Programowanie współbieżne

Lista zadań nr 10

Na ćwiczenia 11 i 17 grudnia 2025 oraz późniejsze

Zadanie 1. Rozważmy klasę **CoarseList** (implementacja listowa zbioru zabezpieczona globalnym zamkiem).

1. Na przykładzie zbioru zaimplementowanego przy pomocy **CoarseList** wyjaśnij, czym są **niezmiennik reprezentacji** (ang. *representation invariant*) oraz **mapa abstrakcji** (ang. *abstraction map*).
2. Przypomnij, jakie punkty linearyzacji należy wybrać w metodach **add()**, **remove()** i **contains()** by następująca mapa abstrakcji była poprawna: "element należy do zbioru ⇔ węzeł na liście, w którym znajduje się ten element jest osiągalny z węzła **head**".
3. Pokaż, że mapa abstrakcji z poprzedniego punktu nie jest poprawna, jeśli metody będą linearyzowane w momencie zajęcia zamka.
4. Zmodyfikuj powyższą mapę abstrakcji tak, by dla metod linearyzowanych w momencie zajęcia zamka, **CoarseList** nadal była poprawną implementacją zbioru.

Wskazówka: TAoMP 2e, rozdział 9.3 — 9.4.

Zadanie 2. Przypomnij zasadę działania klasy **FineList** (implementacja listowa zbioru zabezpieczona drobnoziarnistymi zamkami, TAoMP2e r. 9.5). Wyjaśnij, dlaczego metody **add()** i **remove()** są linearyzowalne. Dla każdej z metod rozważ osobno przypadek wywołania zakończonego sukcesem i porażką.

Wskazówka: TAoMP 2e, rozdział 9.5.

Zadanie 3. Podaj implementację metody **contains()** dla klasy **FineList**. Uzasadnij jej poprawność.

Zadanie 4. Przypomnij zasadę działania klasy **OptimisticList** (optymistyczna implementacja listowa zbioru, TAoMP2e r. 9.6). Pokaż, że wykonanie metody **remove()** może zostać zagłodzone. Zakładamy, że każdy z zamków występujących w elementach listy

jest niegłodzący, głodzenie w metodzie **remove()** musi zatem wynikać z nieograniczonej liczby obrotów pętli **while(true) {}**. W jaki sposób mogą to wymusić inne wątki operujące współbieżnie na liście?

Zadanie 5. Klasa **OptimisticList** wykorzystuje ponowne przejście przez listę w celu sprawdzenia, czy zablokowane zamkami elementy są nadal osiągalne z początku listy (metoda **validate()**). Zamiast tego warunku można sprawdzać warunek silniejszy, czy lista nie uległa modyfikacji. Fakt zmodyfikowania listy można wyrazić za pomocą zwiększenia znacznika czasu (w praktyce może to być licznik zabezpieczony zamkiem). Wzorując się na klasie **OptimisticList** podaj implementację listową zbioru według tego pomysłu.

Zadanie 6. Przypomnij zasadę działania klasy **LazyList** (implementacja listowa zbioru z leniwym usuwaniem, TAoMP2e r. 9.7). Czy można, bez straty poprawności, zmodyfikować metodę **remove()** klasy **LazyList** tak, by zajmowała tylko jeden zamek?

Zadanie 7. Dla każdej z poniższych modyfikacji listowych implementacji zbioru wyjaśnij, że otrzymany algorytm jest nadal linearyzowalny, lub podaj kontrprzykład wskazujący, że nie jest.

1. W klasie **OptimisticList** metoda **contains()** zajmuje zamki w dwóch węzłach przed stwierdzeniem, czy klucz jest tam obecny. Niech zmodyfikowana metoda **contains()** nie zajmuje żadnych zamków.
2. W klasie **LazyList** metoda **contains()** wykonuje się bez użycia zamków, ale bada wartość bitu marked. Niech zmodyfikowana metoda **contains()** pomija badanie tego bitu.