STRUKTURY DANYCH I ZŁOŻONOŚĆ OBLICZENIOWA

PROJEKT

Streszczenie

Badanie efektywności operacji dodawania, usuwania oraz wyszukiwania elementów w różnych strukturach danych.

Paweł Szynal

226026@student.pwr.edu.pl

Zadanie projektowe nr 1

Temat projektu: Badanie efektywności operacji dodawania, usuwania oraz wyszukiwania elementów w różnych strukturach danych.			
Data wykonania ćwiczenia	17.03.2017	**Termin zajęć**	Czwartek godz. 18^{55}-20^{35}
Data oddania sprawozdania	30.03.2017	**Wersja**	1

Skład grupy	**Prowadzący**	**Ocena**
Paweł Szynal, 226026	Dr inż. Dariusz Banasiak	

1 Cel projektu:

Implementacja następujących struktur danych:

- Tablica
- Lista
- Kopiec binarny
- Drzewo czerwono-czarne

Pomiary czasu działania operacji :

- Dodawanie elementu
- Usuwanie elementu
- Wyszukiwanie elementu

2.Przyjęte założenia:

- Podstawowym elementem struktur danych jest 4 bajtowa liczba całkowita ze znakiem
- Wszystkie struktury danych są alokowane dynamicznie
- Ponieważ pojedynczy pomiar czasu jest obarczony dużym błędem pomiaru pomiary wykonane są wielokrotnie (100 razy)
- Językiem programowania w jakim napisany jest program jest C++ (język kompilowany dla kodu natywnego a nie dla interpretowany/uruchamiany na maszynach wirtualnych)
- Visual Studio 2015 Community jest środowiskiem programistycznym (IDE) w jakim pisany jest program.

Spis treści

Wstęp:

Struktura danych to sposób uporządkowania informacji np. w pamięci komputera. W tym sprawozdaniu rozważać będziemy cztery takie struktury danych: tablicę dynamiczna, listę , kopiec oraz drzewo czerwono czarne. Na strukturach danych wykonywane są operacje takie jak dodawanie elementu w dowolnym miejscu struktury, usuwanie dowolnego elementu struktury, wyszukiwanie elementu o zadanej wartości. Operacje te są wykonywane na podstawie zaprogramowanych algorytmów.

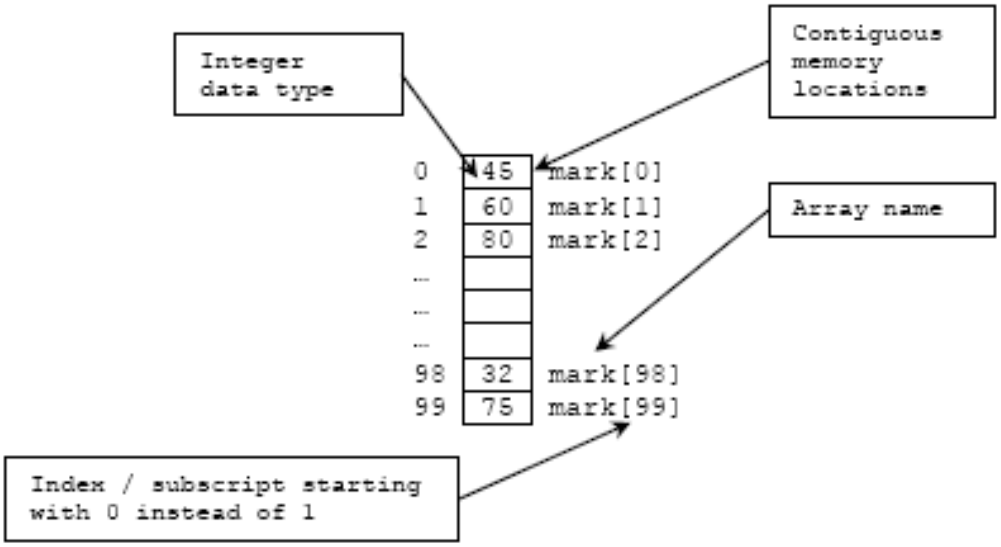
Złożoność obliczeniowa to nauka zajmująca się badaniem i obliczaniem czasowej i pamięciowej złożoności obliczeniowej w zależności od wielkości wejścia – n (ilości danych wejściowych).

Różne struktury mają różną złożoność obliczeniową(czasową i pamięciową), różnych operacji, dlatego programista wybierając strukturę do przechowywania danych kieruje się najczęściej złożonością obliczeniową operacji(algorytmu), która będzie najczęściej wykonywana i potrzebna, dla danej wielkości struktury danych. Czasem w zależności od dostępnego sprzętu i oprogramowania programista musi wybrać algorytm o jak najniższej pamięciowej złożoności obliczeniowej, kosztem znacznego zwiększenia czasowej złożoności obliczeniowej.

Tablica (relokowana dynamicznie)

Tablica jest strukturą dynamiczna, która realokujemy co każde dodanie/ usunięcie elementu, tak, żeby zajmowała jak najmniej miejsca(niska pamięciowa złożoność obliczeniowa). Elementy tej struktury znajdują się w pamięci obok siebie, więc aby dostać się do elementu o danych indeksie wystarczy podać jego wartość i doliczyć tyle do adresu początku tabeli, można to zrobić za pomocą wskaźnika, lub operatora tablicowego „[int]". Jest to duża zaleta tablicy, ponieważ szybko jesteśmy w stanie odczytywać wartości elementów z jej środka(niska czasowa złożoność obliczeniowa). Wadą tablicy jest duża czasowa złożoność obliczeniowa dodawania i usuwania elementu, jest to oczywiście koszt każdorazowej realokacji tablicy.

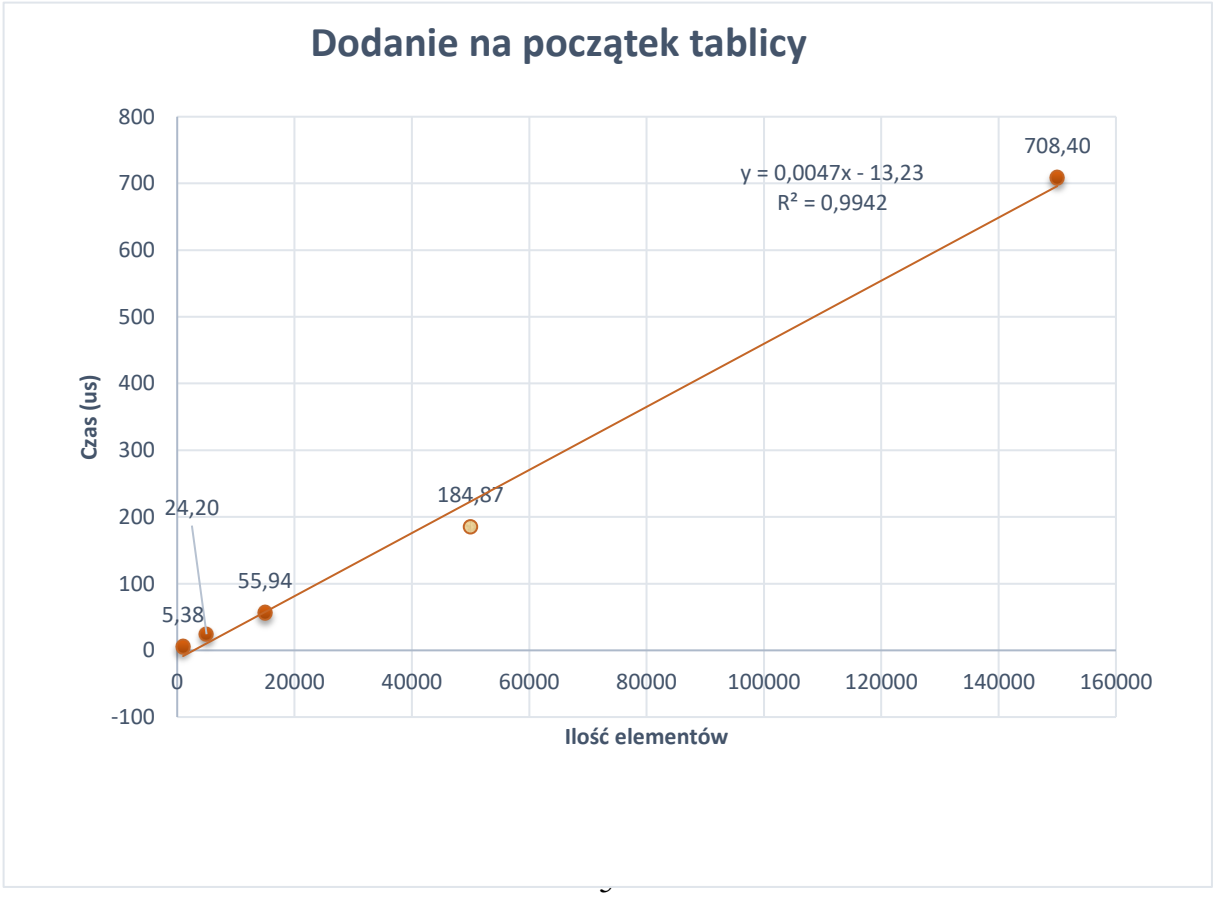
Przykład graficzny tablicy „



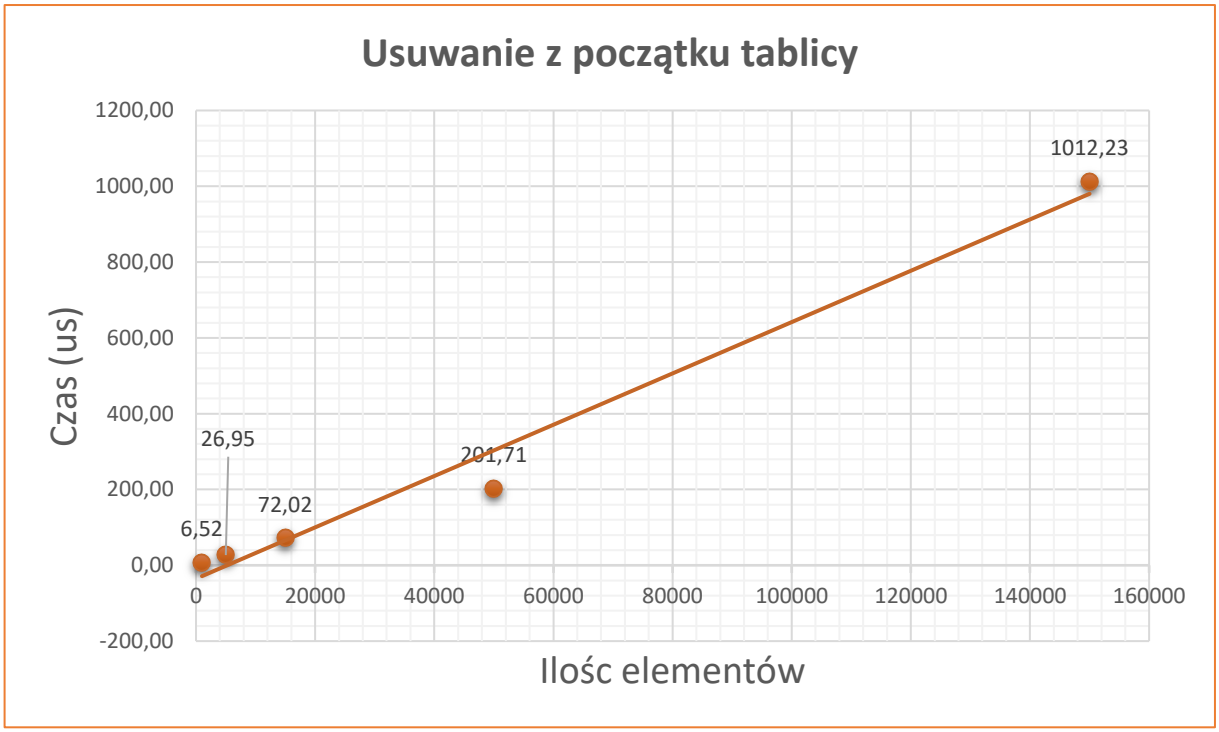
Dodawanie i usuwanie elementu z początku tablicy

Operacje te polegają na przesuwaniu całości zawartości tablicy o 1 miejsce (w stronę 0 dla usuwania i w przeciwnym kierunku dla dodawania elementu). W naszym wypadku algorytm po prostu przepisuje zawartość tablicy w 1 pętli do nowo alokowanej tablicy(bez usuwanego elementu przy odejmowaniu lub z dodatkowym elementem przy dodawaniu). Na końcu usuwamy starą tablicę, a do jej wskaźnika przypisujemy adres nowej tablicy. Czasowa złożoność obliczeniowa tego algorytmu to O(n) – liniowa zależna od ilości elementów w tablicy.

Dodanie na początek tablicy		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	5.38
2	5000	24.20
3	15000	55.94
4	50000	184.87
5	150000	708.40



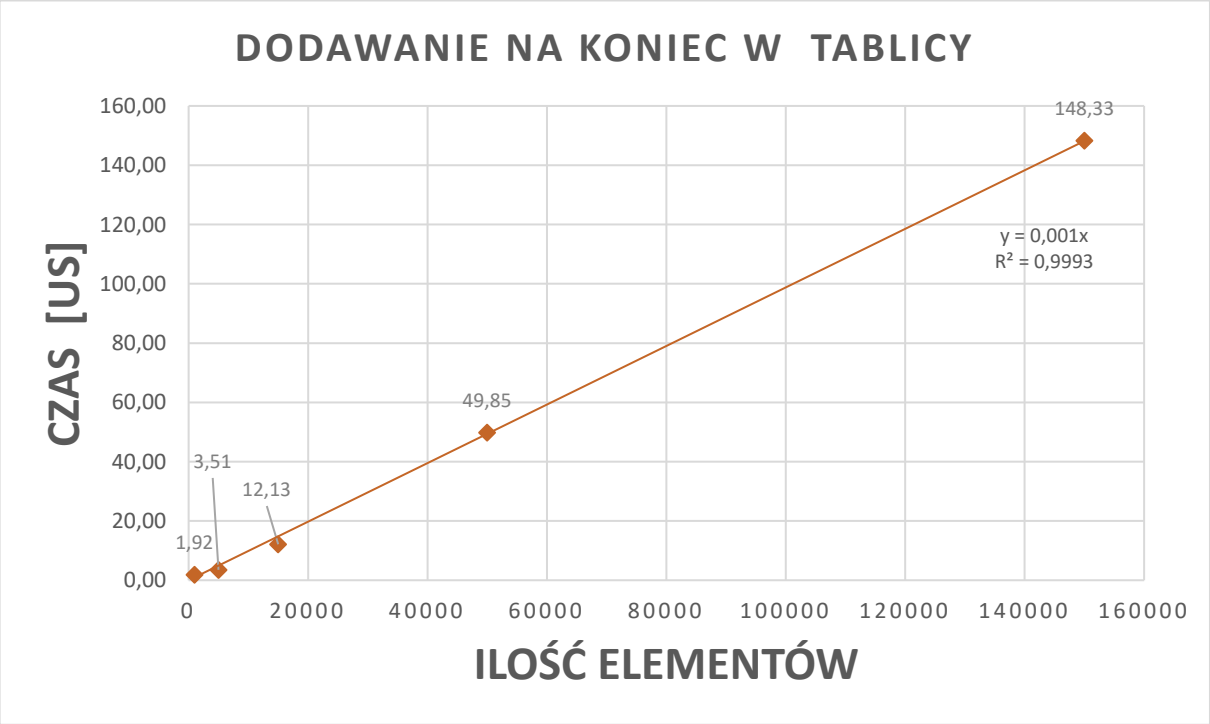
Usuwanie z początku tablicy		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	6.52
2	5000	26.95
3	15000	72.02
4	50000	201.71
5	150000	1012.23



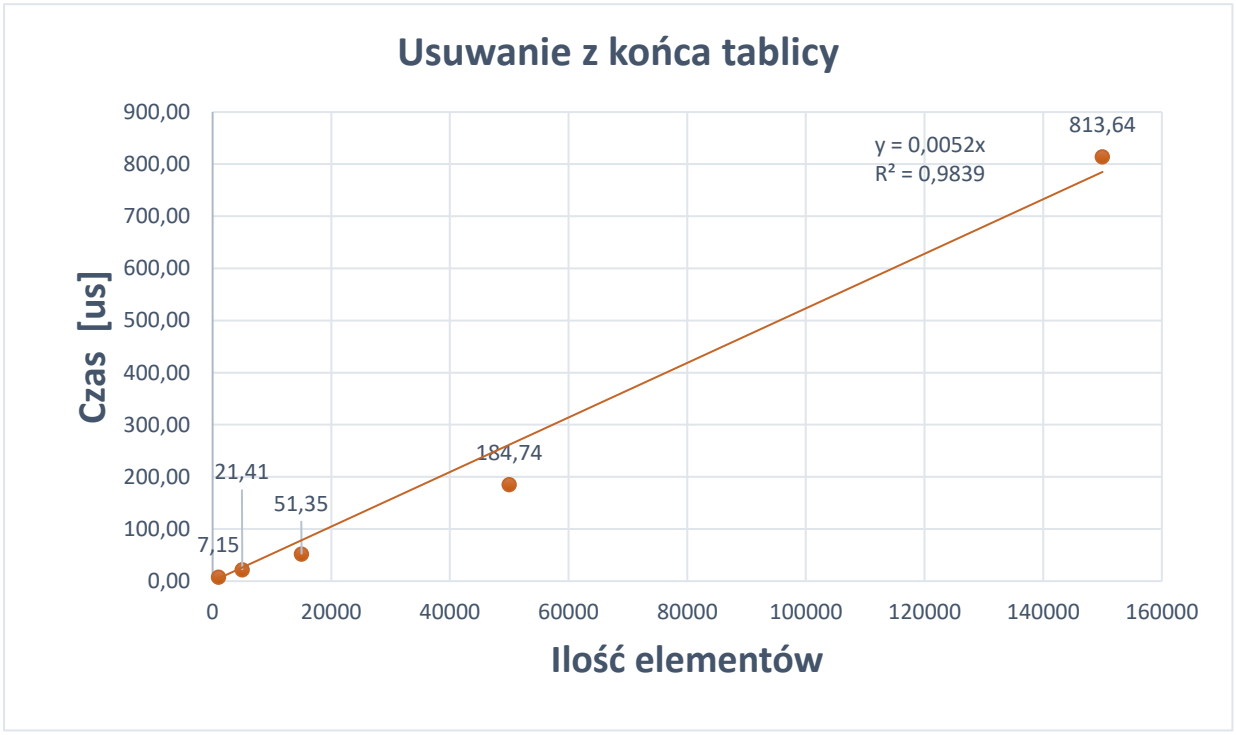
Dodawanie i usuwanie elementu z końca tablicy

Tutaj algorytm ogranicza się do przepisania starej tabeli do nowej i odpowiednio dodaniu elementu na koniec lub usunięciu ostatniego elementu(zmiana rozmiaru). Na końcu usuwamy starą tablicę, a do jej wskaźnika przypisujemy adres nowej tablicy. Czasowa złożoność obliczeniowa tego algorytmu to O(n) – liniowa zależna od ilości elementów w tablicy.

Dodawanie na koniec w tablicy		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	1.92
2	5000	3.51
3	15000	12.13
4	50000	49.85
5	150000	148.33



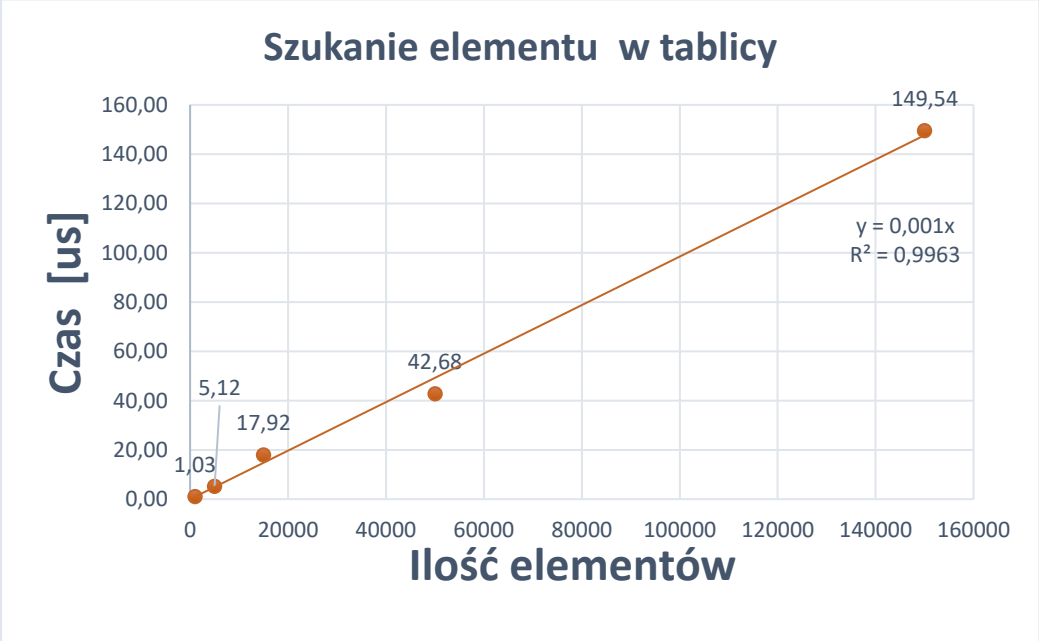
Usuwanie z końca tablicy		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	7.15
2	5000	21.41
3	15000	51.35
4	50000	184.74
5	150000	813.64



Szukanie elementu o zadanym(losowym) kluczu w tablicy

Algorytm w tym punkcie polega na przeszukiwaniu tabeli od początkowego indeksu i sprawdzaniu czy wartość pod tym indeksem zgodna jest z zadanym kluczem. Jeśli algorytm znajdzie taki element to zwraca wartość true – prawda - 1, natomiast jeśli nie znajdzie takiego elementu(przejdzie całą tablic) to zwraca wartość false - fałsz - 0. Czasowa złożoność tego algorytmu jest linowa O(m), gdyż zależy od ilości danych wejściowych.

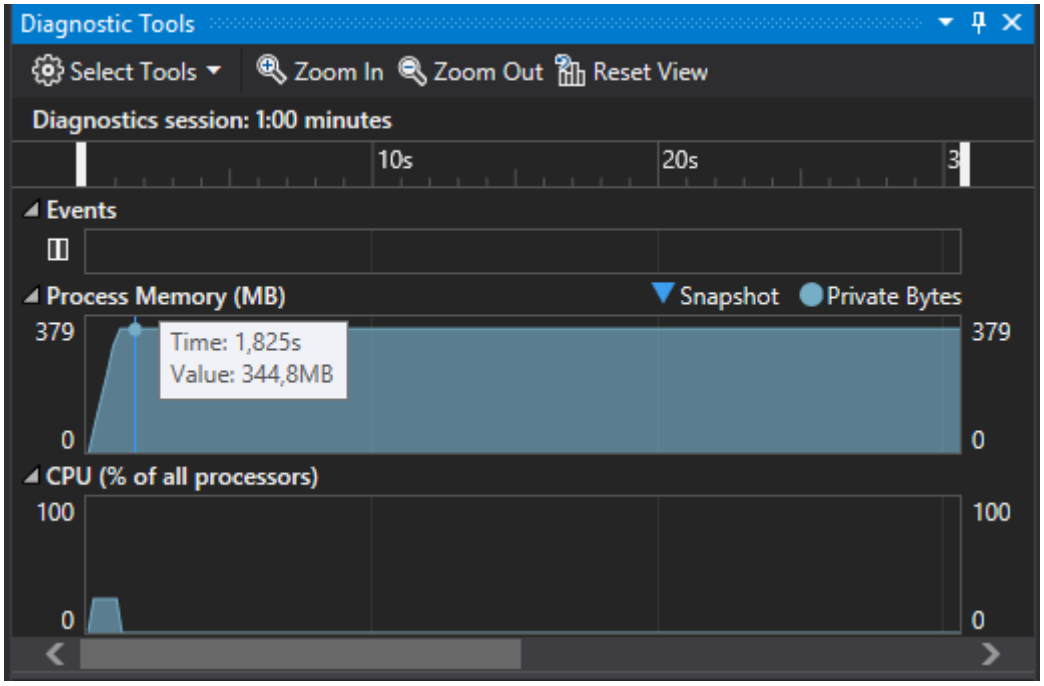
Szukanie elementu w tablicy		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	1.03
2	5000	5.12
3	15000	17.92
4	50000	42.68
5	150000	149.54



Wnioski

Dodawanie i usuwanie elementów z tablicy dynamicznej przebiegło do końca w sposób jaki był spodziewany. Wykres jest linowy jednak czas zmierzony dla struktury 150 tysięcznej jest znacznie wyższy niż wynikałoby to z wcześniejszych pomiarów. Można to zaobserwować porównując np. poniższy wykres z wykresem ” Tablica dodawanie na początek”. Widać, że współczynnik kierunkowy linii trendu na poniższym wykresie jest prawie 5 razy większy. Powodem takiej sytuacji może być fakt, że dla małej ilości danych (do 100k) procesor działa bardzo wydajnie i dopiero dla 150 000 liczb zaczyna odczuwać obciążenie.

Dla 150 000 elementów



Dynamiczna alokacja pamięci w przypadku struktur tablicowych jest istotna w prawidłowym działaniu programu. Nie korzystanie z niej skutkuje niestabilnością programu, który po wykonaniu wielu operacji może zakończyć swoje działanie w wyniku błędu pamięci.
Jednak korzystanie z relokacji po każdym usunięciu/dodaniu elementu wydłuża czas operacji, a dodatkowo uniemożliwia zaobserwowanie rzeczywistej efektywności danego algorytmu.

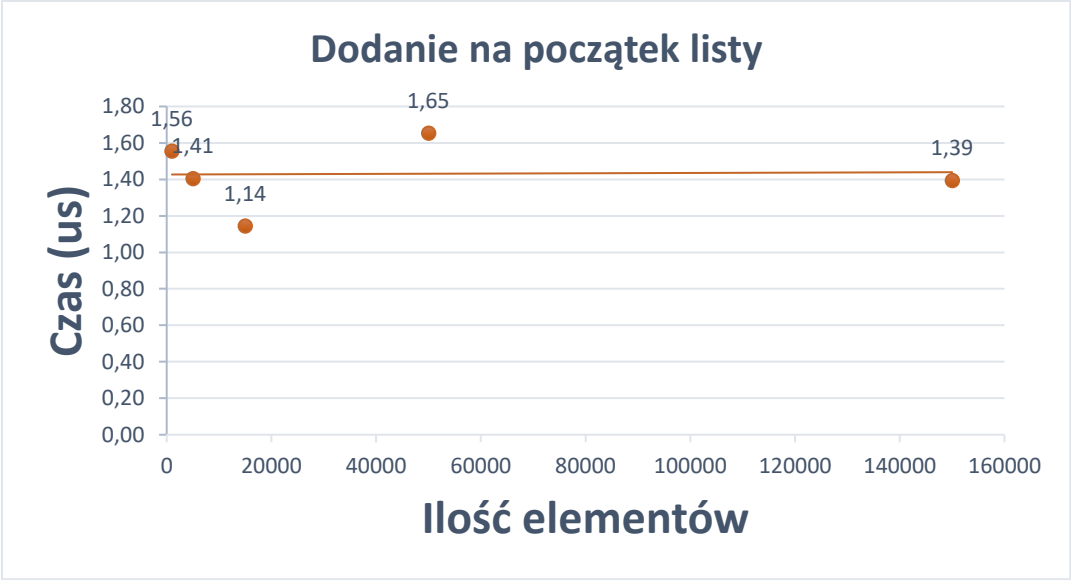
Lista dwukierunkowa

Lista dwukierunkowa to struktura danych, która przechowuje w sobie wskaźnik na swój pierwszy element (head-głowa) i ostatni (tail-ogon). Każdy element listy dwukierunkowej ma w sobie wartość i 2 wskaźniki na poprzedni i na następny element listy. Jeśli element jest głową to wskaźnik na poprzedni element wynosi NULL, natomiast jeśli jest ogonem to wskaźnik na następny element wynosi NULL. Elementy listy nie leżą obok siebie w pamięci komputera. Każdy następny element znajdujemy za pomocą wskaźnika z poprzedniego, lub następnego. Dane tak ułożone w pamięci zajmują więcej miejsca(zmienne wskaźnikowe), co przekłada się na zwiększenie pamięciowej złożoności obliczeniowej, jednak jest zbawienne w skutkach dla czasowej złożoności obliczeniowej przy dodawaniu i usuwaniu elementu do listy. Wadą listy dwukierunkowej jest nie natychmiastowe znajdywanie elementów listy o podanym indeksie. Należy skakać od pierwszego lub ostatniego elementu do elementu o podanym indeksie za pomocą wskaźników na następny lub poprzedni element.

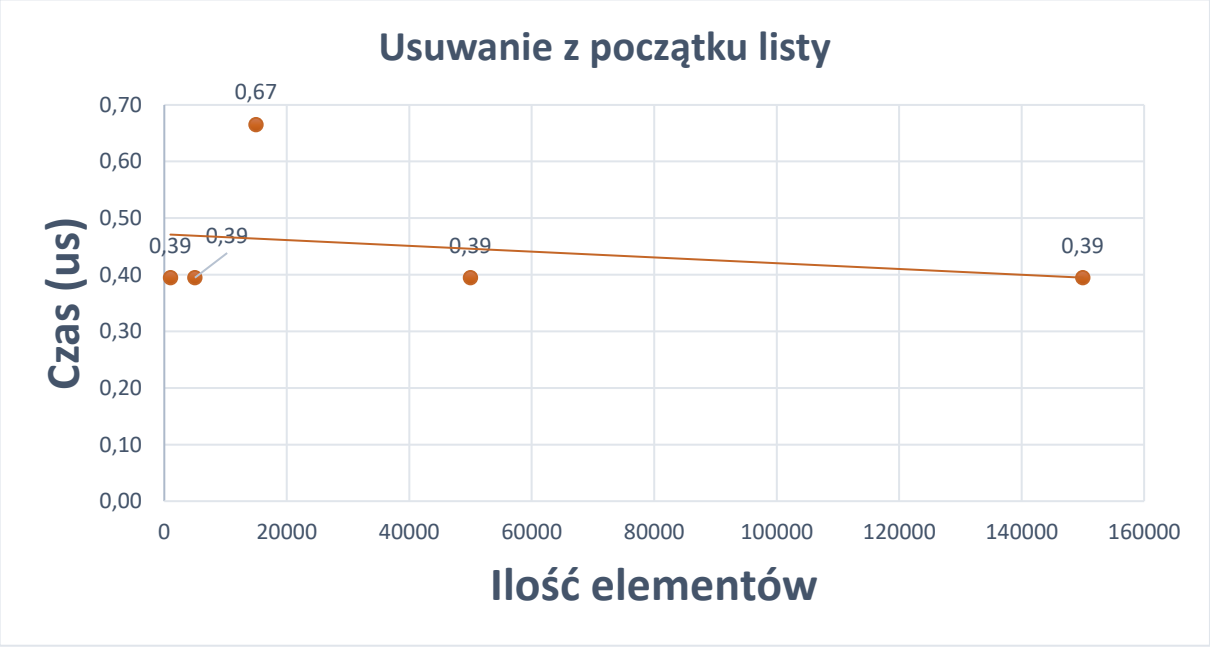
Dodawanie i usuwanie elementu z początku listy

Aby dodać element na początek listy należy stworzyć obiekt lub strukturę 1 elementu listy i ustawić jej wskaźnik elementu poprzedzającego na NULL, a elementu następnego na ten, które aktualnie jest głową. Jeśli głowa istnieje to należy jej poprzednik ustawić na ten element, który stworzyliśmy i na końcu ustawić nowy wskaźnik głowy – head. Natomiast, żeby usunąć pierwszy element listy należy ustawić poprzednik drugiego elementu (o ile istnieje) na NULL. Usunąć głowę, a do wskaźnika głowy przypisać element drugi. Czasowa złożoność tego algorytmu jest stała O(1), gdyż nie zależy od ilości danych wejściowych.

Dodanie na początek listy		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	1.56
2	5000	1.41
3	15000	1.14
4	50000	1.65
5	150000	1.39



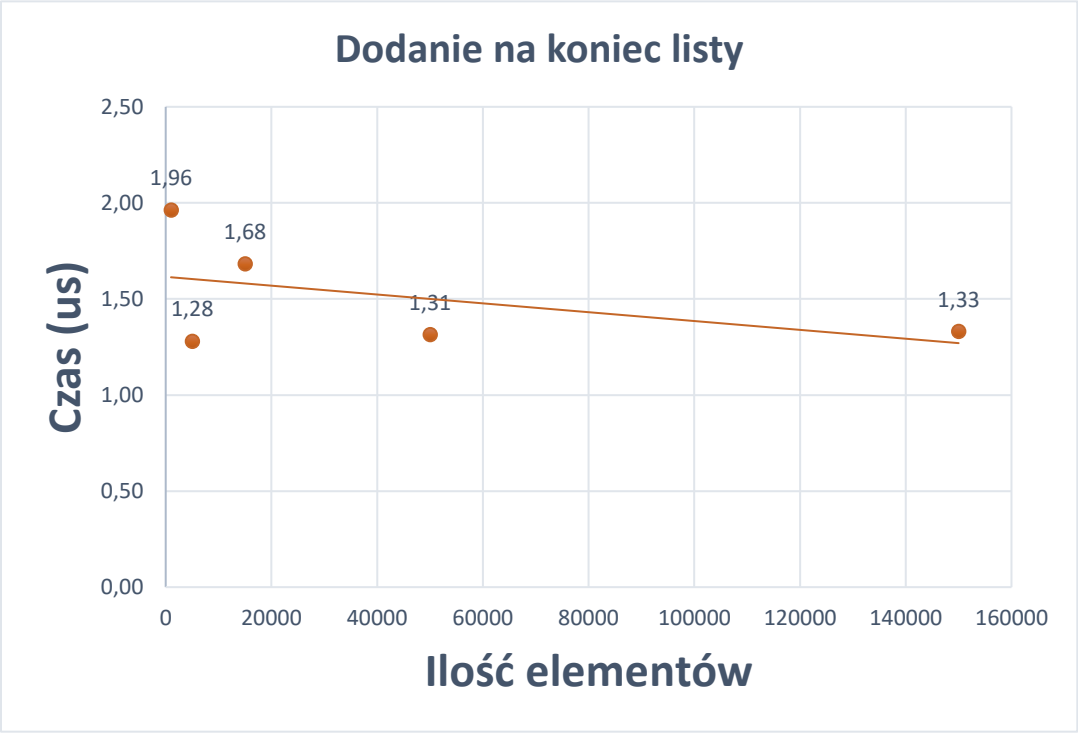
Usuwanie z początku listy		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	0.39
2	5000	0.39
3	15000	0.67
4	50000	0.39
5	150000	0.39



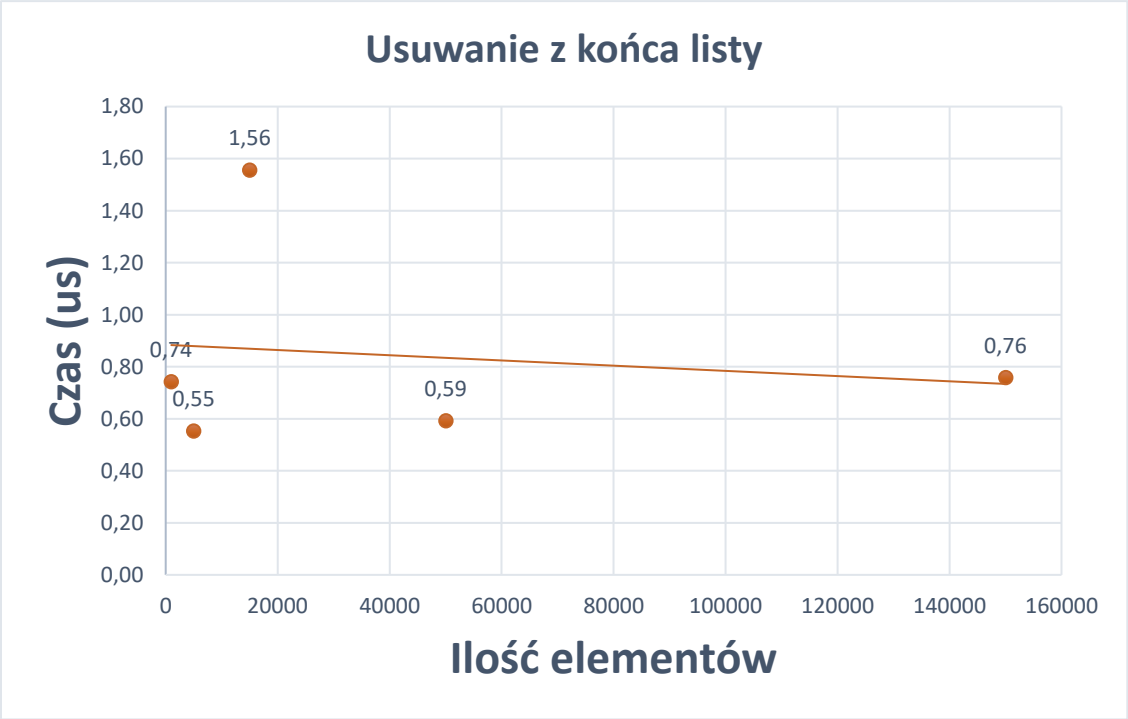
Dodawanie i usuwanie elementu z końca listy

Aby dodać element na koniec listy należy, również stworzyć obiekt lub strukturę 1 elementu listy i ustawić jej wskaźnik elementu następnego na NULL, a elementu poprzedniego na ten, które aktualnie jest ogonem. Jeśli ogon istnieje to należy jego następnik ustawić na ten element, który stworzyliśmy i na końcu ustawić nowy wskaźnik ogona – tail. Natomiast, żeby usunąć ostatni element listy należy ustawić następnik przedostatniego elementu (o ile istnieje) na NULL. Usunąć ogon, a do wskaźnika ogona przypisać element przedostatni. Czasowa złożoność tego algorytmu jest stała O(1), gdyż nie zależy od ilości danych wejściowych.

Dodanie na koniec listy		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	1.96
2	5000	1.28
3	15000	1.68
4	50000	1.31
5	150000	1.33



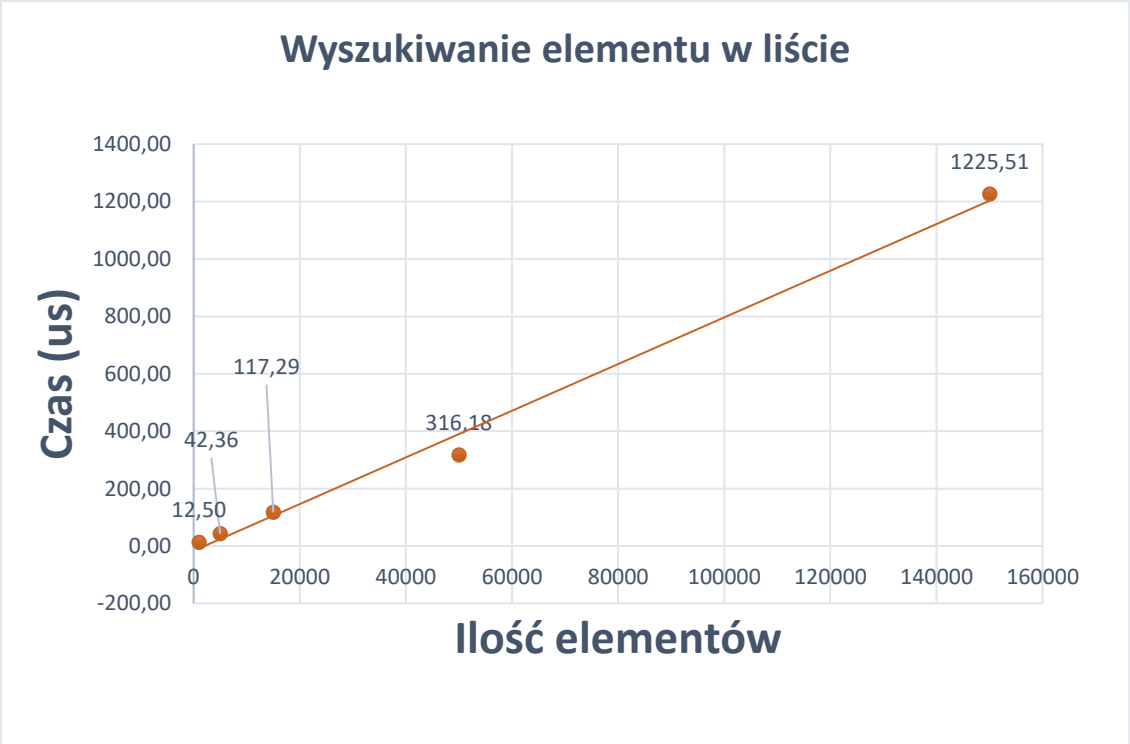
Usuwanie z końca listy		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	0.74
2	5000	0.55
3	15000	1.56
4	50000	0.59
5	150000	0.76



Szukanie elementu o losowym kluczu

Szukanie elementu o zadanym kluczu ogranicza się tylko do przeszukiwania listy(skakania po wskaźnikach) i sprawdzania czy wartość danego elementu równa jest kluczowi. Jeśli tak funkcja zwraca true, jeśli nie znaleziona żadnego elementu spełniającego równanie i sprawdzono całą listę zwraca false. Czasowa złożoność tego algorytmu jest linowa O(m), gdyż zależy od ilości danych wejściowych.

Wyszukiwanie elementu w liście		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	12.50
2	5000	42.36
3	15000	117.29
4	50000	316.18
5	150000	1225.51



Wnioski

Wykresy dla dodawania i usuwania liczb do listy z końca i początku są zbliżone do stałych. Pojedyncze pomiary zaburzają statystykę, i dlatego linia trendu jest liniowa. Błąd ten spowodowany jest niedokładnością funkcji mierzącej czas, jak i różną chwilową hierarchizacją wątków w procesorze.

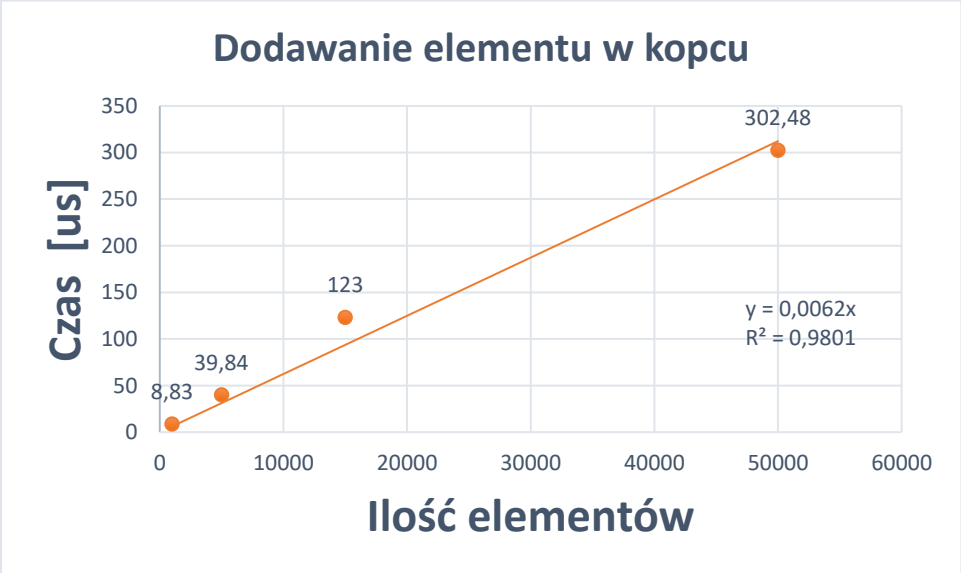
Kopiec

Kopiec binarny to struktura danych oparta na drzewie, w której wartości są uporządkowane niemalejąca lub nierosnąca w stosunku to „starszych" elementów kopca. Każdy element kopca z wyjątkiem korzenia i liści, przechowuje w sobie wartość, oraz wskaźnik na rodzica i synów(lewego i prawego). Wynika z tego, że z każdego elementu z wyjątkiem korzenia, można jednoznacznie przejść poziom wyżej i z każdego elementu z wyjątkiem liści można przejść na dwa sposoby poziom niżej. Wyjątkiem jest też ostatni rodzic, który może mieć jednego potomka. Do zaimplementowania kopca wystarczy w klasie umieścić pole wskaźnika na korzeń, a w nim wskaźniki na 2 kolejne elementy kopca(synów) i w nich kolejne wskaźniki itd. Jeśli wskaźnik na korzeń równy jest NULL to kopiec jest pusty. W naszym algorytmie wykorzystuję własności tablicy. Nie będziemy dynamicznie przydzielać pamięci kopca co każde dodanie wartości do niego, ponieważ zwiększyło by to znacznie złożoność obliczeniową i upodobniło go tym samym do tablicy. W kopcu zaimplementowanym za pomocą wskaźników tworzy się tylko nowe elementy kopca i ustawia odpowiednio wskaźniki, dlatego złożoność obliczeniowa operacji na kopcu jest znacznie mniejsza. Ponadto nowoczesne komputery posiadają tak dużo pamięci, że deklarowanie tablicy na wyrost na większą ilość elementów jest niezauważalne dla komputera. Ponadto w naszym kopcu implementowanym tablicowo umieścimy metody, które będą zwracać indeks w tablicy jego rodzica i synów. Zaletą kopca jest niska czasowa złożoność obliczeniowa dodawania elementu i usuwania elementu. Jest tak ponieważ zmiana jednego elementu w kopcu posiadającym 200k elementów maksymalnie implikuje tylko $\log_2 200000 \sim 18$ operacji porównania i ewentualnej zamiany elementów tak, aby był spełniony warunek kopca(synowie są nie więksi lub niemniejsi od rodzica).

Dodawanie elementu w kopcu

Aby dodać element do kopca należy dodać element do na koniec kopca i następnie sprawdzić czy spełniony jest warunek kopca(w naszym eksperymencie: czy jest mniejszy lub równy rodzicowi). Jeśli warunek kopca nie jest spełniony należy zamienić ze sobą nasz element(syna) z jego rodzicem. Następnie należy sprawdzić warunek poziom wyżej i wykonywać zamiany dopóki jest to konieczne. Kopiec jest uporządkowany, gdy dotarliśmy do korzenia, lub gdy jedno z kolejnych porównań w algorytmie zwróci nam wynik, że zachowany jest warunek kopca. Złożoność obliczeniowa takiej operacji to $O(\log_2 n)$ gdyż zależy od ilości poziomów kopca(drzewa binarnego).

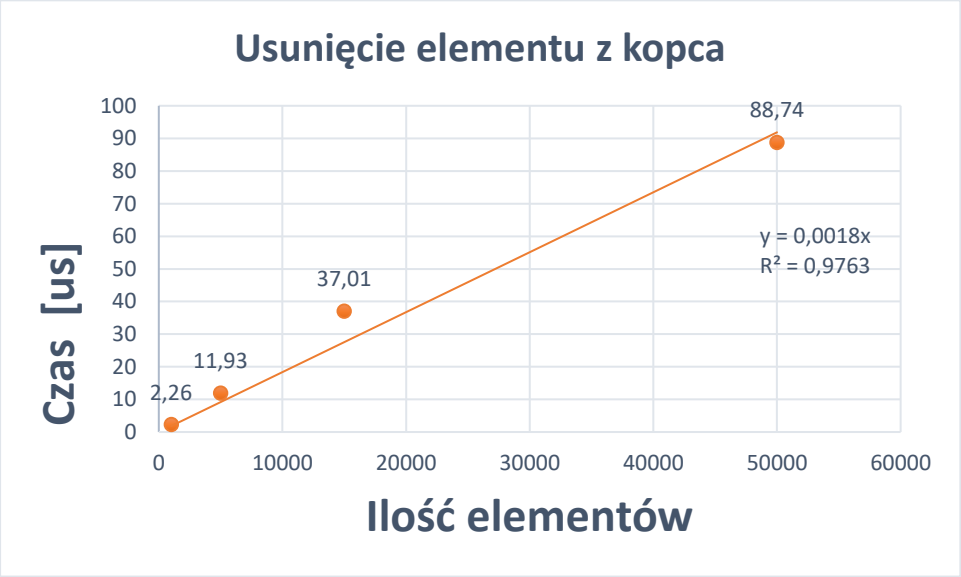
Dodawanie elementu w kopcu		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	8.83
2	5000	39.84
3	15000	123.00
4	50000	302.48
5		



Usuwanie elementu kopca

Aby usunąć korzeń należy przenieść ostatni element kopca(o ile istnieje) w miejsce korzenia, a stary korzeń usunąć. Następnie trzeba sprawdzić warunek kopca(czy korzeń jest większy od obu synów. Jeśli warunek kopca nie jest spełniony należy zamienić korzeń z synem o większej wartości, a następnie powtarzać sprawdzanie i ewentualne zamiany, aż kopiec będzie uporządkowany. Kopiec jest uporządkowany, gdy dojdziemy do liścia, lub gdy jedno z kolejnych porównań zwróci nam wynik, że warunek kopca jest spełniony(rodzic niemniejszy od synów). Złożoność obliczeniowa takiej operacji to $O(\log_2 n)$ gdyż zależy od ilości poziomów kopca(drzewa binarnego – podstawa 2).

Usunięcie elementu z kopca		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	2.26
2	5000	11.93
3	15000	37.01
4	50000	88.74



Wnioski

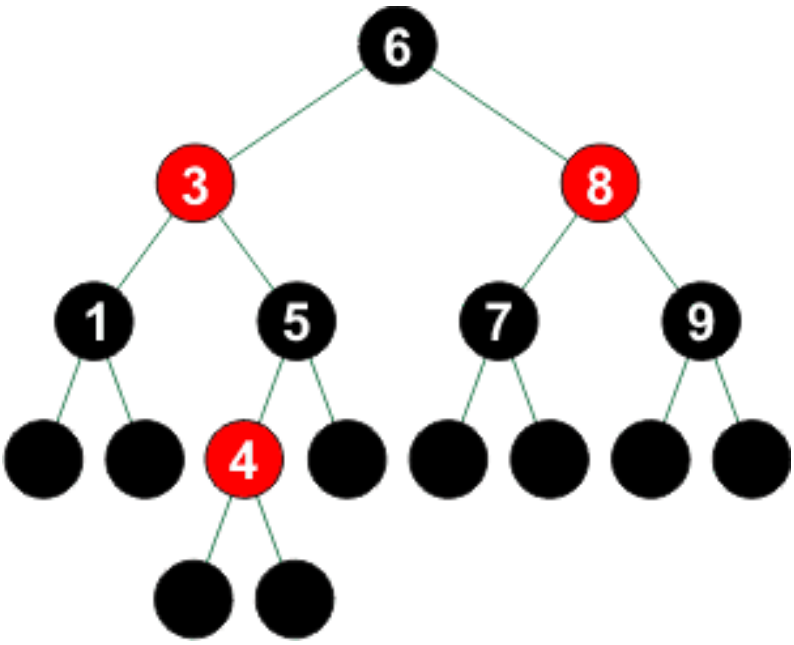
Kopiec jest znacznie szybszą strukturą danych od poprzednich struktur. Na dodawanie elementu i usuwanie korzenia składa się alokacja nowego elementu lub usunięcie elementu, co zajmuje tyle samo czasu co w liście, oraz uporządkowanie kopca po każdej operacji. Jednak złożoność tej operacji jest logarytmiczna i wraz z wielkością struktury rośnie coraz wolniej.

Drzewo Czerwono-Czarne

Drzewo czerwono-czarne (ang. red-black tree) jest odmianą samoorganizującego się binarnego drzewa poszukiwań. Drzewo czerwono-czarne jest skomplikowane w implementacji, lecz charakteryzuje się niską złożonością obliczeniową elementarnych operacji takich, jak wstawianie, wyszukiwanie czy usuwanie elementów z drzewa. Każde drzewo czerwono-czarne spełnia poniższe kryteria :

- Każdy węzeł drzewa jest albo czerwony, albo czarny.
- Każdy liść drzewa (węzeł pusty null) jest zawsze czarny
- Korzeń drzewa jest zawsze czarny
- Jeśli węzeł jest czerwony, to obaj jego synowie są czarni – innymi słowy, w drzewie nie mogą występować dwa kolejne czerwone węzły, ojciec i syn
- Każda prosta ścieżka od danego węzła do dowolnego z jego liści potomnych zawiera tę samą liczbę węzłów czarnych.

Przykład reprezentacji graficznej drzewa czerwono-czarnego:

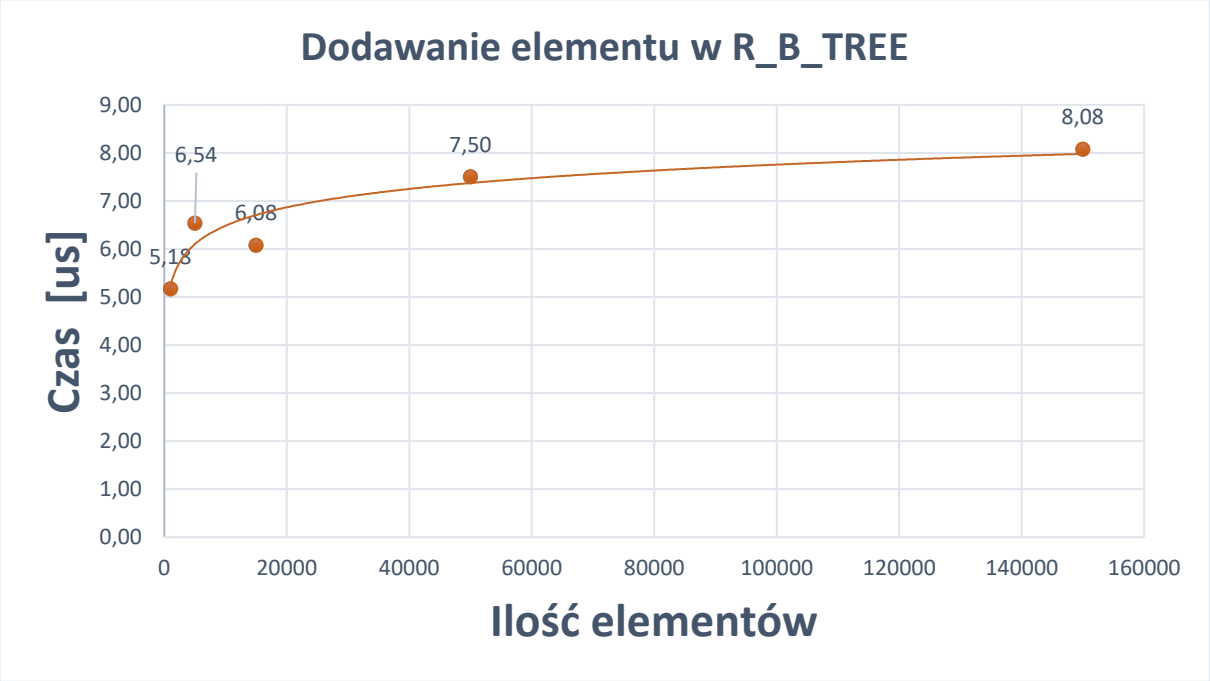


Dodawanie elementu do drzewa czerwono-czarnego

Tworzymy nowy węzeł, po czym wstawiamy go jak zwykły węzeł drzewa binarnego. Następnie sprawdzamy czy warunki drzewa czerwono-czarnego nie zostały zaburzone. W przypadku niezgodności dokonujemy napraw.

Dodawanie elementu w R_B_TREE		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	5.18
2	5000	6.54
3	15000	6.08
4	50000	7.50
5	150000	8.08

Teoretyczna złożoność obliczeniowa: O(logn)

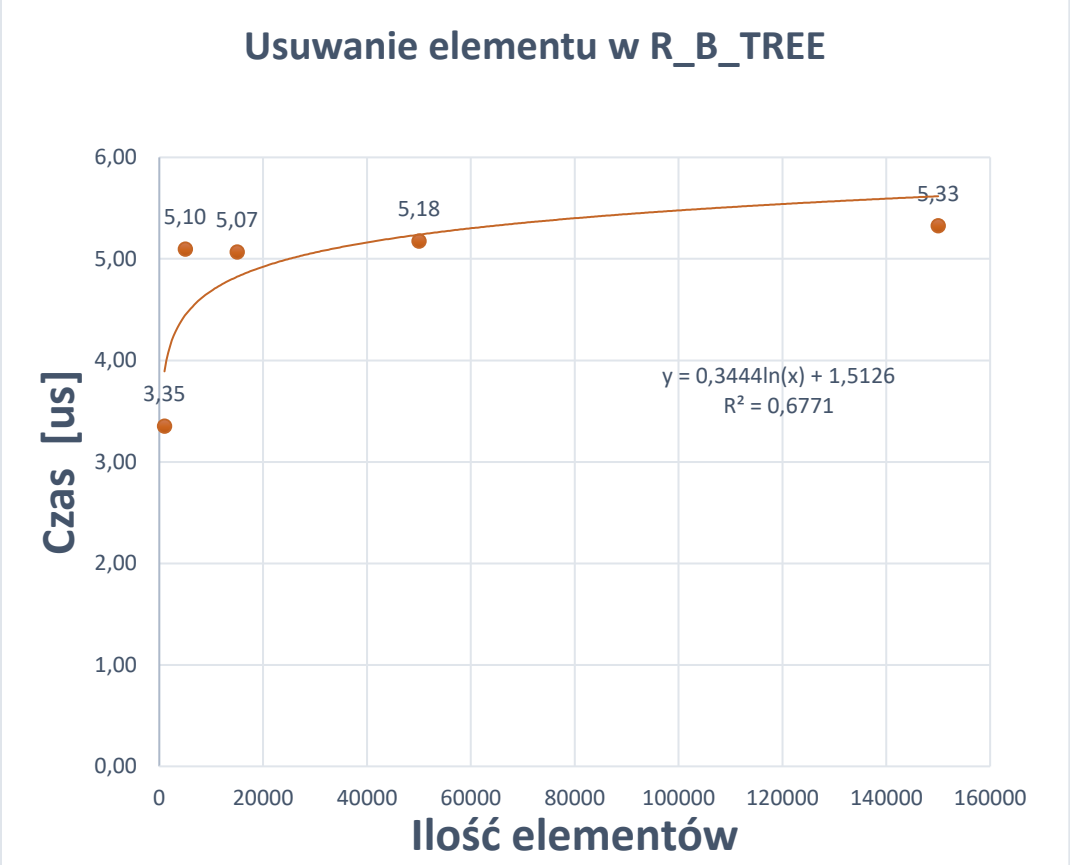


Usunięcie elementu z drzewa czerwono-czarnego o podanej wartości

Wyszukujemy węzeł po czym go usuwamy, jak w drzewie binarnym. Następnie sprawdzamy czy warunki drzewa czerowno-czarnego nie zostały zaburzone. W przypadku niezgodności dokonujemy napraw.

Usuwanie elementu w R_B_TREE		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	3.35
2	5000	5.10
3	15000	5.07
4	50000	5.18
5	150000	5.33

Teoretyczna złożoność obliczeniowa : O(logn)

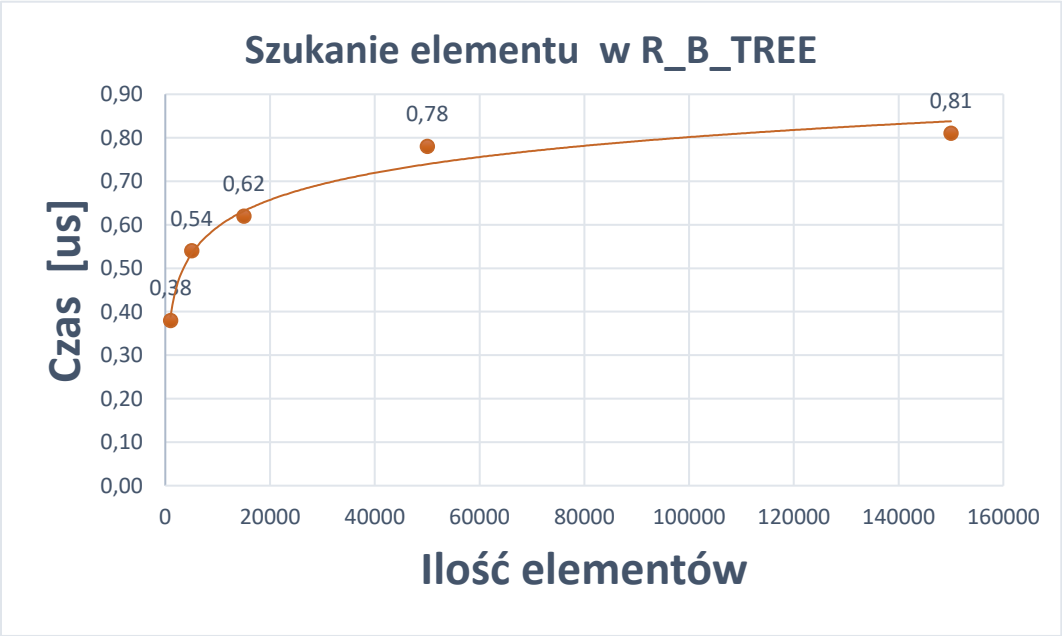


Wyszukanie elementu w drzewie czerwono-czarny o podanej wartości.

Przechodzimy po drzewie w celu sprawdzenia czy występuje tam żądana wartości.

Szukanie elementu w R_B_TREE		
L.p	ilość elementów	Uśredniony czas 100 pomiarów [us]
1	1000	0.38
2	5000	0.54
3	15000	0.62
4	50000	0.78
5	150000	0.81

Teoretyczna złożoność obliczeniowa : O(logn)



Wnioski

Największą zaletą drzew czerwono - czarnych jest zrównoważona oraz uporządkowana struktura, pozwalająca na szybką modyfikację elementów i efektywną implementację algorytmu wyszukiwania.

Literatura:

- Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., Wprowadzenie do Algorytmów, wyd. 4, Warszawa: Wydawnictwa Naukowo-Techniczne

[1] Struktury danych i złożoność obliczeniowa -ćwiczenia – materiały dla studentów
http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/

[2] Obsługa plików:
http://cpp0x.pl/kursy/Kurs-C++/Obsluga-plikow/305

[3] Pomiar czasu:
http://stackoverflow.com/questions/1739259/how-to-use-queryperformancecounter

[4] Badanie efektywności operacji dodawania, usuwania oraz wyszukiwania elementów w różnych strukturach danych

http://antoni.sterna.staff.iiar.pwr.wroc.pl/sdizo/ZP_1.pdf

[5] Implementacja wyświetlania kopca:

http://eduinf.waw.pl/inf/alg/001_search/0113.php