Algorytmy i struktury danych - kolokwium semestr zimowy 2019/2020

Imię i nazwisko: _____

Numer indeksu: | | | | | | |

C

Zadanie:	1	2	3	4	5	6	7	8	9	10	**Łącznie**
Punktacja:	1	2	1	1	2	2	2	1	2	1	15
Wynik:											

1. (1 punkt) Klasy `std::list` i `std::forward_list` z biblioteki standardowej implementują funkcjonalność odpowiednio listy dwu- i jednokierunkowej. Jaka jest złożoność czasowa wstawiania nowego elementu na pozycji środkowej `std::forward_list`, jeżeli nie dysponujemy żadnym zachowanym iteratorem? Dlaczego w przeciwieństwie do klasy `std::list`, która udostępnia metodę `insert` wstawiającą element przed przekazaną pozycją, `std::forward_list` zapewnia `insert_after` wstawiającą element za przekazaną pozycją?

   Do wstawienia nowego elementu potrzebujemy iteratora do pozycji środkowej - wymaga to przejścia przez połowę listy $O(n/2) \to O(n)$, właściwe wstawianie następuje w czasie stałym.

   W przypadku `std::forward_list` nie jest możliwy dostęp do elementu poprzedzającego (lista jednokierunkowa), dlatego wstawienie nowego elementu może odbywać się tylko za elementem wskazywanym przez przekazany iterator.

2. (2 punkty) Danych jest k posortowanych list jednokierunkowych zawierających n elementów każda. Proszę zaproponować algorytm (wystarczy opis słowny), który scali te listy w jedną posortowaną listę. Algorytm powinien działać w czasie $O(N \log k)$ (gdzie $N = nk$) i korzystać ze stałej ilości pamięci pomocniczej $O(1)$.

   Zasada działania jest analogiczna do scalania na zasadzie „*dziel i zwyciężaj*" algorytmu *merge sort*.

   1. Dobierz listy w pary.
   2. Scal każdą parę iterując jednocześnie przez obydwie listy i dobierając kolejno najmniejsze elementy (scalanie można zrobić w miejscu odpowiednio aktualizując wskaźniki).
   3. Powtórz procedurę dla scalonych list aż do uzyskania pojedynczej listy.

   W każdym kroku wykonujemy operacje o złożoności $O(N)$. Powtarzamy dla $\log k$ poziomów.

3. (1 punkt) Proszę uzupełnić implementację metody, która weryfikuje, czy w liście jednokierunkowej występuje cykl (ostatni element wskazuje na któryś z elementów listy zamiast na **nullptr**). Metoda powinna działać w czasie liniowym $O(n)$ i korzystać ze stałej ilości pamięci pomocniczej $O(1)$.

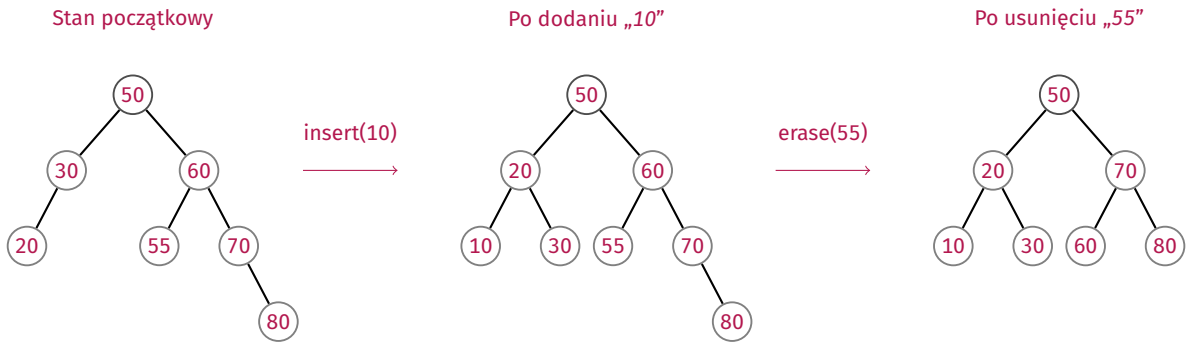
```
bool has_cycle(Node* head) {  // struct Node { int value; Node* next; }
    Node* slow = head;
    Node* fast = head;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;

        if (slow == fast) { return true; }
    }

    return false;
}
```

4. (1 punkt) Proszę opisać różnicę między działaniem kolejki (*queue*) i stosu (*stack*). Którą z tych struktur danych należałoby wykorzystać w iteracyjnym przechodzeniu drzewa binarnego w kolejności poprzecznej (*in-order*)?

Kolejka zwraca elementy w kolejności wstawiania (FIFO), stos zwraca w kolejności od ostatnio wstawionego elementu (LIFO).

Przechodzenie po węzłach drzewa w kolejności *in-order* jest typem przeszukiwania w głąb i można je zrealizować za pomocą stosu.

5. (2 punkty) Dla zadanego drzewa *AVL* proszę naszkicować stany drzewa po sukcesywnie wykonywanych operacjach dodawania elementu o wartości 10 i usuwania elementu o wartości 55.



6. (2 punkty) Proszę wyjaśnić zasadę działania algorytmu sortowania przez wstawianie (*insertion sort*) (opis lub pseudokod). Jaka jest jego złożoność czasowa (w średnim, najlepszym, najgorszym przypadku) i pamięciowa? Czy jest to algorytm stabilny? Proszę zilustrować działanie algorytmu dla przedstawionej tablicy.

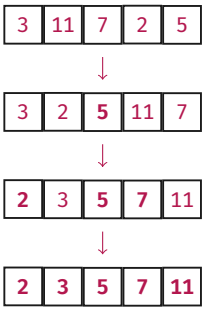
Dla kolejnych elementów z obszaru nieposortowanego wstawia na odpowiednie miejsce w obszarze posortowanym.

- Złożoność czasowa: $O(n)$, $O(n^2)$, $O(n^2)$
- Złożoność pamięciowa: $O(1)$
- Stabilność: tak

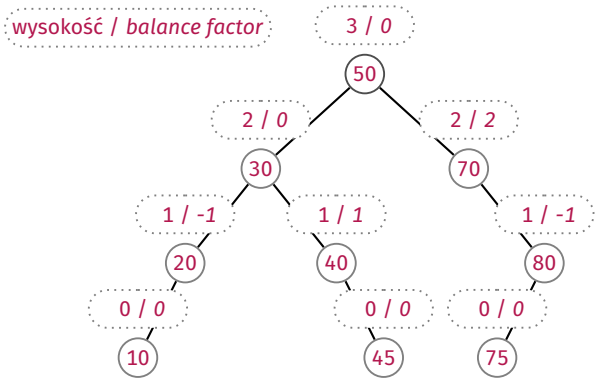
7. (2 punkty) Proszę wyjaśnić zasadę działania algorytmu sortowania szybkiego (*quick sort*) (opis lub pseudokod). Jaka jest jego złożoność czasowa (w średnim, najlepszym, najgorszym przypadku) i pamięciowa? Czy jest to algorytm stabilny? Proszę zilustrować działanie algorytmu dla przedstawionej tablicy.

Sortuje rekursywnie dzieląc na mniejsze fragmenty względem wartości *pivot* (elementy mniejsze przed *pivotem*, elementy większe równe za *pivotem*).

- Złożoność czasowa: $O(n \log n)$, $O(n \log n)$, $O(n^2)$
- Złożoność pamięciowa: $O(\log n)$
- Stabilność: nie

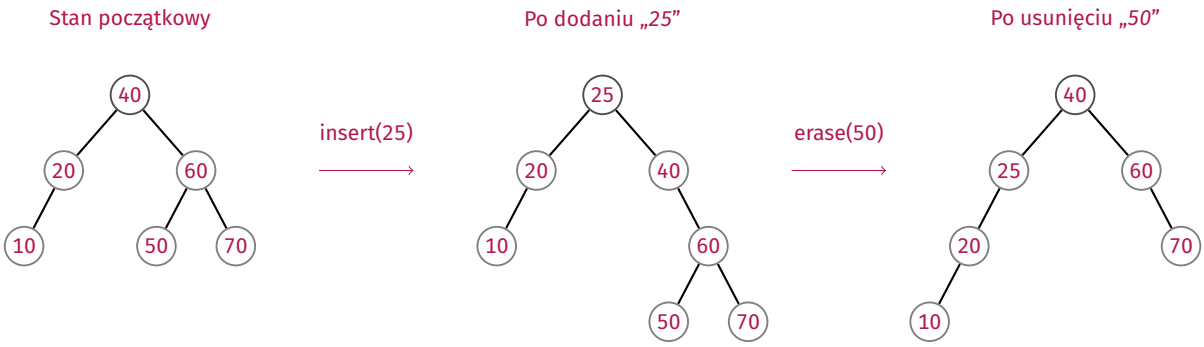


8. (1 punkt) Do każdego z węzłów drzewa proszę dopisać wysokość w danym węźle i współczynnik wyważenia (*balance factor*). Czy przedstawione drzewo jest drzewem AVL?

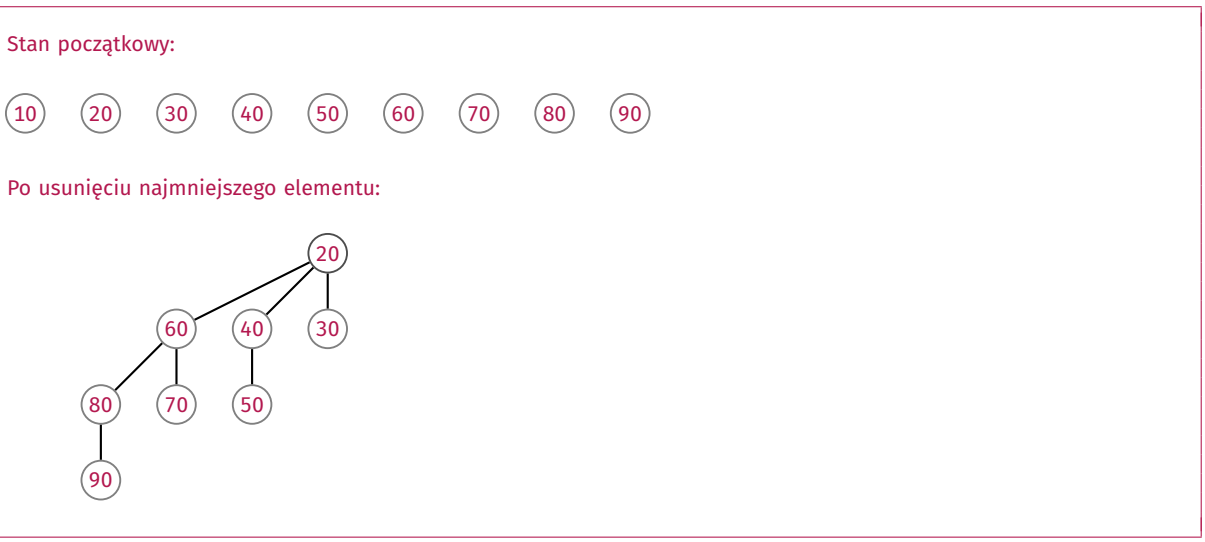


Nie jest AVL.

9. (2 punkty) Dla zadanego drzewa *splay* proszę naszkicować stany drzewa po sukcesywnie wykonywanych operacjach dodawania elementu o wartości 25 i usuwania elementu o wartości 50.



10. (1 punkt) Dany jest kopiec Fibonacciego przedstawiony na rysunku. Proszę naszkicować wynik operacji usuwania najmniejszego elementu przeprowadzony na tym kopcu.



Miejsce na notatki: