Algorytmy 2
Laboratorium: drzewo czerwono-czarne

Przemysław Klęsk

30 września 2019

1 Cel

Celem zadania jest wykonanie uproszczonej implementacji struktury danych nazywanej *drzewem czerwono-czarnym* (ang. *red-black tree*). Jest ona przykładem samorównoważącego się binarnego drzewa poszukiwań (ang. *self-balancing binary search tree*). Poza standardową własnością drzew BST (klucz lewego potomka $<$ klucz rodzica $<$ klucz prawego potomka[1]), na drzewa czerwono-czarne nakłada się następujące dodatkowe własności (warunki):

1. każdy węzeł ma kolor (czerwony lub czarny),
2. korzeń jest czarny,
3. każdy liść jest czarny (przy czym jako liście *nie* są traktowane najniższe węzły, zaś puste wskaźniki potomne odchodzące od nich — tzw. NULLe lub NILe),
4. czerwony węzeł musi mieć czarne dzieci,
5. każda ścieżka z dowolnego ustalonego węzła do dowolnego osiągalnego liścia ma tyle samo czarnych węzłów.

Własności te muszą oczywiście być zachowane po zakończeniu dowolnej operacji dodawania lub usuwania na drzewie, nawet jeśli w trakcie są chwilowo popsute. Tak naprawdę kluczową rolę odgrywa kombinacja dwóch ostatnich własności (4 i 5), i gwarantuje ona (dowód na wykładzie), że wysokość drzewa czerwono-czarnego w dowolnym momencie wynosi co najwyżej $2\log_2(n+1)$, gdzie n to liczba przechowywanych elementów. Implikuje to w ogólności logarytmiczny czas wyszukiwania $O(\log n)$. Jednocześnie można również pokazać, że wszelkie potrzebne manipulacje naprawcze (przekolorowania i rotacje) wymagają nakładu pracy proporcjonalnego co najwyżej do wysokości drzewa, a tym samym operacje dodawania i usuwania (zawierające w sobie naprawy) mają także złożoność $O(\log n)$.

Dla uproszczenia, w ramach niniejszego zadania pominięta zostanie implementacja operacji usuwania z drzewa czerwono-czarnego. Wymagane są: dodawanie (wraz z naprawianiem — przekolorowania

[1] możliwe arbitralne domknięcie jednej z nierówności

i rotacje), wyszukiwanie, przejścia pre-order i in-order, zbadanie wysokości drzewa. Tradycyjnie, dodatkowym celem zadania jest wykonanie odpowiednich pomiarów czasowych w celu sprawdzenia teoretycznej złożoności obliczeniowej.

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, flagę logiczną określającą kolor (czerwony / czarny).

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.

5. **Interfejs drzewa czerwono-czarnego** (uproszczony) powinien udostępniać następujące funkcje / metody:

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

   (b) 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),

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

   (d) czyszczenie drzewa tj. usunięcie wszystkich elementów,

   (e) wyznaczenie wysokości drzewa,

   (f) dodanie nowego elementu do drzewa (argumenty: dane i informacja lub komparator definiujące klucz porządkowania)

   (g) 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 kolorach poszczególnych węzłów 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).

   (h) rotację w lewo wskazanej pary dziecko-rodzic (argumenty: wskaźnik na dziecko, wskaźnik na rodzica),

(i) rotację w prawo wskazanej pary dziecko-rodzic (argumenty: jak wyżej).

Punkty (h), (i) reprezentują narzędziowe funkcje naprawcze (na użytek wewnętrzny drzewa) — nie powinny być one używane bezpośrednio przez użytkownia zaś wywoływane jedynie z wnętrza funkcji (f) dodającej nowy element.

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 czerwono-czarnego 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 czerwono-czarnego, o której mowa w punkcie (g) interfejsu:

```
red black tree:
  size: 8
  {
  (2: [black, p: NULL, l: 0, r: 6] (62, a)),
  (0: [black, p: 2, l: 4, r: 3] (55, q)),
  (4: [red, p: 0, l: NULL, r: NULL] (35, d)),
  (3: [red, p: 0, l: NULL, r: NULL] (57, n)),
  (6: [red, p: 2, l: 1, r: 5] (72, v)),
  (1: [black, p: 6, l: NULL, r: NULL] (69, u)),
  (5: [black, p: 6, l: 7, r: NULL] (83, i)),
  (7: [red, p: 5, l: NULL, r: NULL] (74, x))
  }
```

   W powyższym napisie symbole `p`, `l`, `r` oznaczają indeksy odpowiednio rodzica, lewego i prawego dziecka. W przykładzie właściwe dane przechowywane w węzłach to pary (liczba, znak) wypisane przy końcu każdego wiersza. Powyższe drzewo powstało w wyniku dodawania kolejno następujących danych: `(55, q)`, `(69, u)`, `(62, a)`, `(57, n)`, `(35, d)`, `(83, i)`, `(72, v)`, `(74, x)`, uznając za klucz porządkowania liczbę będąca pierwszym elementem pary (a znak tylko w przypadku remisu; tu ten przypadek nie występuje).

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 czerwono-czarnego (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.

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
    red_black_tree<some_object*>* rbt = new red_black_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
            rbt->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 wysokoscia oraz pomiarow
            czasowych

        // 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)
            red_black_tree_node<some_object*>* result = rbt->find(so, some_objects_cmp);
            if (result != NULL)
                hits++;
            delete so;
        }
        t2 = clock();
        ... // wypis na ekran pomiarow czasowych i liczby trafien

        rbt->clear(true); // czyszczenie drzewa wraz z uwalnianiem pamieci danych
    }
    delete rbt;
    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 $\{2, 3, 3.5, 4, 4.5, 5\}$).