Algorytmy 2
Laboratorium: drzewo BST

Przemysław Klęsk

25 października 2020

1 Cel

Celem zadania jest wykonanie implementacji struktury danych nazywanej *drzewem wyszukiwań binarnych* (lub także *binarnym drzewem poszukiwań*) (ang. *binary search tree*). W drzewie BST każdy węzeł posiada (oprócz danych właściwych) wskaźniki na lewego i prawego potomka oraz wskaźnik na rodzica. Najistotniejszą własnością drzew BST jest to, że: klucz lewego potomka $<$ klucz rodzica $<$ klucz prawego potomka[1]. Własność ta musi być zachowana po zakończeniu dowolnej operacji dodawania lub usuwania na drzewie. To dzięki tej własności możliwe jest wyszukiwanie binarne poprzez porównywanie wartości zadanego klucza względem aktualnego węzła i wędrówkę w dół drzewa, odpowiednio w lewo lub w prawo, poczynając od korzenia.

Głównym zamierzeniem tej struktury danych jest szybkość wyszukiwania. Naturalnie chcielibyśmy, aby czas wyszukiwania *nie* skalował się liniowo wraz z liczbą elementów umieszczonych w drzewie, a jedynie wraz z logarytmem z tej liczby. Przy zapewnieniu odpowiedniego zrównoważenia drzewa BST, jego wysokość i tym samym czas wyszukiwania są właśnie rzędu $O(\log n)$. Jednym z mechanizmów równoważących drzewa BST jest algorytm DSW (skrót od nazwisk: Day–Stout–Warren) opierający się na wielokrotnym wykonywaniu operacji *rotacji*. Istnieją także samorównoważące się warianty drzew BST np. *drzewa czerwono-czarne*. Przy braku mechanizmów równoważących, drzewo może w pesymistycznym przypadku zdegenerować się do listy, co przełoży się na wyszukiwanie w czasie $O(n)$. Z drugiej strony warto także zaznaczyć, że jeżeli klucze danych, które umieszczamy w drzewie, pod względem statystycznym mają rozkład bliski równomiernemu, to powstające drzewo będzie w naturalny sposób zrównoważone.

W ramach niniejszego zadania oczekiwane jest wykonanie podstawowej implementacji drzewa BST bez równoważenia. Wymaganymi operacjami są: dodawanie, wyszukiwanie, usuwanie, a także przejścia po węzłach drzewa w porządku pre-order i in-order, oraz możliwość podania wysokości drzewa. Tradycyjnie, dodatkowym celem zadania jest wykonanie odpowiednich pomiarów czasowych w celu sprawdzenia teoretycznej złożoności obliczeniowej.

[1] możliwe arbitralne domknięcie jednej z nierówności

2 Instrukcje, wskazówki, podpowiedzi

1. Podobnie jak w poprzednich zadaniach dozwolone są implementacja strukturalna lub obiektowa, przy czym ponownie wymagane jest użycie mechanizmu szablonów (`template`) języka C++ dla zachowania ogólności.

2. Każdy węzeł drzewa powinien zawierać: dane właściwe (lub wskaźnik na nie), wskaźnik na rodzica, wskaźniki na lewego i prawego potomka.

3. Samo drzewo powinno przechowywać wskaźnik na korzeń i aktualny rozmiar.

4. Dla łatwości sprawdzenia prawidłowej konstrukcji drzewa zaleca się, aby każdy węzeł był wyposażony dodatkowo w pewien unikalny indeks całkowity. Pozwoli to uniknąć obserwowania samych adresów w pamięci RAM przy sprawdzaniu powiązań rodzic-dziecko oraz przy oglądaniu napisowej reprezentacji drzewa.

5. **Interfejs drzewa BST** powinien udostępniać następujące funkcje / metody:

   (a) dodanie nowego elementu do drzewa (argumenty: dane i informacja lub komparator definiujące klucz porządkowania / wyszukiwania — np. wskaźnik na funkcję),

   (b) wyszukanie elementu (argumenty: dane wzorcowe oraz informacja lub komparator definiujące klucz wyszukiwania; wynik: wskaźnik na odnaleziony element drzewa lub `NULL` w przypadku niepowodzenia),

   (c) usuwanie znalezionego wcześniej węzła drzewa (argumenty: wskaźnik na węzeł do usunięcia),

   (d) przejście pre-order drzewa (argumenty i sposób przekazania wyniku wg uznania programisty, możliwe m.in. zostawienie wyniku — porządku przejścia — na zewnętrznej liście przekazywanej przez wskaźnik w ramach rekurencji),

   (e) przejście in-order drzewa (argumenty i sposób przekazania wyniku jak wyżej),

   (f) czyszczenie drzewa tj. usunięcie wszystkich elementów,

   (g) wyznaczenie wysokości drzewa,

   (h) zwrócenie napisowej reprezentacji drzewa — np. funkcja / metoda `to_string(...)` (format wynikowego napisu nie musi odzwierciedlać struktury drzewa, ale powinien być czytelny dla prowadzącego, zawierać informacje o poszczególnych węzłach oraz powiązaniach rodzic-dziecko np. z wykorzystaniem indeksów całkowitych; odpowiednio małe drzewo należy wypisać w całości, większe w formie skróconej; wskazówka: budowę napisu można oprzeć np. na porządku pre-order).

   Uwaga: na potrzeby usuwania elementów — punkt (c) — wygodnym może być przygotowanie pomocniczej prywatnej metody znajdującej właściwego kandydata do zastąpienia usuwanego węzła (w przypadku ogólnym); kandydatem może być najmniejszy potomek (ze względu na klucz) w prawym poddrzewie usuwanego węzła lub największy potomek w lewym poddrzewie usuwanego węzła.

6. W programie można wykorzystać ogólne wskazówki z poprzednich zadań dotyczące:

   - dynamicznego zarządania pamięcią (`new`, `delete`) — w szczególności przemyślenia miejsc odpowiedzialnych za uwalnianie pamięci danych,
   - wydzielenia implementacji interfejsu drzewa BST do odrębnego pliku `.h`,
   - pracy z napisami (użycie typu `std::string`),
   - pomiaru czasu (funkcja `clock()` po dołączeniu `#include <time.h>`),
   - użycia wskaźników na funkcje,
   - generowania losowych danych (funkcje `rand()` i `srand(...)`).

7. Poniżej przedstawiono przykładową napisową reprezentację drzewa BST przechowującego liczby całkowite, które były dodawane do drzewa w następującej kolejności: `10`, `15`, `12`, `5`, `30`, `25`, `35`, `7`, `-2`, `33`.

```
binary search tree:
  size: 10
  height: 4
  {
  (0: [p: NULL, l: 3, r: 1], data: 10),
  (3: [p: 0, l: 8, r: 7], data: 5),
  (8: [p: 3, l: NULL, r: NULL], data: -2),
  (7: [p: 3, l: NULL, r: NULL], data: 7),
  (1: [p: 0, l: 2, r: 4], data: 15),
  (2: [p: 1, l: NULL, r: NULL], data: 12),
  (4: [p: 1, l: 5, r: 6], data: 30),
  (5: [p: 4, l: NULL, r: NULL], data: 25),
  (6: [p: 4, l: 9, r: NULL], data: 35),
  (9: [p: 6, l: NULL, r: NULL], data: 33)
  }
```

   W powyższym napisie symbole `p`, `l`, `r` oznaczają indeksy odpowiednio rodzica, lewego i prawego dziecka. Dane właściwe są podane na końcu każdego wiersza.

   W drugim przykładzie pokazanym poniżej, właściwe dane przechowywane w węzłach to pary: (liczba, znak). Tym razem drzewo powstało w wyniku dodawania kolejno następujących danych: `(3, w)`, `(1, t)`, `(5, y)`, `(9, u)`, `(7, x)`, `(3, z)`, `(6, u)`, `(3, k)`, `(9, v)`, `(5, m)`; uznając za klucz porządkowania liczbę będąca pierwszym elementem pary (a znak tylko w przypadku remisu).

```
binary search tree:
  size: 10
  height: 4
  {
  (0: [p: NULL, l: 1, r: 2], data: (3, w)),
  (1: [p: 0, l: NULL, r: 7], data: (1, t)),
  (7: [p: 1, l: NULL, r: NULL], data: (3, k)),
  (2: [p: 0, l: 5, r: 3], data: (5, y)),
  (5: [p: 2, l: NULL, r: 9], data: (3, z)),
  (9: [p: 5, l: NULL, r: NULL], data: (5, m)),
  (3: [p: 2, l: 4, r: 8], data: (9, u)),
  (4: [p: 3, l: 6, r: NULL], data: (7, x)),
  (6: [p: 4, l: NULL, r: NULL], data: (6, u)),
  (8: [p: 3, l: NULL, r: NULL], data: (9, v))
  }
```

8. W ramach ćwiczenia zaleca się spróbować wyrysowywać na papierze uzyskiwane drzewa na podstawie ich reprezentacji napisowej (np. po kolejnych operacjach dodawania).

3 Zawartość funkcji `main()`

Główny eksperyment zawarty w funkcji `main()` ma polegać na: wielokrotnym dodawaniu coraz większej liczby elementów (danych) do drzewa BST (rzędy wielkości od 10^1 aż do 10^7), a następnie wyszukiwaniu w nim pewnych losowych danych. Należy raportować czasy dodawania i wyszukiwania (całkowite i średnie) oraz wysokości otrzymanych drzew. Dodatkowo, należy raportować: stosunek wysokości drzewa do rozmiaru danych, logartym (o podstawie 2) z rozmiaru danych, i wreszcie stosunek wysokości drzewa do logarytmu z rozmiaru danych.

W celu uniknięcia zbyt częstych kolizji (remisów) kluczy, sugeruje się rozszerzenie zakresu wartości losowych dla generowanych danych (uwaga: funkcja `rand(...)` ma niewygodne ograniczenie do stałej `RAND_MAX` — należy przemyśleć pewne obejście tego ograniczenia).

Poglądowy schemat eksperymentu:

```
int main()
{
    ...
    const int MAX_ORDER = 7; // maksymalny rzad wielkosci dodawanych danych
    binary_search_tree<some_object*>* bst = new binary_search_tree<some_object*>(); // stworzenie
        drzewa
    for (int o = 1; o <= MAX_ORDER; o++)
    {
        const int n = pow(10, o); // rozmiar danych

        // dodawanie do drzewa
        clock_t t1 = clock();
        for (int i = 0; i < n; i++)
```

```
        {
            some_object* so = ... // losowe dane
            bst->add(so, some_objects_cmp); // dodanie (drugi argument to wskaznik na komparator)
        }
        clock_t t2 = clock();
        ... // wypis na ekran aktualnej postaci drzewa (skrotowej) wraz z pomiarami czasowymi i w/w
             wielkosciami

        // wyszukiwanie
        const int m = pow(10, 4); // liczba prob wyszukiwania
        int hits = 0; // liczba trafien
        t1 = clock();
        for (int i = 0; i < m; i++)
        {
            some_object* so = ... // losowe dane jako wzorzec do wyszukiwania (obiekt chwilowy)
            binary_search_tree_node<some_object*>* result = bst->find(so, some_objects_cmp);
            if (result != NULL)
                hits++;
            delete so;
        }
        t2 = clock();
        ... // wypis na ekran pomiarow czasowych i liczby trafien

        bst->clear(true); // czyszczenie drzewa wraz z uwalnianiem pamieci danych
    }
    delete bst;
    return 0;
}
```

4 Sprawdzenie antyplagiatowe — przygotowanie wiadomości e-mail do wysłania

1. Kod źródłowy programu po sprawdzeniu przez prowadzącego zajęcia laboratoryjne musi zostać przesłany na adres *algo2@zut.edu.pl*.

2. Plik z kodem źródłowym musi mieć nazwę wg schematu: `nr_albumu.algo2.nr_lab.main.c` (plik może mieć rozszerzenie `.c` lub `.cpp`). Przykład: `123456.algo2.lab06.main.c` (szóste zadanie laboratoryjne studenta o numerze albumu 123456). Jeżeli kod źródłowy programu składa się z wielu plików, to należy stworzyć jeden plik, umieszczając w nim kody wszystkich plików składowych.

3. Plik musi zostać wysłany z poczty ZUT (zut.edu.pl).

4. Temat maila musi mieć postać: `ALGO2 IS1 XXXY LAB06`, gdzie `XXXY` to numer grupy (np. `ALGO2 IS1 210C LAB06`).

5. W pierwszych trzech liniach pliku z kodem źródłowym w komentarzach muszą znaleźć się:

   - informacja identyczna z zamieszczoną w temacie maila (linia 1),
   - imię i nazwisko autora (linia 2),
   - adres e-mail (linia 3).

6. Mail nie może zawierać żadnej treści (tylko załącznik).

7. W razie wykrycia plagiatu, wszytkie uwikłane osoby otrzymają za dane zadanie ocenę 0 punktów (co jest gorsze niż ocena 2 w skali $\{2, 3, 3.5, 4, 4.5, 5\}$).