 5. Przykład drzewa

W opisie poniżej będziemy odnosić się do rysunku powyżej. Drzewa składają się z **wierzchołków** (**węzłów**) oraz **krawędzi** łączących te wierzchołki. Wyróżniamy również określenie takie jak **korzeń**, które oznacza najwyższy wierzchołek (na rys. wartość 7). Jeżeli zaczynamy łączyć ze sobą węzły za pomocą krawędzi, to ciąg takich krawędzi nazywa się **ścieżką**. Drzewo może mieć tylko jedną ścieżkę, która doprowadzi nas z korzenia do każdego wierzchołka. Możemy wyróżnić również **poziom węzła**, który można określić zliczając ilość krawędzi od korzenia do danego węzła. Drzewo może mieć również **wysokość**, która określa największy istniejący poziom. Czyli na rysunku, wartość 7 jest na poziomie 0, wartości 5 i 10 są na poziomie 1, wartości 3, 6, 9 i 12 są na poziomie 2, a wartości 2 i 11 są na poziomie 3. Wysokość drzewa to 3. Wyróżniamy też określenie takie jak **dzieci**, czyli węzły połączone z danym wierzchołkiem. Dziećmi wierzchołka z wartością 5 są wartości 3 i 6. Wierzchołek może mieć dowolną liczbę dzieci, a jeżeli nie ma ich wcale to jest nazywany **liściem**. Czyli liście w przykładzie to 2, 6, 9, 11. Wierzchołek jest też rodzicem, jeżeli ma dzieci. Wtedy jest rodzicem swojego dziecka. Każdy węzeł ma jednego rodzica, oprócz korzenia, który nie ma rodziców.

Rodzajów drzew jest dużo, natomiast na ten moment wyjaśnijmy sobie takie drzewo jak **BST** (*Binary Search Tree*).

Czym jest drzewo binarne (binary tree)

Drzewo binarne to specjalny rodzaj drzewa, w którym liczba dzieci każdego węzła wynosi nie więcej niż 2. Stąd binarne. Czyli mamy maksymalnie prawe dziecko i lewe dziecko. Mówi się wtedy o prawym synu i lewym synu.

Czym jest BST

BST (*Binary Search Tree*) jest specjalnym rodzajem drzewa, w którym każde lewe poddrzewo danego węzła zawiera elementy o wartościach mniejszych niż wartość w danym węźle, a prawe poddrzewo danego węzła zawiera elementy o wartościach nie mniejszych niż wartość w węźle. Wartości, o których wspominam często są kluczami (np. jeżeli będziemy rozmawiali o mapie, wtedy klucze w lewym poddrzewie będą mniejsze niż klucz w węźle, a klucze w prawym poddrzewie będą nie mniejsze). W węzłach oprócz klucza jest też przetrzymywana informacja o rodzicu i o dzieciach.

Po co stosuje się takie struktury? Żeby było szybciej. Wyobraź sobie teraz, że mając drzewo jak w przykładzie na obrazku chcesz znaleźć w takim drzewie wartość 11. Nie musisz przeszukiwać wszystkich wartości w zbiorze, wystarczy, że wiesz, że zbiór jest ułożony w BST. Wtedy zaczynasz od góry i patrzysz, że `11 > 7`, więc idziesz do prawego poddrzewa (bo pamiętasz jaka jest struktura BST). Następnie sprawdzasz, że `11 > 10`, więc znowu idziesz do prawego poddrzewa. Potem sprawdzasz, że `11 < 12`, więc idziesz do lewego poddrzewa i znajdujesz szukaną wartość. Dzięki temu nie musieliśmy porównywać wszystkich elementów w zbiorze, dokonaliśmy o wiele mniej operacji, niż gdybyśmy musieli przejść przez każdy element w zbiorze w pętli i dla każdego wołać metodę `equals()`.

Wracając do wyjaśnienia czym jest TreeSet

TreeSet przechowuje swoje elementy w posortowanej strukturze drzewa. Główną zaletą jest to, że ta implementacja `Set` jest zawsze posortowana. `TreeSet` implementuje specjalny interfejs o nazwie `NavigableSet`, który umożliwia szukanie elementów w secie szukając elementu większego/mniejszego niż inny element. Jeżeli chcemy korzystać z `TreeSet` z obiektami klasy napisanej przez nas, należy pamiętać, żeby albo określić `Comparator`, którego chcemy używać, albo żeby nasza klasa, której obiekty będziemy przechowywać w `TreeSet` implementowała interfejs `Comparable`. Inaczej dostaniemy błąd w trakcie działania programu.

```
class Cat {

    private String name;

    public Cat(final String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Cat: " + name;
    }
}

// Gdzieś w kodzie
// Klasa Cat nie implementuje interfejsu Comparable, więc musimy podać
// Comparator na etapie tworzenia kolekcji,
// inaczej dostaniemy błąd w trakcie działania programu
Set<Cat> catSet = new TreeSet<>((o1, o2) -> o1.name.compareTo(o2.name));
catSet.add(new Cat("cat3"));
catSet.add(new Cat("cat2"));
catSet.add(new Cat("cat4"));
catSet.add(new Cat("cat1"));
catSet.add(new Cat("cat2"));
// [Cat: cat1, Cat: cat2, Cat: cat3, Cat: cat4]
System.out.println(catSet);
```

LinkedHashSet

LinkedHashSet używamy gdy zachodzi potrzeba utrzymania kolejności iteracji. Czyli jednak jest możliwość eliminowania duplikatów i jednocześnie zachowania kolejności dodawania elementów ☺. Iterując po `HashSet`, kolejność elementów jest nieprzewidywalna, podczas gdy `LinkedHashSet` pozwala nam iterować po elementach w kolejności, w jakiej zostały wstawione. `LinkedHashSet` jest zaimplementowany pod spodem poprzez podwójnie linkowaną `LinkedList`, dzięki której możemy zapamiętać kolejność, w jakiej elementy zostały wstawione. Tutaj należy zwrócić uwagę, że możemy iterować po elementach w kolejności ich wstawiania, ale nie mamy indeksów, na których zostały te elementy wstawione, tak jak w przypadku listy. Możemy natomiast stworzyć listę, przekazując do jej konstruktora nasz set, wtedy dostaniemy indeksy. Ponownie należy zwrócić uwagę na to, że z racji, że ta implementacja ma *hash* w nazwie, to w przypadku przetrzymywania w kolekcji naszych własnych klas/obiektów, należy im zaimplementować metody `equals()` oraz `hashCode()`.

```
class Dog {

    private String name;

    public Dog(final String name) {
        this.name = name;
    }

    @Override
    public boolean equals(final Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        final Dog dog = (Dog) o;
        return Objects.equals(name, dog.name);
```

```
    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }

    @Override
    public String toString() {
        return "Dog: " + name;
    }
}

// Gdzieś w kodzie
Set<Dog> dogs = new LinkedHashSet<>();
dogs.add(new Dog("doggo3"));
dogs.add(new Dog("doggo2"));
dogs.add(new Dog("doggo1"));
dogs.add(new Dog("doggo3"));
dogs.add(new Dog("doggo2"));
dogs.add(new Dog("doggo4"));

// [Dog: doggo3, Dog: doggo2, Dog: doggo1, Dog: doggo4]
System.out.println(dogs);
```

Natomiast indeksów nie ma, nawet jak użyjemy referencji typu `LinkedHashSet`:

```
LinkedHashSetSet<Dog> dogs = new LinkedHashSet<>();
dogs.get(0); // błąd kompilacji, bo nie ma takiej metody
```

Queue i Deque

Kolekcja, która porządkuje swoje elementy w określonej kolejności do przetworzenia. Na przykład, gdy chcesz kupić bilet, a ktoś czeka w kolejce, ustawiasz się za tą osobą. Typowa kolejka przetwarza swoje elementy w kolejności first-in-first-out (pierwszy przyszedł, pierwszy wyszedł). Niektóre implementacje kolejki zmieniają to, aby używać innej kolejności. Do tego należy dodać, że jeżeli normalnie elementy są dodawane na końcu kolejki, to możliwe jest też dodanie elementu na początku, coś jak przepuszczenie kobiety w ciąży na początek kolejki.

Poznaliśmy następujące metody implementacji kolejek:

LinkedList

LinkedList, pojawiła się o niej wzmianka już wcześniej, oprócz tego, że jest to lista, jest to też podwójna kolejka. Wcześniej wspominaliśmy, że `LinkedList` w Javie jest zaimplementowana jako doubly linked (podwójnie linkowana) lista. Kolejka z podwójnym zakończeniem różni się od zwykłej kolejki tym, że można wstawiać i usuwać elementy zarówno z przodu, jak i z tyłu kolejki. Główną zaletą `LinkedList` jest to, że implementuje zarówno interfejs `List`, jak i `Queue`. Wadą jest to, że nie jest tak wydajna jak "czysta" kolejka. Innym popularnym formatem jest **LIFO** (ostatni na wejściu, pierwszy na wyjściu).

ArrayDeque

ArrayDeque to "czysta" kolejka dwustronna. Przechowuje swoje elementy w tablicy o zmiennym rozmiarze (znowu jakaś klasa ma `Array` w nazwie i pod spodem jest realizowana przez tablicę). Główną zaletą `ArrayDeque` jest to, że jest bardziej wydajna niż `LinkedList`. Filozofia co do wymowy jest ciekawa, bo jedni mówią **ArrayDeck** a inni **ArrayDekju**. Ja na żywo słyszałem chyba tylko deck, ale mój mózg podpowiada mi d-queue.

`LinkedList` i `ArrayDeque` obydwie implementują interface `Deque`, który natomiast dziedziczy z interfejsu `Queue`. Dlatego o tym mówię, gdyż ostatnia poruszana kolejna `PriorityQueue` implementuje interface `Queue` bezpośrednio.

Do `ArrayDeque` możemy również podchodzić jak do stosu, w zależności od tego, jakich metod będziemy używać. I to właśnie `ArrayDeque` jest rekomendowana do użycia jeżeli potrzebujemy osiągnąć zachowanie Stosu (nie stosujemy starej klasy `Stack`). `ArrayDeque` możemy używać jako:

- **stos** - osiągniemy zachowanie **LIFO** (*last-in-first-out*), jeżeli będziemy używać następujących metod: `push`/`pop`/`peek`,
- **kolejkę** - osiągniemy zachowanie **FIFO** (*first-in-first-out*), jeżeli będziemy używać następujących metod: `offer`/`poll`/`peek`.

Oczywiście jedno i drugie możemy w praktyce ze sobą łączyć i osiągniemy wtedy kolejkę, która od czasu do czasu jest stosem. Znowu, wszystkiego należy używać z głową.

Parę przydatnych metod z interfejsu `Deque`:

metoda	opis	używamy przy kolejce?	używamy przy stosie?
boolean add(E e)	Dodaje element na końcu kolejki i zwraca true albo rzuca wyjątek (np. bo jest za mało miejsca)	Tak	Nie
E element()	Zwraca następny element lub rzuca wyjątek jeżeli kolejka jest pusta	Tak	Nie
boolean offer(E e)	Dodaje element na końcu kolejki i zwraca true jeżeli się udało	Tak	Nie
E remove()	Usuwa i zwraca następny element lub rzuca wyjątek jeżeli kolejka jest pusta	Tak	Nie
void push(E e)	Dodaje element na początku kolejki	Tak	Tak
E poll()	Usuwa i zwraca następny element lub zwraca null jeżeli kolejka jest pusta	Tak	Nie
E peek()	Zwraca następny element lub zwraca null jeżeli kolejka jest pusta	Tak	Tak
E pop()	Zwraca następny element lub rzuca wyjątek jeżeli kolejka jest pusta	Nie	Tak

```
List<Integer> init = Arrays.asList(3, 9, 1, 7);
```

```
Queue<Integer> queue = new ArrayDeque<>();
for (Integer element : init) {
    System.out.println("queue.offer element: " + element + ": " + queue.offer(element));
    System.out.println("queue: " + queue);
}
System.out.println();

// musimy wyjąć rozmiar do zmiennej,
// bo będzie się on zmieniał w trakcie działania programu
// możesz spróbować i zobaczyć co dziwnego się stanie
int size = queue.size();
for (int i = 0; i <= size; i++) {
    System.out.println("queue.peek: " + queue.peek());
    System.out.println("queue: " + queue);
    System.out.println("queue.poll: " + queue.poll());
    System.out.println("queue: " + queue);
}
```

Na ekranie wydrukuje się wtedy coś takiego:

```
queue.offer element: 3: true
queue: [3]
queue.offer element: 9: true
queue: [3, 9]
queue.offer element: 1: true
queue: [3, 9, 1]
queue.offer element: 7: true
queue: [3, 9, 1, 7]

queue.peek: 3
queue: [3, 9, 1, 7]
queue.poll: 3
queue: [9, 1, 7]
queue.peek: 9
queue: [9, 1, 7]
queue.poll: 9
queue: [1, 7]
queue.peek: 1
queue: [1, 7]
queue.poll: 1
queue: [7]
queue.peek: 7
queue: [7]
queue.poll: 7
queue: []
queue.peek: null
queue: []
queue.poll: null
queue: []
```

Patrząc na przykład wyżej możesz zwrócić uwagę, że elementy są dodawane w kolejności przychodzenia, i w tej samej kolejności są z kolejki wyjmowane. Czyli 3, 9, 1, 7. Kolejność zdejmowania jest taka sama, czyli 3, 9, 1, 7.

Ta sama klasa może być używana jak Stos, czyli ostatni położony na stosie element, będzie z niego zdjęty jako pierwszy.

```
List<Integer> init = Arrays.asList(3, 9, 1, 7);

Deque<Integer> stack = new ArrayDeque<>();
for (Integer element : init) {
    stack.push(element);
    System.out.println("stack: " + stack);
}
System.out.println();

int limit = stack.size();
for (int i = 0; i <= limit; i++) {
    System.out.println("stack.peek: " + stack.peek());
    System.out.println("stack: " + stack);
    System.out.println("stack.poll: " + stack.poll());
    System.out.println("stack: " + stack);
}
```

Na ekranie wydrukuje się wtedy coś takiego:

```
stack: [3]
stack: [9, 3]
stack: [1, 9, 3]
stack: [7, 1, 9, 3]

stack.peek: 7
stack: [7, 1, 9, 3]
stack.poll: 7
stack: [1, 9, 3]
stack.peek: 1
stack: [1, 9, 3]
stack.poll: 1
stack: [9, 3]
stack.peek: 9
stack: [9, 3]
stack.poll: 9
stack: [3]
stack.peek: 3
stack: [3]
stack.poll: 3
stack: []
stack.peek: null
stack: []
stack.poll: null
stack: []
```

Patrząc na przykład wyżej można zauważyć, że elementy są dodawane jak faktury kładzione na stole. Czyli na dole jest 3, potem kładziemy 9, potem 1, potem 7. Gdy zdejmujemy te liczby, zdejmujemy je w kolejności 7, 1, 9, 3.

PriorityQueue

PriorityQueue jest używana, gdy obiekty mają być przetwarzane na podstawie priorytetu. Wiadomo, że kolejka działa zgodnie z algorytmem first-in-first-out, ale czasami elementy kolejki muszą zostać przetworzone zgodnie z priorytetem, wtedy, w grę wchodzi PriorityQueue. Przykładowo kobieta w ciąży może mieć pierwszeństwo w kolejce. PriorityQueue jest oparta na zestawie priorytetów. Elementy

kolejki priorytetowej są uporządkowane zgodnie z implementowaną metodą `compareTo()` z interfejsu `Comparable` lub przez `Comparator` dostarczany w czasie budowy kolejki, w zależności od używanego konstruktora.

Elementy, przekazane na kolejkę, będą z niej zdejmowane zgodnie z tym jakbyśmy je posortowali.

```
List<Integer> init = Arrays.asList(3, 9, 1, 7);

Queue<Integer> queue = new PriorityQueue<>();
for (Integer element : init) {
    System.out.println("queue.offer " + element + ": " + queue.offer(element));
    System.out.println("queue: " + queue);
}
System.out.println();

int limit = queue.size();
for (int i = 0; i <= limit; i++) {
    System.out.println("queue.peek: " + queue.peek());
    System.out.println("queue: " + queue);
    System.out.println("queue.poll: " + queue.poll());
    System.out.println("queue: " + queue);
}
```

Na ekranie wydrukuje się wtedy coś takiego:

```
queue.offer 3: true
queue: [3]
queue.offer 9: true
queue: [3, 9]
queue.offer 1: true
queue: [1, 9, 3]
queue.offer 7: true
queue: [1, 7, 3, 9]

queue.peek: 1
queue: [1, 7, 3, 9]
queue.poll: 1
queue: [3, 7, 9]
queue.peek: 3
queue: [3, 7, 9]
queue.poll: 3
queue: [7, 9]
queue.peek: 7
queue: [7, 9]
queue.poll: 7
queue: [9]
queue.peek: 9
queue: [9]
queue.poll: 9
queue: []
queue.peek: null
queue: []
queue.poll: null
queue: []
```

Zwróć uwagę, że elementy są zdejmowane z kolejki zgodnie z tym jakbyśmy je posortowali, czyli 1, 3, 7,

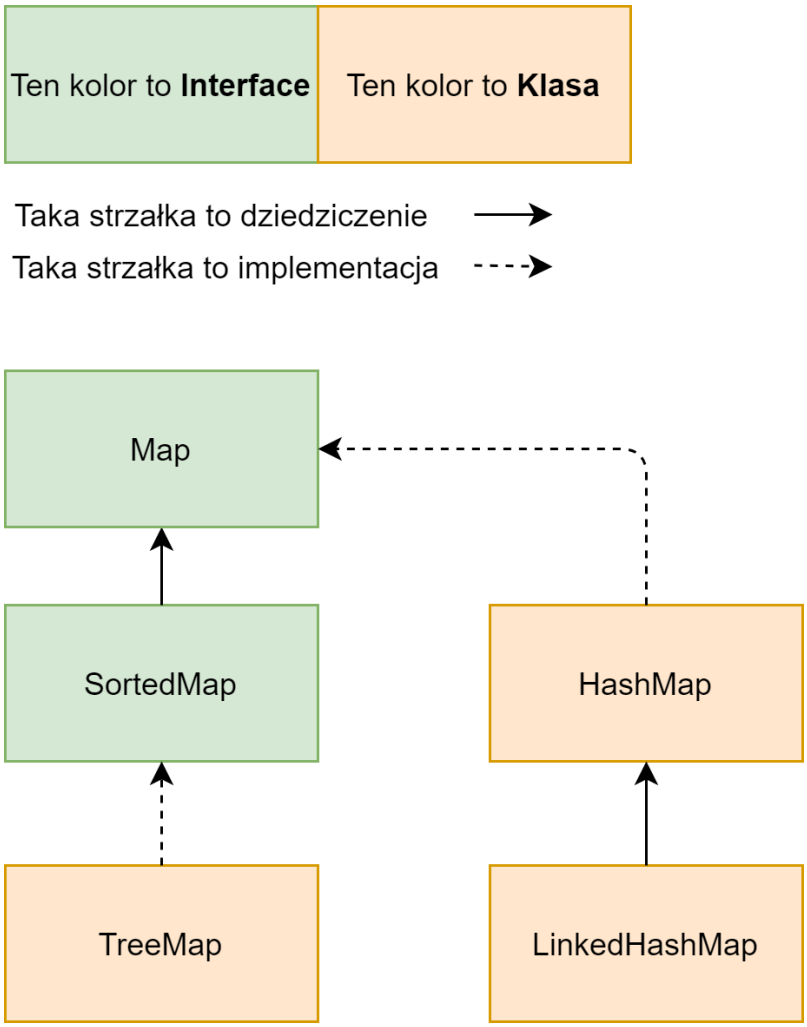
9. Ciekawe natomiast jest to, że na etapie przetwarzania elementów na kolejce, wcale nie są one posortowane tak jak będą z niej zdejmowane. Taka ciekawostka.

Możemy również przekazać do takiej kolejki własny komparator, który np. będzie zdejmował elementy z kolejki w porządku malejącym.

Należy wtedy zdefiniować `Comparator` w ten sposób:

```
Queue<Integer> queue = new PriorityQueue<>((o1, o2) -> o2 - o1);
```

Map



Obraz 6. Hierarchia Map

Map również jest interfejsem, który posiada różne implementacje, które różnią się pewnymi niuansami. Najczęściej używane to `HashMap`, `TreeMap` i `LinkedHashMap`. Wspomnimy też o `Hashtable` (zwróć uwagę na różnicę w wielkiej literze 't' poprzednio dla struktury `HashTable` była ona z wielkiej litery, a dla klasy `Hashtable` jest z małej)

Mapa to zbiór, który odwzorowuje klucze na wartości, bez możliwości przetrzymywania duplikatów kluczy. Elementy mapy to pary **klucz:wartość**.

Mapy należy używać, gdy chcemy identyfikować wartości za pomocą klucza. Na przykład, gdy

korzystamy z listy kontaktów w telefonie, wyszukujemy "Roman", zamiast przeglądać każdy numer telefonu po kolei.

Hashtable

`Hashtable` jest jak klasa `Vector`, ponieważ jest naprawdę stary i napisany z myślą o używaniu wielowątkowym. W nazwie pojawia się litera 't', która zgodnie z nazwą struktury powinna być z wielkiej litery, ale to błąd starych czasów. W formie analogii do starych klas można powiedzieć, że `ArrayList` do `Vector` ma się tak samo, jak `HashMap` do `Hashtable`. `Hashtable` to klasa w Javie. `HashTable` to struktura danych, na której ta klasa bazuje. Taka ciekawostka ☺.

HashMap

HashMap przechowuje klucze zgodnie ze strukturą `HashTable`. Oznacza to, że ponownie, używamy metody `hashCode()` do przechowywania danych i wydajniejszego pobierania wartości. Konsekwencją używania tej struktury jest utrata kolejności wstawiania elementów.

W większości przypadków i tak się tym nie przejmujemy. Jeżeli natomiast nam na tym zależy, możemy wykorzystać `LinkedHashMap`.

Biorąc pod uwagę, że `Map` nie rozszerza `Collection`, w interfejsie `Map` określono więcej metod. Ponieważ w `Map` istnieją zarówno klucze, jak i wartości, definiując Mapę określamy typ generyczny zarówno dla klucza jak i dla wartości.

Przykład użycia:

```
Map<String, String> cars = new HashMap<>();
cars.put("Volvo", "XC60");
cars.put("Fiat", "Panda");
cars.put("Volkswagen", "Golf");

// Get key: XC60
System.out.println("Get key: " + cars.get("Volvo"));
// Remove key: Panda
System.out.println("Remove key: " + cars.remove("Fiat"));
// Size: 2
System.out.println("Size: " + cars.size());
// isEmpty: false
System.out.println("isEmpty: " + cars.isEmpty());
// containsKey: true
System.out.println("containsKey: " + cars.containsKey("Volkswagen"));
// containsValue: true
System.out.println("containsValue: " + cars.containsValue("Golf"));
```

HashMapa posiada taką klasę jak `Entry`, która jest klasą zagnieżdżoną (o których dowiemy się już niebawem). Stąd też żeby dostać się do klasy `Entry`, można napisać to w ten sposób:

```
Set<Map.Entry<String, String>> entries = cars.entrySet();
```

Otrzymujemy dzięki temu `Set` "wystąpień", czyli Entriesów, czyli wszystkich par 'klucz:wartość' dodanych do Mapy.

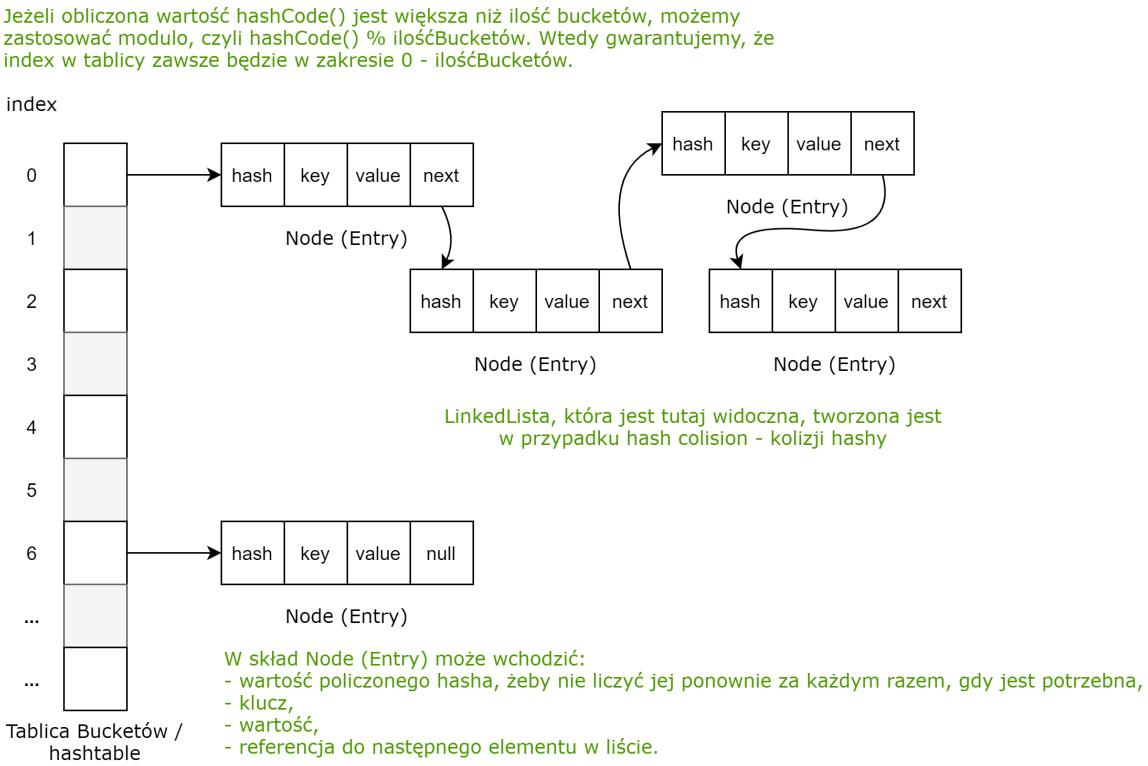
Należy pamiętać o klasie `Entry`, gdyż implementacja `HashMap` wylicza `hashCode()` z klucza, który jest zdefiniowany w `Entry`, a nie z jej całości. Na tej podstawie następnie następuje przetwarzanie tak jak w pokazanej strukturze `HashTable`.

Czyli wyliczamy `hashCode()` z klucza tej pary i `Entry` umiejscawiamy w odpowiednim miejscu w tablicy. Jeżeli wystąpi sytuacja, w której kilka Entriesów ma taką samą wartość `hashCode()`, umiejscawiamy je w tym samym miejscu w tablicy, natomiast przechowujemy je wtedy w liście.

Konkretne miejsce w tablicy *hash table*, która przetrzymuje nam elementy, nazywane jest bucketem (wiaderkiem). W jednym wiaderku możemy mieć listę elementów, które mają taką samą wartość `hashCode()`.

Jeszcze raz wspomnę o przypadku, gdy mamy więcej różnych obliczonych wartości `hashCode()` niż wiaderek - należy wtedy zastosować `hashCode() % liczbaWiaderek`.

W praktyce może to wyglądać w ten sposób:



Obraz 7. HashMap działanie

Kolizja hashy to taka sytuacja, w której, do jednego bucketa trafia kilka nodów (entriesów), bo mają tę samą wartość `hashCode()`. Można ją rozwiązać dodając elementy wpadające do tego samego bucketa do listy. W praktyce może to być np. `LinkedList`, gdyż częściej będziemy do takiej listy zapisywać elementy na jej końcu niż te elementy odczytywać.

Na końcu jeszcze raz napiszmy, że gdy szukamy jakiegoś klucza w mapie, w pierwszej kolejności policzymy dla szukanego klucza wartość `hashCode()`, następnie mając tę wartość znajdziemy wiaderko (bucket), w którym szukany klucz może się znajdować i jeżeli w danym wiaderku będzie kilka elementów na liście, to z każdym kluczem na liście będziemy musieli dokonać porównania poprzez wykorzystanie metody `equals()`. Jeżeli taki klucz w mapie występuje, to przyspieszamy jego znalezienie szukając tylko w konkretnym wiaderku. To jednocześnie pokazuje dlaczego ważny jest kontrakt między

`equals()` a `hashCode()`.

Kolejny niuans jest taki, że jeżeli szukanego klucza w mapie nie ma, to możemy to wykryć na wczesnym etapie, bo np. dane wiaderko może być puste i wtedy szybko skończyliśmy przeszukiwanie elementów.

Java 8

Należy tutaj również dołożyć kolejną cegiełkę. Przykładowy sposób działania `HashMap` pokazuje, że w jednym wiaderku możemy mieć listę elementów. W Java 8 została wprowadzona pewna modyfikacja mająca na celu poprawę wydajności działania. Jeżeli ilość elementów w danym wiaderku przekroczy pewien próg, który jest określany parametrem `TREEIFY_THRESHOLD`, wspomniana lista jest zamieniana na drzewo binarne. Gdy zaczniesz szukać w internecie ile wynosi wartość `TREEIFY_THRESHOLD` znajdziesz informację, że jest to 8. Wyjaśnieniem czemu struktura zostaje zamieniona z listy na drzewo jest poprawa optymalności przeszukiwań elementu. Różne struktury pozwalają znaleźć elementy w innym czasie. W przypadku przekroczenia pewnej ilości elementów, drzewo pozwoli znaleźć szukany element szybciej niż lista. Będzie bardziej zrozumiałe gdy wyjaśnimy sobie czym jest notacja `Big O`, o czym możesz przeczytać na końcu tej notatki.

Są to natomiast szczegóły implementacyjne, najważniejsze jest to, żeby pamiętać logikę działania mapy "pod maską" ☺.

Podsumowując, wiem, że wypisane tutaj zostało dużo wiedzy, ale jest ona ważna. Powodzenia w jej przyswojeniu ☺.

TreeMap

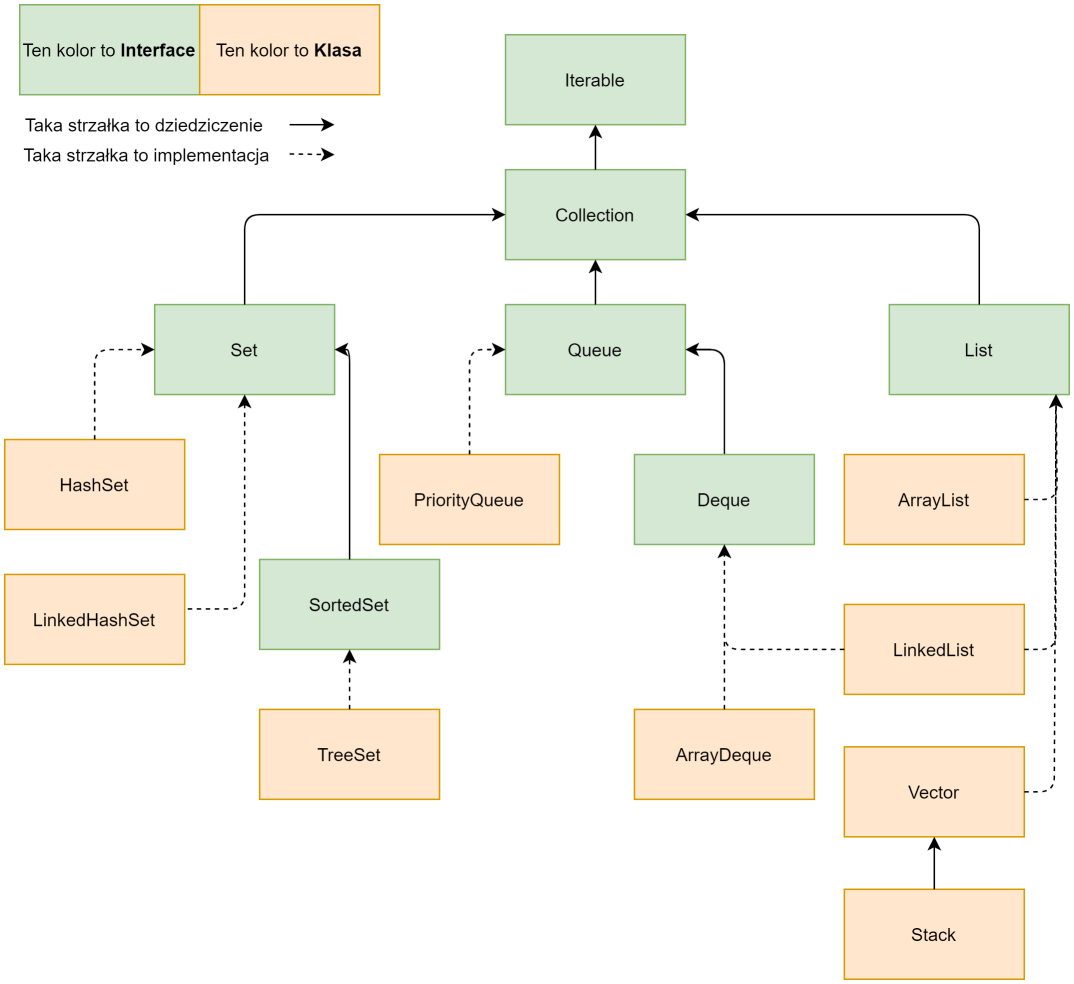
TreeMap przechowuje swoje elementy w posortowanej strukturze drzewa. Główną zaletą jest to, że zestaw jest zawsze posortowany na podstawie kluczy Mapy. Jeżeli chcemy korzystać z `TreeMap` z klasą napisaną przez nas, należy pamiętać, żeby albo określić `Comparator`, którego chcemy używać, albo żeby nasza klasa, która będzie nam służyła za klucz w `TreeMap` implementowała interfejs `Comparable`. Inaczej dostaniemy błąd w trakcie działania programu.

LinkedHashMap

LinkedHashMap używamy gdy zachodzi potrzeba utrzymania kolejności iteracji elementów. Iterując po `HashMap`, kolejność elementów jest nieprzewidywalna, podczas gdy `LinkedHashMap` pozwala nam iterować po elementach w kolejności, w jakiej zostały wstawione.

Podsumowanie

Postarajmy się w tym momencie jeszcze raz ułożyć sobie w głowie wszystkie kolekcje, które zostały dotychczas pokazane. Rzuć zatem okiem jeszcze raz na poniższą grafikę.



Obraz 8. Hierarchia Collections

Notacja Big O

Chcę lekko podrapać po powierzchni temat notacji ***Big O***, czyli np. O(n). W momencie gdy zaczniesz czytać chociażby o różnych algorytmach sortujących, zaczną się tam pojawiać zapisy podobne do O(n), $O(n^2)$ itp. Poniżej chcę ogólnie wyjaśnić, co te zapisy oznaczają.

Na co to komu?

W programowaniu komputerowym używamy notacji ***Big O***, aby mówić o wydajności algorytmów. Gdyby się nad tym zastanowić, to jest całkiem jasne, że wykonanie tej samej czynności przez może zająć 1 sekundę albo 10 sekund. Lepiej jest stosować algorytmy, które dokładnie to samo mogą osiągnąć w krótszym niż dłuższym czasie.

Jeżeli chodzi o wydajność algorytmów, nie można odnosić się do wydajności algorytmów w sekundach, bo każdy komputer wykona ten sam algorytm w innym czasie, ze względu na różnicę parametrów, RAM, CPU, obecną zajętość zasobów. Notacja ta daje 'uniwersalną' metodę porównania poziomu złożoności zadań.

Notacja ***Big O*** pozwala porównać rząd wielkości wydajności, a nie dokładną wydajność. Zakłada również najgorszy czas trwania. Jeśli napiszesz algorytm, który może zająć trochę czasu lub być natychmiastowy, notacja odnosi się do tego dłuższego. Oznaczenie małe `n` jest używane, aby odzwierciedlić liczbę elementów lub rozmiar danych, o których mówisz. Często w przypadku notacji ***Big***

O będziemy odnosić się do czasu wykonania algorytmu, ale nie w rozumieniu sekund tylko zależności czasu trwania od ilości elementów, które trzeba przetworzyć. Poniżej wymieniam najczęstsze wartości notacji duże O, które zobaczysz, i ich znaczenie:

- **O(1)** - stały czas - nie ma znaczenia, jak duża jest ilość elementów do przetworzenia przez algorytm. Przykładowo, zwróć ostatni element listy. Za każdym razem, wykonanie takiej operacji powinno zająć stały czas.
- **O(log n)** - logarytmiczny czas - logarytm to funkcja matematyczna, która rośnie znacznie wolniej niż rozmiar danych. Można sobie to zapamiętać w ten sposób, że jak zobaczymy algorytm z czasem logarytmicznym to powinno być szybciej niż z liniowym. Wyszukiwanie binarne (binary search) działa w czasie logarytmicznym, ponieważ nie obejmuje większości elementów dużych kolekcji.
- **O(n)** - liniowy czas - wydajność będzie rosła liniowo w stosunku do wielkości kolekcji. Zapętlenie listy i zwrócenie liczby elementów pasujących do słowa "Java" zajmie liniowy czas, bo musimy sprawdzić wszystkie elementy w kolekcji.
- **O(n^2)** - kwadratowy czas - kod zawierający pętlę w pętli, w których każda pętla przechodzi przez dane, zajmuje n^2. Przykładem może być wydrukowanie każdego elementu z listy jako para z każdym innym elementem z listy. Czyli jeżeli mamy listę [1, 5, 8], to możemy mieć przykładowo 3 takie pary: [1, 5], [1, 8], [5, 8]. Aby stworzyć taką parę musimy przeiterować listę 2 razy, czyli mieć 2 zagnieżdżone pętle, stąd O(n^2).

Jeżeli teraz zobaczysz tabelę, która przykładowo porównuje ze sobą algorytmy sortujące, będziesz w stanie zrozumieć, w czym te oznaczenia pomagają ☺.

Przypadki

Gdy mówimy o wydajności algorytmów możemy rozróżniać przypadki. W praktyce spotkamy się z takimi stwierdzeniami:

- **worst-case** - najgorszy możliwy czas trwania danego zadania.
- **average-case** - uśredniony czas trwania, biorąc pod uwagę wszystkie możliwe rodzaje danych wejściowych.
- **amortized analysis** - analiza zamortyzowana określa czas niezbędny do wykonania ciągu operacji, bo może się okazać, że wykonanie ciągu operacji jest mniej kosztowne niż wskazuje na to złożoność pojedynczej operacji.