Algorytmy i Struktury danych (2023)

Lista zadań 2 (tablice i drzewa)

Pomocna przy wykonaniu tej listy jest implementacja drzewa BST `tree-2023-recursive.cc`. Plik jest załączony do maila. Można też go pobrać z folderu 'Wykład 2' w MS Teams lub z serwisu: `http://panoramx.ift.uni.wroc.pl`.

1. 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.

2. Bisekcja. 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 $n = 1, ..., 20$.

3. 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) rekurencyjnie oblicza maksimum `n-1` elementów i porównuje z ostatnim elementem;

   (c) dzieli tablicę na dwie części, rekurencyjnie znajduje ich maksima i wybiera większe z nich.

   Ile porównań między elementami tablicy n-elementowej 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. Która wersja jest więc najlepsza?

4. 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.

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

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

   (b) jeśli X ma jedno dziecko Y, to usuwamy X, a wskaźnik na X zastępujemy wskaźnikiem na Y.

   (c) jeśli X ma dwoje dzieci, to znajdujemy najmniejszy element Y w jego prawym poddrzewie, dane i klucz z węzła Y kopiujemy do X i usuwamy Y.

   Uzasadnij, dlaczego postępowanie wg punktu (c) nie psuje prawidłowego rosnącego porządku kluczy wypisywanych w porządku `inorder` i dlaczego Y ma co najwyżej jedno dziecko, więc do jego usunięcia można zastosować punkt (a) lub (b).

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

7. 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: $h = 2$ ...). Skorzystaj z wzoru na sumę ciągu geometrycznego. Wywnioskuj, jaka jest najmniejsza, a jaka największa głębokość drzewa binarnego o n węzłach?

8. Przeanalizuj operacje `find`, `insert`, `remove` zawarte w pliku `tree-2023-recursive.cc`. Jak ich pesymistyczna złożoność czasowa $T(h)$ zależy od głębokości drzewa h?

9. W pliku `tree-2023-recursive.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/lub jego głębokości h? To samo zadanie wykonaj też dla funkcji `int count(node *t)`.

Zadania programistyczne

1. 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;};`

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

3. Wiedząc, że `node` zawiera wskaźnik na rodzica `parent`, napisz nierekurencyjną wersję powyższej funkcji.

4. (2 pkt.) Bazując na rozwiązaniu poprzedniego zadania, napisz klasę `BSTiter` oraz funkcje `BSTiter begin(node *t)` oraz `BSTiter end(node *t)`, które pozwolą wypisać wszystkie klucze z drzewa `t` za pomocą instrukcji:

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

   oraz równoważnie:

   ```
   for(auto x:t)
       cout<< x <<endl;
   ```

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

5. (3 pkt.) Wykonaj poprzednie zadanie, tak by nie korzystać z pola `parent`, Zaimplementuj go w postaci szablonu, który będzie działał dla dowolnych węzłów zawierających: pola `key`, `left` i `right` .

   Wskazówka: do części prywatnej iteratora dodaj stos (`std::stack`) elementów typu `node*` zawierający węzły, powyżej bieżącego.