Organizacja i Architektura Komputerów
Projekt

Uladzimir Kaviaka

06 czerwca 2021

Spis treści

1 Wstęp

Tematem mojego projektu jest optymalizacja zużycia procesora CPU. Każdy wykonywany algorytm potrzebuje procesora, więc żeby zmniejszyć zużycie procesora nasze algorytmy muszą być napisane w poprawny sposób. Oprócz tego każdy algorytm jest wykonywany nad czymś, naprzykład sotrowanie jest wykonywane nad ciągiem liczb.

W tym projekcie będą przedstawione algorytmy operujące na drzewie binarnych poszukiwań, wiadomo że to drzewo może mieć rózny stopień rownoważenia i jeżeli nasze algorytmu zaimplementowane w najbardziej poprawny sposób ale operujemy na strukturze nieoptymalnej to wtedy jest oczywiste że dostaniemy nie najlepsze wyniki czasowe. Więc żeby poprawny algorym był najbardziej efektywnym musimy przeprowadzać operacje na optymalizowanej strukturze. Naprzykład algorytm sotrtowania przez wstawanie daje bardzo dobre wyniki gdy mamy do czynienia z ciągiem który już jest częściowo posortowany. Analogiczny przypadek z drzewem BST, gdy nasze drzewo jest zrównoważone wszystkie algorytmy mają najbardziej optymalną złożoność czasową.

Czym jest wydajność algorytmiczna?

Wydajność algorytmiczna - jest właściwością algorytmu która odnosi się do ilości zasobów obliczeniowych używanych przez algorytm.

Wydajność algorytmu można zmierzyć na podstawie wykorzystania różnych zasobów, ale żeby określić wykorzystanie zasobów algorytm musi być przetestowany. Logicznym jest to żeby uzyskać maksymalną wydajność algorytmu zużycie zasobów trzeba doprowadzić do minimum.

Skąd się wzięlo to słowo - wydajność?

Znaczenie wydajności zostało podkreślone przez Adę Lovelace w odniesieniu do mechanicznego silnika analitycznego Babbage'a

> *W prawie każdym obliczeniu możliwa jest wielka różnorodność ustaleń dotyczących następujących po sobie procesów, a różne czynniki muszą wpływać na wybór spośród nich dla celów silnika obliczeniowego. Jednym z zasadniczych celów jest wybranie takiego układu, który będzie miał tendencję*

do ograniczania się do minimalny czas niezbędny do wykonania obliczeń.

Wczesce komputery były bardzo ograniczone przez szybkość operacji zarówno jaki i ilość dostępnej pamięci. W wyniku tego zaistniał tak zwany **Kompromis czasowo-przestrzenny**. Idea tego kompromisu poelgała na tym że zadanie można wykonać na dwa sposoby w zależności od potrzeb:

1. Zadanie wykonywane bardzo szybko ale wymaga dużo pamięci
2. Zadanie wykonywane wolniej ale nie wymaga dużo pamięci.

W tych czasach inżynierzy wyciągnęli z tego kompromisu swoje rozwiązanie: trzeba używać najszybszego algorytmu który zmieści się w dostępnej pamięci.

Jaki algorytm możemy nazwać wydajnym?

Algorytm jest uznawany za wydajny jeżeli jego zużycie zasobów jest na poziomie aceptowanym lub niżej tego poziomu. Ale co znaczy poziom akceptowany? Poziom akceptowany to taki poziom który opisuje rozsądny czas i rozsądne użycie zasobów komputera dla danej wielkości instancji.

Zasoby używane przez algorytm można mierzyć na wiele sposobów, zaś dwie najczęstsze miary to szybkość i zużycie pamięci. Te miary zależa od rozmiaru danych wejściowych oraz od struktury na której te operacje są przeprowadzane.

Czym jest złożoność asymptotyczna?

W analizie algorytmów często stosuje się złożoność asymptotyczna. Najczęściej używana notacja do opisania efektywności algorytmów jest notacja *Big O notation* reprezentująca złożoność algorytmu w funkcji wielkości danych wejściowych. Ta notacja jest asymptotyczną miarą skoro czas potrzebny algorytmowi jest proporcjonalny do wielkości danych wejściowych oraz pozwala pomijać wyrażenia niższego rzędu które przycinają się mniej niż do wzrostu funkcji która rośnie arbitralnie.

Przykładowe złożoności asymptotyczne

W tabeli poniżej pokazane niektóre złożoności wraz z algorytmem który pokazuje taką złożoność.

Tabela 1: Przykładowe złożoności asymptotyczne

O(1)	Stała	Usunięcie pierwszego elementu z tablicy
O(log(n)))	Logarytmiczna	Dodanie elementu do zrównoważonego drzewa BST
O(n)	Liniowa	Usunięcie ostatniego elementu z tablicy
O(nlog(n))	Quasilinearna	Quicksort
O(n^2)	Kwadratowa	Sortowanie bąbełkowe
O(c^n)	Wykładnicza	Wyszukiwanie brute-force

Optymalizacja programu - czym jest?

Optymalizacja - proces modyfikacji systmeu oprogramowania do jakiegoś aspektu działać bardziej efektywnie lub używać mniej zasobów. Program nie może być całkiem optymalny a tylko w odniesieniu do określonej miary jakości, jak było omówiono powyżej, program może być szybki ale wyamgać dużo pamięci i tutaj jest widoczna ta różnica, czyli ze względu czasowego program jest optymalny ze strony użycia zasobów nie jest optymalny. I inny przykład kiedy algorytm jest wolniejszy ale nie wymaga dużo pamięci, już w tym przykładzie ze względu użycia pamięci program jest optymalny ale ze względu na czas wykonania - nie.

Poziomy optymalizacji

Optymalizacja może przebiegać na róznych poziomach. Zazwyczaj wyższe poziomy mają większy wpływ i trudniej jest je zmienić w późniejszym etapie projektu. Więc optymalizacja powinna przebiegać od najwyższego poziomu do najniższego.

1. Poziom projektu

Na najwyższym poziomie optymalizacji jest optymalizacja projektu. Optymalizować porjekt można ze względu na dostępne zasoby, ograniczenia, oczekiwane użycie. Wybór platformy i języka programowania występuje na tym poziomie a ich zamiana wymaga przepisania dużej ilości kodu. Chociaż system modułowy może trochę ułatwić sytuację. Naprzykład program w języku Python może przepisać sekcje krytyczne dla języka C.

2. Poziom algorytmów i struktur

Następnym poziomem jest dobór odpowiednich algorytmów oraz

struktur danych oraz wydajna optymalizacja tych algrytmów oraz struktur. W przypadku algorytmów najczęściej trzeba zapewnić to aby te algorytmu miały złożoność O(1), O(log(n)) lub O(n), algorytmow o złożoności czasowej O(n^2) trzeba unikać. Odnośnie algorytmów liniowych, te algorytmy często zastępowane algorytmami stałymi (O(1)) lub algorytmami logarytmicznymi (O(log(n)) wynika to z tego, że algorytmy liniowe powodują problemy jeżeli często są wywoływane.

3. Poziom kodu

Możliwe że nikt nad tym się nie zastanawia ale trzeba zwracać uwagę na to co pisze się w kodzie, bo konkretna logika która jest opisana na różne sposoby na poziomie kodu może nie przcyiągać uwagi. Naprzykład mamy nieskończoną pętlę while(1) oraz for(;;). W przypadku pętli while kompilator C oceniał warunek czy jest jedynka i następnie robił skok, podczas gdy for(;;) miał skok bezwarunkowy.

4. Poziom kompilacji

Użycie optymalizującego kompilatora zapewnia optymalizację w takim stopniu w jakim to kompilator może przedwidzieć.

5. Poziom zespołu

Na najniższym poziomie znajduje się kod maszynowy, użycie języka assemblera może stworzyć nabardziej wydajny i efektywny kod, jeżeli programista umie posługiwać się instrukcjami tego języka. Programy obecnie bardzo rzadko pisane w kodzie assemblera ze względu na czas i koszty. Większość jest pisana na językach wysokiego poziomy następnie kompilowana do asemblera i już na poziomie maszynowym była optymalizowana. Przy użyciu nowoczesnych kompilatorów trudniej jest napisać kod który będzie bardziej optymalny niż ten który wygeneruje kompilator

Wąskie gardła - czym są?

Podczas optymalizacji możemy zaobserwować tzw. zjawisko "wąskiego gardła", czynnik który ogranicza optymalną wydajność programu. W kodzie to zwykle "punkt aktywny" czyli ten rozdział programu który potrzebuje jak najwięcej zasobów

2 Cele projektu

Napisanie oprogramowania, na podstawie którego można będzie wskazać metody optymalizacji zużycia procesora CPU.

3 Narzędzia do realizacji projektu

1. Wybrany język programowania — C++
2. System operacyjny — Windows 10
3. Kompilator — GNU GCC
4. Środowisko — Code::Blocks 20.03
6. Struktura do badania optymalizacji — Drzewo BST
7. Algorytmy — usuwanie, dodawanie, wyszukiwanie

4 Założenia

Pisane programy powinny być poprawne, ale najważniejszą cechą jest szybkość wykonywania programu. Na przykład program, którego zadaniem jest nakładanie filtrów na obrazek, wydajność jest najważniejszym atrybutem. Kompilatory samodzielnie próbują optymalizować kod, dla przykładu kompilator GCC z parametrem -Og wykonuje podstawową optymalizację.

Dobrym parametrem oceny użycia procesora jest ilość taktów, które procesor wykonuje, żeby wykonać pracę na jednym elemencie, można wprowadzić zmienną, jak ilość taktów na jeden elementu (CFE – cycle for element), wtedy CFE – 4, będzie mówiło o tym że processor wykonuje 4 takty żeby wykonać pracę na jednym elemencie.

Metodami redukcji użycia procesora mogą na przykład być:

- Usunięcie niepotrzebnych zwrotów do pamięci.

  Chodzi tu o wykorzystanie rejestrów procesora zamiast pamięci, przykładem może być dodanie wszystkich elementów tablicy. Mając globalną zmienną np. SUM w każdej iteracji można wykonywać SUM += A[i] i tutaj widoczne, że w sumie 3 zwroty do pamięci: pobierz SUM, pobierz A[i], aktualizuj SUM, ale można zwrócić uwagę, że całkiem niepotrzebne jest wczytywanie i zapisanie SUM po kolejnej iteracji. Zamiast tego można

wprowadzić akumulator, do którego będziemy wpisywali kolejne elementy tablicy i tylko po wykonaniu pętli zapisać wartość akumulatora w zmiennej SUM. Kompilator samodzielnie umieści akumulator w rejestrze, co pozwoli uniknąć niepotrzebnych zwrotów do pamięci.

- Optymalizacja pętli.

  Przykładem może być też dodanie elementów tablicy. Mając tablicę ze 100 elementów można napisać zwykły for który będzie dodawał po kolei do akumulatora kolejne wartości tablicy, ale procesor wykonując taki algorytm nie tylko wykonuje dodawanie elementów do akumulatora, ale dodatkowo wykonuje instrukcje obsługi pętli: inkrementowanie licznika, sprawdzenie warunku pętli, wejście do kolejnej iteracji. Więc metoda do sprawdzenia może polegać na tym, że zwiekszamy ciało pętli redukując ilość iteracji pętli. Co daje zysk, że procesor w jednej iteracji wykonuje znacznie więcej korzystnej pracy oraz wykonanie obsługi pętli jest już optymalizowane.

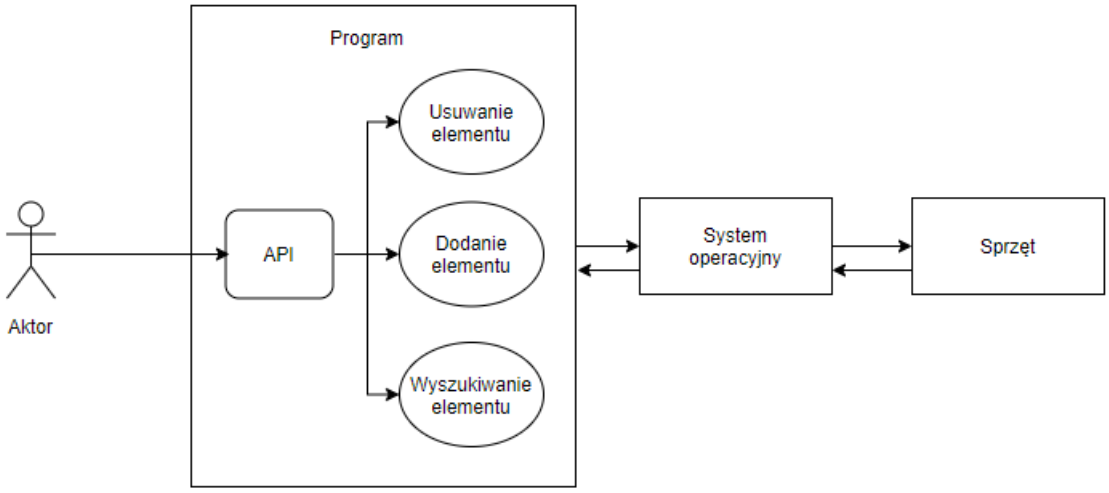
- Zmniejszenie wywołan funkcji

  Usunięcie niepotrzebnych drobnych funkcji

Drzewo binarnych poszukiwań jest przydatną strukturą do badania efektywności algorytmów, czas wykonywania podstawowych operacji, takich jak: usuwanie, dodawanie, wyszukiwanie zależy od ilości elementów w drzewie.

Te trzy podstawowe algorytmy można optymalizować używając operacji równoważenia drzewa, dla porównania, złożoność wyszukiwania elementu w drzewie BST niezrównoważonym wynosi w najgorszym przypadku O(n) zaś po zastosowaniu algorytmu równoważenia dla drzewa, złożoność wyszukiwania już będzie optymalna i będzie wynosiła O(log(n)). Analogiczne rezultaty będą dla pozostałych operacji na drzewie binarnym po zastosowaniu równoważenia.

5 Architektura oprogramowania (uogólniona)



6 Proof of Concept

Idea Proof-of-Concept

Demonstracja tego, że uda się stworzyć oprogramowanie, które będzie umowżliwało realizację początkowych założeń.

Moim zadaniem jest zaprojektowanie algorytmów działających na drzewie BST za pomocą których można wskazać w jaki sposób można zredukować zużycie procesora CPU. Realizowane będą metody usuwania, dodawania oraz wyszukania elementu dla zwykłego drzewa oraz drzewa po zrównoważeniu, te założenia da się zaimplementować posługując się znanymi algorytmami. Jako pomóc w implemetacji tych algorytmów będzie książka:

Thomas H.Cormen, Clifford Stein, Ronald L. Rivest, Charles E. Leiserson, Introduction to Algorithms, 2009

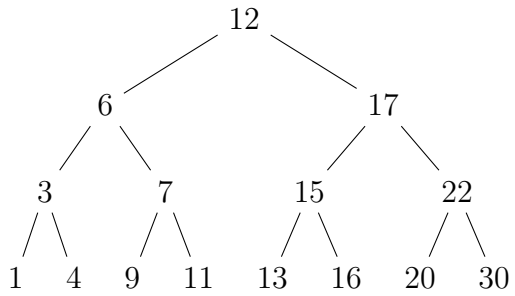
W danym miejscu chciałbym zaprezentować możliwość zaimplementowania algorytmu wyszukiwania w drzewie za pomocą "pseudokodu" który da się bez problemu zinterpretować na rzeczywisty język programowania:

```
BST_TREE_SEARCH(Node, Key):
if(Node == Null) albo (Node->Key == Key)
    zwróć Node;
Jeśli Key<Node->Key
    zwróć BST_TREE_SEARCH(Node->Left, Key);
zwróć BST_TREE_SEARCH(Node->Right, Key);
```

Ten algorytm już mam zaimplementowany w języku C++, więc poniżej jest realizacja tego algroytmu w wersji iteracyjnej zamiast rekurencyjnej

```
nodeBST* bstTree::findElement(int element) {
    if(root==NULL) {
        return NULL;
    }
    else
    {
        nodeBST* tmpNode = root;
        while(tmpNode!=NULL) {
            if(element > tmpNode->value) {
                tmpNode = tmpNode->rightItem;
            }
            else
            {
                if(element<tmpNode->value) tmpNode = tmpNode->leftItem;
                    else
                    {
                        return tmpNode;
                    }
            }
        }
        return NULL;
    }
}
```

Co już pozwala zrobić test dla przykładowego drzewa BST, niech nasze drzewo wygląda następująco:



Teraz trzeba to drzewo przenieść do pamięci komputera i wyświetlic za pomocą konsoli, po czym można sprawdzić algorytm wyszukiwania dla wartości 15:

```
            30
        22
            20
    17
            16
        15
            13
12
            11
        9
            7
    6
            4
        3
            1


Element 15 znaleziony

Process returned 0 (0x0)   execution time : 0.023 s
Press any key to continue.
```

Wyszukanie wartości w drzewie

Patrzać na zrzut z konsoli można zobaczyć że udało się zaimplementować algorytm poprawnie.

7 Opis testów

W poprzednim punkcie była pokazana możliwość zaimplementowania założeń teoretycznych odnośnie potrzebnych algorytmów za pomocą języka programowania C++. W tym punkcie będą omówione testy wszystkich algoytmów na drzewie dla następnego wykonania projektu końcowego.

- Dodanie elementów do drzewa

- Usuwanie elementów z drzewa
- Wyszukiwanie elemntów w drzewie
- Równoważenie drzewa

Test 1. Dodanie elementów do drzewa
Założenie: Algorytm musi poprawnie wykonywać dodanie elementów do drzewa.
Uzasadnienie: Żeby wykonać poprawne dodanie elementu, trzeba sprawdzić czy jest mniejsze od korzenia czy większe w zależności od tego będzie wstawiane do lewego lub prawego poddrzewa. W przypadku gdy w drzewie nie ma żadnego elementu, wstawiany element musi być dodany jako korzeń drzewa. Po wykonaniu wstawiania nowego elementu drzewo powinno wciąż spełniać warunki drzewa BST.

Test 2. Usuwanie elementów z drzewa
Założenie: Algorytm musi poprawnie usunąć dowolny element z drzewa
Uzasadnienie: Żeby usuwanie było wykonane poprawnie trzeba rozpatrzyć trzy przypadki które mogą powstać podczas wykonania tej operacji:

- Wierzchołek ma dwóch potomków
- Wierzchołek ma jednego potomka
- Wierzchołek nie ma potmoków

W zależności od przypadku trzeba poprawnie usunąć wierzchołek używając metody poprzednika albo następnika. Po wykonaniu usuwania elementu drzewo powinno wciąż spełniać warunki drzewa BST

Test 3. Wyszukiwanie elementow w drzewie
Założenie: Algorytm musi poprawnie wyszukać element w drzewie
Uzasadnienie: Żeby znaleźć konkretny element trzeba najpierw sprawdzić w jakim z poddrzew, lewym czy prawym głównego drzewa musi się znaleźć w zależności od tego czy jest mniejszy lub większy od korzenia. Po czym powtarzając to samo póki algorytm nie zwróci wskaźnik na znaleziony element w przypadku gdy nasz element był w drzewie albo zwróci wartość NULL w przypadku gdy nasz element

nie był w drzewie.

Test 4. Równoważenie drzewa
Założenie: Algorytm musi wykonać równoważenie używając algorymtu Day-Stout-Warren.
Uzasadnienie: Po wykonaniu algorytmu równoważenia DSW złożoności czasowe powyższych algorytmów muszą być zgodne z teoretycznymi. Wykonanie równoważenia polega na dwóch etapach:

- Prostowanie drzewa

  Prostowanie drzewa powinno za pomocą rotacji w prawo prowadzić do tego że drzewo będzie w postaci listy. Drzewo sprowadzone do takiej postaci nazywa się "kręgosłupem".

- Równoważenie drzewa

  Równoważenie które jest drugim etapem algorymtu DSW, przywraca postać drzewa dla listy powstałej po prostowaniu drzewa używając tutaj prawych zarówno jak i lewych rotacji. Wykonuje się na co drugim węźle względm jego rodzica, każdy taki przebieg skraca długość linii o połowę. Po ukończeniu tego etapu powstaje drzewo *doskonale zrównoważone*. Które jest optymalniejsze niż drzewo niezrównoważone.

8 Optymalizacja zużycia CPU

Jak było opisane w Etapie 1 projektu, sprawdzanie zużycia procesora będzie się odbywało na podstawie umownej jesdnostki **CFE** (cycles for element), która mówi o tym ile taktów procesora jest potrzebne żeby wykonać operację na jednym elemencie, oraz będzie użyta umowna jednostka **CPU TIME** która będzie pokazywała ile czasu było potrzbne procesorowi aby wykonać algorytm.

Do obliczania ilości taktów procesora będę używał funkcji **rdtsc**
Do odmierzenia czasu będę używał funkcji **clock** z biblioteki C która się nazywa time.h

Przykładowy kod demonstrujący użycie tych funkcji:

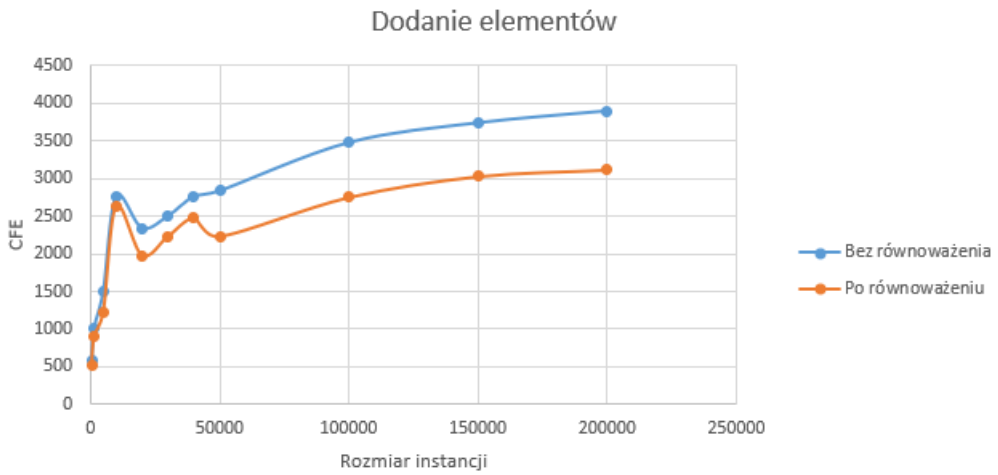
```
int main():
    SIZE = 100000;
    double time1 = clock() / (double)CLOCKS_PER_SEC;
    long cycles1 = rdtsc();
    /* Algorytm którego parametry mierzymy */
        Wyszukiwanie elementu w drzewie
    /* Koniec algorytmu */
    double time2 = clock() / (double)CLOCKS_PER_SEC;
    long cycles2 = rdtsc();
    long cycles = cycles2 - cycles1;      // Ilość taktów
    double cfe = cycles / (double)SIZE;   // Taktów na jeden element
    double cpu_time = time2 - time1;      // Czas pracy CPU
    printf("TIME_CPU: %.2f sec.", cpu_time);
    printf("CFE:      %.2f", cfe);
```

Ten prykładowy program pokazuje w jaki sposób te wartości można obliczyć, zasada jest taka sama w przypadku zmiennych CFE oraz CPU TIME, przed rozpoczęciem algorytmu przechowywamy w zmiennych wartości początkowe naszych parametrów, po wykonaniu algorytmu aktualizaujemy nasze wartośći i obliczamy ich różnicę.

9 Testy

Mając wszystkie algorytmy już zaimplementowane, można prowadzić testy i mierzyć paramtery według których można będzie wskazać, czy faktycznie optymalizacja struktur danych daje zysk w użyciu procesora. Jednocześnie, czy można mając wykresy, jednoznacznie powiedzieć, że optymalizacja struktury pozwala zmniejszyć zużycie procesora CPU. Poniżej są wykresy demonstrujące najpierw zależność CFE (cykli na element) w zależności od algorytmu i struktury (Drzewa BST) optymalizowanej oraz nieoptymalizowanej. Poniżej zależność CPU TIME od algorytmów oraz optymalizacji struktury.

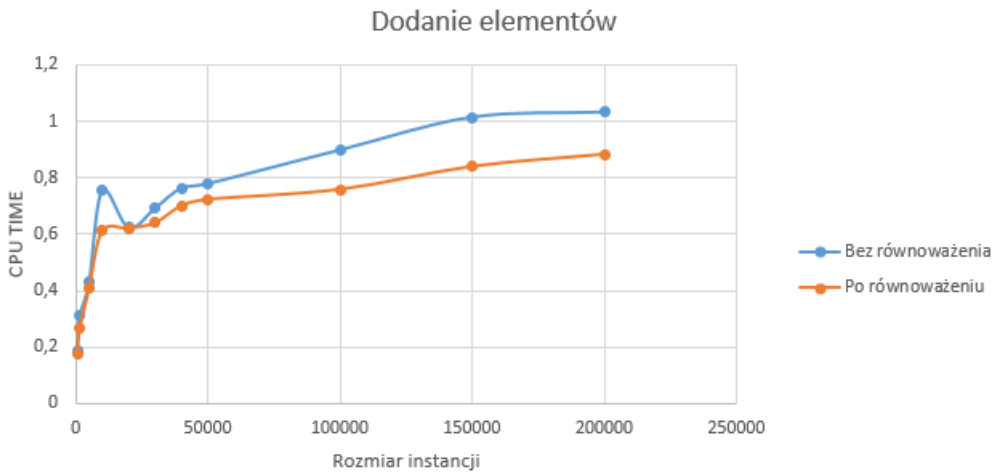
Test 1. Dodanie elemtnów do drzewa - CFE



Ten wykres pokazuje że faktycznie liczba cykli procesora na jeden element jest mniejsza dla drzewa całkiem zrównoważonego. To można wytłumaczyć w ten sposób: żeby dodać element do drzewa niezrównoważonego w najgorszym przypadku musimy przejść przez n poziomów drzewa, zaś mając drzewo zrównoważone ilośc poziomów skraca się do log(n+1), gdzie n to ilość węzłów.

Wniosek: Algorytm dodawania dla optymalizowanej struktury daje zysk prawie o **17%** mniej cykli procesora jest potrzebne aby wstawić element do optyamlizowanej struktury

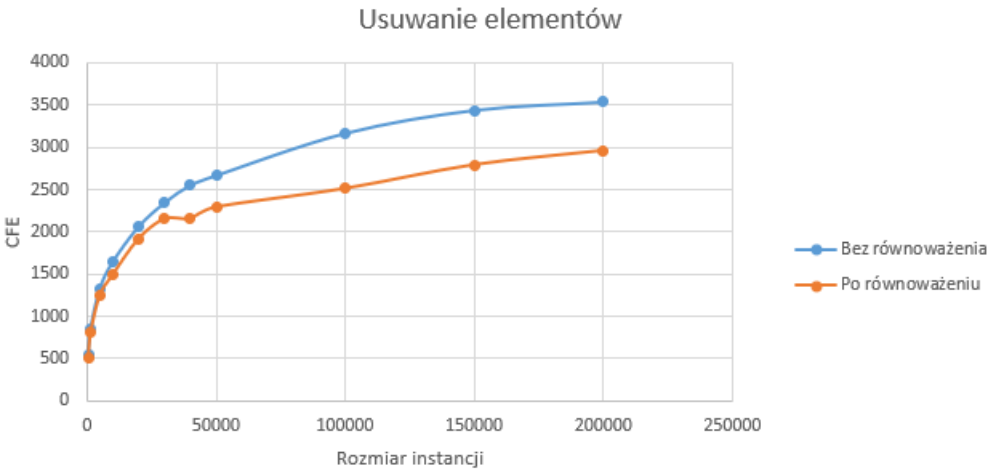
Test 1. Dodanie elemtnów do drzewa - CPU TIME



Ten wykres pokazuje że czas wstaiwania elementów do optymalizowanej struktury jest mniejszy niż do struktury bez optymalizacji. To tłumaczy się w ten sam sposób, złożoność czasowa dodania elementu do nieoptymalizowanego drzewa jest O(n) zaś dla optymalizowanego O(log(n)).

Wniosek: Algorytm dodawania dla optymalizowanej struktury daje zysk o **12%** mniej czasu procesora jest potrzebne aby wstawić element do optyamlizowanej struktury

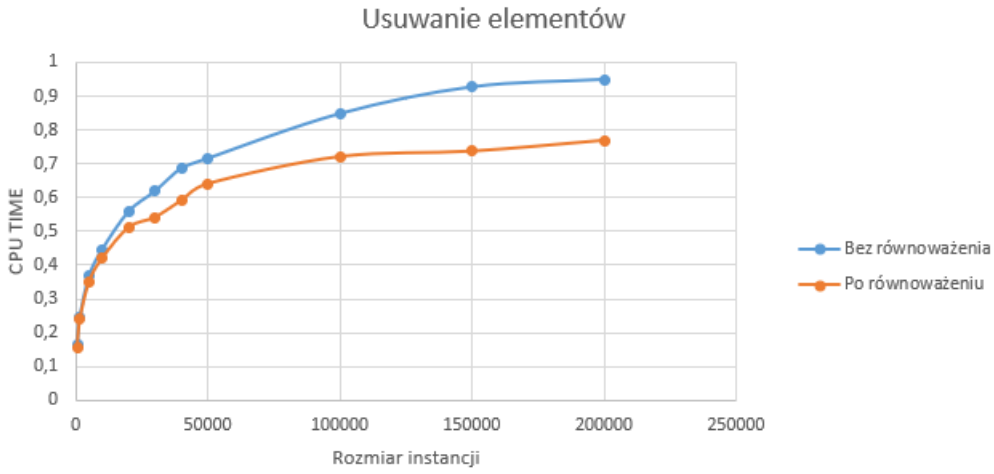
Test 2. Usuwanie elemtnów z drzewa - CFE



Ten wykres pokazuje również że liczba cykli procesora jest mniejsza w przypadku usuwania elementów ze struktury zoptymalizowanej. Wytłumaczyć to można w ten sposób: aby usunąc element trzeba go najpierw znaleźć, żeby znaleźć element w niezrównoważonym drzewie w najgorszym przypadku musimy sprawdzić n węzłów zaś w przypadku zrównoważonego ta liczba redukuje się do log(n).

Wniosek: Algorytm usuwania dla optymalizowanej struktury daje zysk o **14%** mniej cykli procesora jest potrzebne aby usunąć element z optymalizowanej struktury

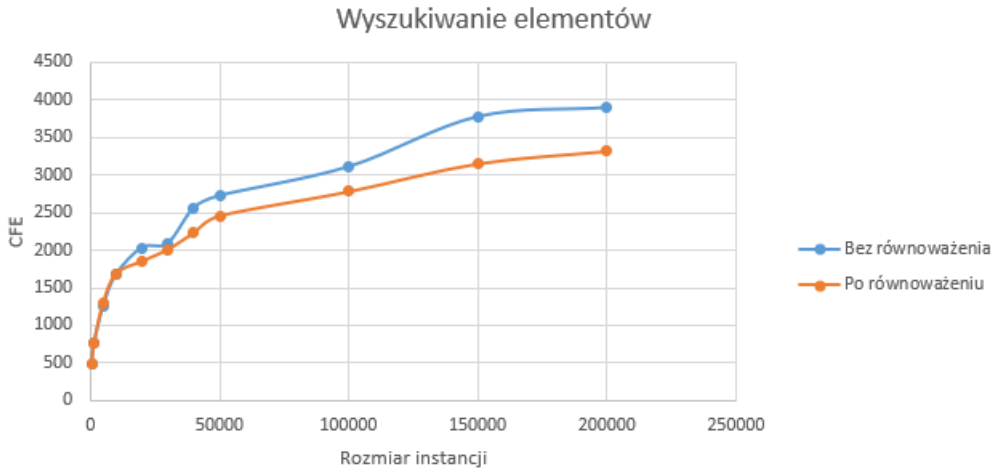
Test 2. Usuwanie elemtnów z drzewa - CPU TIME



Ten wykres pokazuje że czas usuwania elementów z optymalizowanej struktury jest mniejszy niż ze struktury bez optymalizacji. Wytłumaczyć można w ten sposób że, złożoność czasowa usuwania elementu z nieoptymalizowanego drzewa jest w najgorszym przypadku O(n) zaś dla struktury optymalizowanej O(log(n)).

Wniosek: Algorytm usuwania dla optymalizowanej struktury daje zysk o **14%** mniej czasu procesora jest potrzebne aby usunąć element z optyamlizowanej struktury

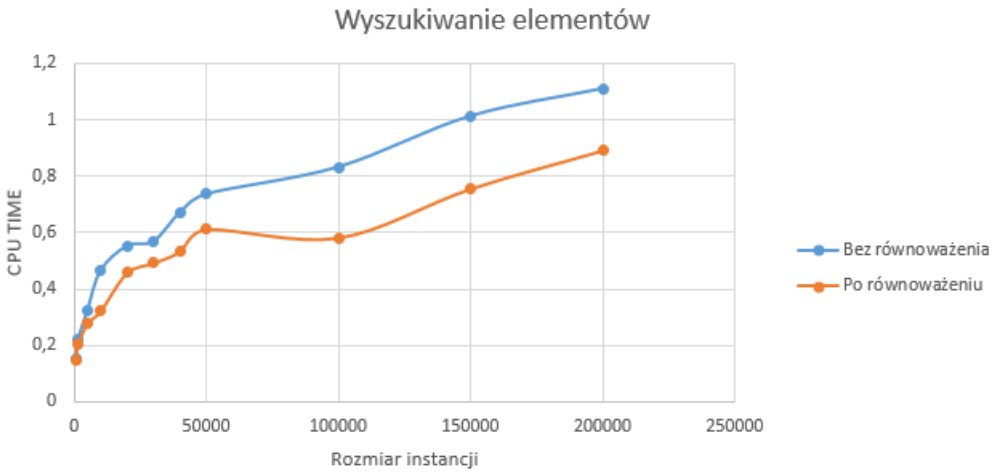
Test 3. Wyszukiwanie elementów w drzewie - CFE



Ten wykres pokazuje że liczba cykli procesora potrzebna do wyszukiwania elementu w optymalizowanej strukturze jest mniejsza niż w strukturze bez optymalizacji. Powodem tego jest: żeby znaleźć element w drzewie nierzrównoważonym musimy w najgorszym przypadku sprawdzić n węzłów dla zrównoważonego musimy sprawdzić log(n) węzłów.

Wniosek: Algorytm wyszukiwania w optymalizowanej strukturze daje zysk o **10%** mniej cykli procesora jest potrzebne aby znaleźć element w optyamlizowanej strukturze

Test 3. Wyszukiwanie elementów w drzewie - CPU TIME



Ten wykres pokazuje że czas pracy procesora potrzebny do wyszukiwania elementu w optymalizowanej strukturze jest mniejszy niż w strukturze bez optymalizacji. Wynika to ze złożoności czasowej: dla wyszukiwania w drzewie BST bez równoważenia jest ona O(n), w przypadku drzewa zrównoważonego O(log(n)).

Wniosek: Algorytm wyszukiwania w optymalizowanej strukturze daje zysk o **19%** mniej czasu procesora jest potrzebne aby znaleźć element w optyamlizowanej strukturze

Literatura

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Wprowadzenie do algorytmów*, 2007

[2] Jon Bentley: *Pisanie skutecznych programów*, ISBN 0-13-970251-2

[3] Krzysztof Lichota: *Optymalizacja programów OpenSource*

[4] Konrad Kokoszkiewicz: *Optymalizacja wydajności programów w języku C*

[5] Kurt Guntheroth: *C++ Optymalizacja kodu*, 2016