Algorytmy i Struktury danych (2022)

Lista zadań 3 (rekurencja, drzewa, sortowanie)

1. Korzystając z twierdzenia o rekurencji uniwersalnej rozwiąż następujące zależności:

   (a) $T(N) = 5T(n/3) + n$,
   (b) $T(N) = 4T(n/2) + n^2$,
   (c) $T(N) = 9T(n/3) + n^2$,
   (d) $T(N) = 6T(n/3) + n^2$,
   (e) $T(N) = 3T(n/3) + n$,
   (f) $T(N) = 5T(n/2) + n^2$,
   (g) $T(N) = T(n/2) + 1$.

2. Ile (dokładnie) porównań wykona algorytm `insertion_sort` w wersji z wartownikiem (liczbą $-\infty$ zapisaną pod adresem `t[-1]`), jeśli dane $(a_1, ..., a_n)$ o rozmiarze `n` zawierają `k` inwersji. Liczba inwersji to liczba takich par (i, j), że $i < j$ i $a_i > a_j$. Jaka jest maksymalna możliwa liczba inwersji dla danych rozmiaru n? Wylicz "średnią" złożoność algorytmu, jaka średnią z maksymalnej i minimalnej ilości porównań jaką wykona. Uwaga: Prawdziwą średnią złożoność oblicza się, jako średnią po wszystkich możliwych permutacjach danych wejściowych.

3. Iteracyjna wersja procedury `mergesort` polega na scalaniu posortowanych list. Zaczyna łączenia pojedynczych elementów w posortowane pary, potem par w czwórki itd. aż do połączenia dwóch ostatnich list w jedną.
   (a) Zakładając, że rozmiar tablicy jest potęgą dwójki $n = 2^k$ oraz, że procedura `merge` wykonuje dokładnie tyle porównań, ile jest elementów po scaleniu, oblicz ile dokładnie porównań wykona cały algorytm.
   (b) Ile razy jest wywołana procedura `merge` w trakcie działania całego algorytmu? Jak zmieni się wynik punktu (a), jeśli założymy, że `merge` zawsze, wykonuje o jedno porównanie mniej, niż zakładaliśmy w punkcie (a)?

4. Napisz procedurę `void insertion_sort(lnode*& L)` – sortowanie przez wstawianie działające na liście jednokierunkowej. Zadbaj o to, by algorytm modyfikował jedynie pola `next` istniejących węzłów (nie używaj `new` ani `delete`). Jeśli list wejściowa jest posortowana, algorytm powinien wykonać tylko $n-1$ porównań. Wskazówka: algorytm wkładać elementy na "nową" listę w kolejności malejącej, a na końcu wywołać `reverse(L)`.

5. Zakładając, że w każdym węźle drzewa BST jest dodatkowe pole `int nL`; pamiętające ilość kluczy w lewym poddrzewie, napisz funkcję `BTSnode* ity(BSTnode *t, int i)`, która zwróci wskaźnik `i`-ty (w kolejności `in order`) węzeł. Przyjmujemy, że numeracja zaczyna się od 0, czyli `ity(t,0)` to węzeł zawierający najmniejszy klucz. Dla wartości $i \geq n$ oraz $i < 0$ wynikiem funkcji powinien być `nullptr`. Pesymistyczna złożoność algorytmu powinna być równa głębokości drzewa. Nie korzystaj z rekurencji.

   Skoryguj procedurę `insert`, i konstruktor `node`, by prawidłowo aktualizowały wartości `nL` w trakcie dodawania elementów.

6. * Skoryguj procedurę `remove`, by prawidłowo aktualizowała wartości `nL` w trakcie usuwania elementów. Napisz funkcję `void remove_ity(BSTnode*&t,int i)`, która usunie `ity(t,i)` węzeł z drzewa.