Algorytmy i Struktury danych (2022)

Lista zadań 2 (tablice, listy, drzewa)

Pomocny przy wykonaniu tej listy jest plik `list.cc` - struktura listy jednokierunkowej, oraz `tree-2020.cc` - drzewa BST - implementacja. Pliki załączone są do maila, ale można też je pobrać z serwisu: `http://panoramx.ift.uni.wroc.pl`. Zapoznaj się z ich zawartością.

1. Ile trzeba porównań, by znaleźć element x w posortowanej tablicy `t` o rozmiarze n. Podaj minimalną wartość gwarantującą sukces i strategię, jak to zrobić. Postaraj się podać wzór ogólny, który pozwoli wyliczyć dokładną wartość dla dowolnego n. Sprawdź go dla $n = 1, ..., 20$.

2. Ile trzeba porównań, by znaleźć element x w nieuporządkowanej tablicy `t` o rozmiarze n. Oblicz wartość średnią i wariancję zakładając, że element x może znajdować się z jednakowym prawdopodobieństwem, pod dowolnym indeksem tablicy.

3. Ile dokładnie potrzeba mnożeń, by obliczyć w standardowy sposób: (a) iloczyn skalarny dwóch wektorów rozmiaru n. (b) Wartość wielomianu stopnia n (schemat Hoernera). (c) Iloczyn dwóch wielomianów stopnia n. (d) Iloczyn dwóch macierzy $n \times n$. (e) Wyznacznik macierzy $n \times n$ (sprowadzenie do postaci trójkątnej przez eliminację Gaussa). Dla każdego przypadku napisz też jakiej klasy $\Theta(n^k)$ są to funkcje. Uwaga: Dzielenie to mnożenie przez odwrotność, więc liczymy je podobnie.

4. Rozważ trzy wersje znajdowania maksimum w tablicy `int maks(int t[],int n)`.
   (a) iteracyjna - `{int x=a[--n];while(n--)if(t[n]<x)x=t[n]; return x;}`
   (b) rekurencyjna - znajdź maksimum `n-1` elementów; porównaj je z ostatnim elementem
   (c) rekurencyjna - podziel tablicę na dwie części; znajdź ich maksima; wybierz większe z nich.
   Ile porównań między elementami wykonuje każda z wersji? Ile pamięci wymaga każda z tych wersji? Uwzględnij fakt, że głębokość rekurencji ma wpływ na zużycie pamięci, ponieważ powstaje wiele kopii zmiennych lokalnych.

5. Napisz procedurę `void reverse(lnode*& L)`, która odwróci listę L modyfikując jedynie pola `next` elementów listy L oraz wskaźnik L.

6. (2pkt) Napisz procedurę `lnode* merge(lnode* L1, lnode* L2)`, która złączy posortowane listy `L1` i `L2` w jedną posortowaną listę, zmieniając jedynie wartości pól `next` i używając tylko dwóch dodatkowych wskaźników. Ilość porównań nie powinna przekroczyć długości wynikowej listy.

7. Zapoznaj się z procedurą wstawiania klucza do drzewa BST. Jakie drzewo powstanie po wstawieniu do pustego drzewa BST liczb od 1 do n w kolejności rosnącej? Jaka potem będzie głębokość drzewa? Ile porównań kluczy wykonano w trakcie tworzenia tego drzewa? Jaka jest złożoność w tego procesu w notacji O? Uwaga: Element wstawiamy na pierwsze napotkane puste miejsce zaczynając od korzenia. Jeśli miejsce jest zajęte, to gdy element jest mniejszy od klucza w węźle, idziemy do lewego poddrzewa, a gdy większy lub równy – do prawego poddrzewa.

8. Uzasadnij, że w każdym drzewie BST zawsze ponad połowa wskaźników (pól `left` i `right`) jest równa `NULL`.

9. Ile maksymalnie węzłów może mieć drzewo BST o głębokości h? Wylicz dokładną wartość, przyjmując, że głębokość oznacza ilość poziomów, na których występują węzły (sam korzeń: $h = 1$, korzeń i dzieci: $h = 2$ ...). Wywnioskuj, jaka jest najmniejsza, a jaka największa głębokość drzewa binarnego o n węzłach?

10. W pliku `tree-2020.cc` znajdziesz funkcję `int height(node *t)`, która wyliczy głębokość (ilość poziomów na jakich występują węzły) drzewa BST. Jak zależy czas wykonania tej funkcji od ilości n węzłów drzewa i jego głębokości h?

11. Przeanalizuj operacje dla drzewa BST (`find`, `insert`, `remove`) zawarte w pliku `tree-2020.cc`. Jak ich pesymistyczna złożoność czasowa $T(h)$ zależy od głębokości drzewa h?

12. Implementacja usuwania węzła z drzewa binarnego działa wg następującego schematu:

    (a) jeśli usuwany węzeł nie ma dzieci, to go usuwamy a odpowiedni wskaźnik zmieniamy na `NULL`.

    (b) jeśli ma jedno dziecko, to go usuwamy, a odpowiedni wskaźnik w węźle rodzica zastępujemy wskaźnikiem na to dziecko.

    (c) jeśli ma dwoje dzieci, to nie usuwamy tego węzła, lecz najmniejszy element w jego prawym poddrzewie, a dane i klucz tego elementu wpisujemy do węzła, który miał być usunięty.

    Uzasadnij, dlaczego postępowanie wg punktu (c) nie psuje prawidłowego rosnącego porządku kluczy wypisywanych w porządku `inorder`.

13. Jak zmodyfikować operacje dla drzewa BST (`insert`, `remove`) bez użycia rekurencji, aby działały poprawnie dla drzewa o węzłach gdzie występuje też wskaźnik na ojca. `struct node{int x; node *left; node *right; node *parent;};`

14. Napisz rekurencyjną procedurę `void inorder_do(node *t,void f(node*))`, która wykona funkcję `f` na każdym węźle drzewa `t` w kolejności `in_order`.

15. Wiedząc, że `node` zawiera wskaźniki na rodzica `parent`, napisz nierekurencyjną wersję powyższej funkcji.

16. (2 pkt.) Zakładając, że w każdym węźle drzewa BST jest również wskaźnik na ojca, napisz klasę `iterator` oraz funkcje `iterator begin(node *t)` oraz `iterator end(node *t)`, które pozwolą wypisać wszystkie klucze z drzewa `t` za pomocą instrukcji:

    ```
    for(iterator begin(t); i!=end(t); i++)
        cout<< *i <<endl;
    ```

    Jedyną składową (w części prywatnej) powinien być wskaźnik na bieżący węzeł.

17. (3 pkt.) Zmodyfikuj `iterator` drzewa BST, tak by nie korzystał z pola `parent`. Wskazówka: do części prywatnej iteratora dodaj stos elementów typu `node*` zawierający węzły, powyżej bieżącego.