Zestaw 6

Język Java

Termin 12.12.2018

Klasy umieścić w pakiecie **generics**.
Dodać komentarze i użyć narzędzia **javadoc** do wygenerowania dokumentacji.
Szczegóły dotyczące typów generycznych można znaleźć w wykładzie.
Zadanie nr 1 jest obowiązkowe, dodatkowo należy wybrać jedno z zadań nr 2 lub 3.

Zadanie 1. Porównanie wydajności kolekcji

Standardowa biblioteka Javy zawiera szeroki zestaw **kolekcji**, czyli klas, których obiekty służą do przechowywania pewnej liczby innych obiektów. W Javie, kolekcje możemy podzielić ze względu na implementowane interfejsy na następujące typy: **listy** (`java.util.List<E>`), **kolejki** (`java.util.Queue<E>`), **zbiory** (`java.util.Set<E>`) i **mapy** (`java.util.Map<E>`).

Proszę porównać wydajność (czas wykonywania) operacji: `add()`, `remove()`, `contains()`, dla wszystkich klas: `ArrayList`, `HashSet`, `LinkedList`, `Stack`, `Vector`, `PriorityQueue` i `TreeSet` z pakietu `java.util`. Przyjmując, że w każdej z kolekcji będą przechowywane zmienne typu int/Integer (32 bity), proszę oszacować średnią ilość pamięci potrzebną do przechowania jednego takiego elementu w kolekcji. Wynik proszę przesłać w formie tabelki tekstowej:

```
klasa/test      add() [sek.]    remove [sek.]   contains() [sek.]   narzut pamięciowy [bajty]
ArrayList       ??? +/- ???     ??? +/- ???     ??? +/- ???         ??? +/- ???
...             ...             ...             ...                 ...
```

gdzie ??? +/- ??? oznacza wartość średnia z pomiarów +/- odchylenie standardowe

> **Wskazówka:** Do pomiaru czasu wykonania można wykorzystać mechanizmy Instant (od Java 8), System.currentTimeMillis() lub System.nanoTime().
> **Uwaga:** Aby wyniki były wiarygodne czas wykonania zadań powinien wynosić przynajmniej kilka sekund. W tym celu należy w pętli wykonać operacje `add()`, następnie `contains()`, a na końcu `remove()`. Należy dopasować liczbę iteracji.

Uwaga: Należy unikać używania przestarzałej klasy `Vector`, która w zamyśle miała zapewniać bezpieczeństwo wątków, jednak w rzeczywistości wada konstrukcyjna spowodowała, że nie robi tego poprawnie. Ponadto, użycie synchronizacji generuje dodatkowy narzut. W zamian należy użyć nowszej klasy `ArrayList`, która nie zapewnia, zbędnej w przypadku jedno-wątkowym, synchronizacji. Aby stworzyć listę synchronizowaną można użyć metody `Collections.synchronizedList` na obiekcie `ArrayList`.

Zadanie 2. Kolejka priorytetowa (do wyboru)

Proszę zaimplementować kolejkę priorytetową, czyli strukturę, przechowującą pary (obiekt, priorytet), w oparciu o kopiec i implementującą następujące operacje:

- `void add(T t, int priotity)` - wstawia nowy element do kolejki
- `T get()` - zwraca i usuwa z kolejki element o najwyższym priorytecie. Jeśli w kolejce znajduje się kilka obiektów o najwyższym priorytecie, zwracany jest ten z nich, który został tam wstawiony najwcześniej.

Złożoność czasowa operacji `add()` i `get()` powinna być możliwie najmniejsza. W rozwiązaniu nie wolno korzystać z klasy `java.util.PriorityQueue`.

Zadanie 3. Drzewo wyszukiwań binarnych (do wyboru)

Napisać klasę `BinarySearchTree<E extends Comparable<? super E>>` implementującą `Collection<E>` oraz klasę `Node<E extends Comparable<? super E>>`. Klasy te mają obsługiwać dynamiczną strukturę danych - **binarne drzewo poszukiwań** [Cormen, 2013]. W szczególności zdefiniować metody `add` (wstawianie elementu), `toArray` i `iterator`. Pozostałe metody z pustą definicją z interfejsu `Collection` powinny wyrzucać wyjątek `UnsupportedOperationException`. Elementy struktury danych mają być dodawane dynamicznie w taki sposób aby przechodzenie drzewa zwracało je w posortowanej kolejności. Ostatnia metoda ma zwracać obiekt klasy `BinaryIterator<E extends Comparable<? super E>>`, która implementuje interfejs `Iterator<E>`. Zaimplementować metody iteracyjne (na potrzeby `iterator`) i rekurencyjne (na potrzeby `toArray`) do przechodzenia drzewa (*in-order*).

Zadanie:

Napisać interfejs `Pair<K, V>`, który posiada metody `K getKey()` i `V getValue()`. Napisać klasę `OrderedPair<K extends Comparable<? super K>, V>` implementującą interfejsy `Comparable <OrderedPair<K, V>>` i `Pair<K, V>`. Zdefiniować konstruktor oraz metody `compareTo`, `toString`, `getKey` i `getValue`.

Stworzyć obiekt `list` klasy `BinarySearchTree` i kilka obiektów klasy `OrderedPair<String, String>`. Sprawdzić działanie iteratora wypisując posortowane elementy przy użyciu pętli `for(Object e : list)`.

Pytania do zestawu

1. Co to są kolekcje? Jakie kolekcje są zaimplementowane w Standardowej Bibliotece Javy? Czym się różnią?
2. Jakie *struktury danych* implementują poszczególne *kolekcje*? Jakie są ich mocne i słabe strony?
3. Jakie są średnie złożoności obliczeniowe operacji `contains()`, `add()` i `remove()` dla poszczególnych *kolekcji*?
4. Co to są, do czego służą i jakie zalety mają typy generyczne w Javie? Czym się różnią od klas szablonowych w C++?
5. Co oznacza zapis `<E extends Comparable<? super E>>`? Podaj odpowiednie przykłady
6. Co oznacza zapis `for(T e : collection)`? Jakie warunki musi spełniać `collection`?

Zadania dodatkowe

Zadanie A1. `MergeSort`

Zaimplementować algorytm sortowania przez scalanie z wykorzystaniem wątków. Można to zrobić na wiele sposobów, ale zamiast implementować klasę dziedziczącą po `Thread` z metodą `public void run()`, warto użyć mechanizmu Future i Callable (przykład użycia).
Schemat:

```
import java.util.concurrent.*;

public class Compute implements Callable<Integer> {
    public Integer call() throws Exception { ... obliczenia ... }
    public static void someFunction() throws Exception {
        ExecutorService pool = Executors.newFixedThreadPool(...);
        Future<Integer> f    = pool.submit(new Compute(...));
        // Wywołać odpowiednią liczbę wątków
        ...
        Integer r    = f.get();
        // Odebrać wyniki
        ...
        pool.shutdownNow();
    }
}
```

Obiekt klasy implementującej interfejs `Callable<V>` jest wątkiem, który zwraca wynik (np. obliczeń) typu `V`. Alternatywnie można stworzyć obiekt klasy anonimowej (patrz Zestaw 3.) implementującej interfejs `Callable`.

Dodać funkcję statyczną `public static void sort(int[])` i ustawić maksymalną liczbę wątków.

Zmierzyć czas sortowania (np. 10^7 elementów) wersji jednowątkowej i wielowątkowej. Można wykorzystać mechanizmy Instant (od Java 8), System.currentTimeMillis() lub System.nanoTime().

Następnie rozszerzyć klasę o typy generyczne `MergeSortGen<E extends Comparable<? super E>>`.

Andrzej Görlich
agoerlich@netmail.if.uj.edu.pl
http://th.if.uj.edu.pl/~atg/Java

Print: Portrait, A4, Black and white, margins: none