Jakub Pawłowski
235002

09.04.2018

Struktury Danych i Złożoność Obliczeniowa

Zadanie projektowe nr 1

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

1. Opis zadania projektowego

Celem zadania projektowego było zaimplementowanie struktur danych oraz metod wykonujących operacje dodawania, usuwania oraz wyszukiwania elementów.
Zaimplementowane zostały:

Tablica

Lista

Kopiec binarny

W projekcie zostały przyjęte następujące założenia:

- Podstawowym elementem struktur jest 4 bajtowa liczba całkowita ze znakiem integer
- Wszystkie struktury danych są alokowane dynamicznie
- Dla tablicy i listy zostały rozpatrzone operacje dodawania i usuwania elementu na początek, koniec i losowe miejsce w strukturze
- Zostały pomierzone czasy wykonywania operacji w funkcjach dodawania, usuwania i wyszukiwania
- Językiem programowania był język C++
- Nie zostały użyte gotowe biblioteki zawierające struktury i algorytmy
- Kod źródłowy jest komentowany

2. Złożoności obliczeniowe operacji

Złożonością obliczeniową operacji nazywamy zasoby ilość zasobów komputerowych koniecznych do wykonania programu realizującego algorytm. Przedstawiamy ją jako funkcję pewnego parametru, określającego rozmiar rozwiązywanego zadania. Wyróżniamy dwa zasadnicze podziały złożoności: złożoność pamięciową i złożoność czasową.

Złożoność pamięciowa to ilość pamięci wykorzystanej w celu realizacji algorytmy. Wyrażana jest w liczbie bajtów lub liczbie zmiennych typów elementarnych. Złożoność pamięciowa jest zawsze powiązana jako funkcja rozmiaru danych.

Złożoność czasowa jest to czas, w którym realizowane jest działanie algorytmu. Czas określany jest w standardowych jednostkach czasu, liczbie cykli procesora. Złożoność czasowa jest zawsze powiązana jako funkcja rozmiaru danych.

Rozróżniamy trzy przypadki złożoności:

- Przypadek optymistyczny, określa najkrótszy czas wykonania algorytmu dla najkorzystniejszego zbioru danych
- Przypadek średni, określa czas i pamięć dla typowo losowych zbiorów danych
- Przypadek pesymistyczny, określa zużycie zasobów dla najbardziej niekorzystnego zbioru danych

Funkcja	Średnia	Pesymistyczna
Dodanie wartości	O(n)	O(n)
Usunięcie wartości	O(n)	O(n)
Wyszukanie wartości	O(n)	O(n)
Dostęp do elementu	O(-)	O(1)

Złożoności operacji wykonywanych na tablicy

Funkcja	Średnia	Pesymistyczna
Dodanie wartości	O(-)	O(n)
Usunięcie wartości	O(-)	O(n)
Wyszukanie wartości	O(n)	O(n)
Dostęp do elementu	O(n)	O(n)

Złożoność operacji wykonywanych na liście

Funkcja	Średnia	Pesymistyczna
Dodanie wartości	O(1)	O(1)
Usunięcie wartości	O(1)	O(1)
Wyszukanie wartości	O(n)	O(n)
Dostęp do elementu	O(n)	O(n)

Złożoność operacji wykonywanych na kopcu

3. Pomiar czasu

Do mierzenia czasu zostały użyte funkcje QueryPerformanceCounter. Funkcja mierzenia czasu operuje na liczbie taktów procesora. Pozwala to na dokładne uzyskanie wartości czasu jaki potrzebował algorytm na realizacje. Czas uzyskuje się badając różnicę funkcji startTimer() i endTimer(). A wynik zwracany jest milisekundach.

```
LARGE_INTEGER startTimer()
{
    LARGE_INTEGER start;
    DWORD_PTR oldmask = SetThreadAffinityMask(GetCurrentThread(), 0);
    QueryPerformanceCounter(&start);
    SetThreadAffinityMask(GetCurrentThread(), oldmask);
    return start;
}
LARGE_INTEGER endTimer()
{
    LARGE_INTEGER stop;
    DWORD_PTR oldmask = SetThreadAffinityMask(GetCurrentThread(), 0);
    QueryPerformanceCounter(&stop);
    SetThreadAffinityMask(GetCurrentThread(), oldmask);
    return stop;
}
```

Przed użyciem deklarujemy funkcje. A przed użyciem badanego algorytmu rozpoczynamy liczenie czasu funkcją startTimer(). Po algorytmie wywołujemy funkcję endTimer(). Do zmiennej typu double przypisujemy różnicę czasów.

użycie w programie

```
LARGE_INTEGER performanceCountStart,performanceCountEnd;
performanceCountStart = startTimer(); //zapamiętujemy czas początkowy
display(tab,6);  //tutaj funkcje, których mierzymy wydajność
performanceCountEnd = endTimer(); //zapamiętujemy koniec czasu
double tm = performanceCountEnd.QuadPart - performanceCountStart.QuadPart;
cout << endl << "Time:" <<tm <<endl;
```

4. Implementacja tablicy

Tablica jest jedną z najprostszych struktur danych. Jest to zbiór wartości z przypisanymi indeksami. W projekcie używane są tablice alokowane dynamiczne. Dzięki czemu ich wielkość może być minimalna. W ten sposób zaoszczędzając pamięć. W przypadku relokacji pamięci definiowane są nowe tablice, zawartość starych tablic jest przekopiowywana a ich zawartość jest usuwana w celu oszczędności pamięci. Zaletą takiego rozwiązania jest ilość pamięci potrzebnej do przechowywania danych, która jest minimalna. Wadą takiego rozwiązania jest potrzeba zwalniania pamięci, za którą odpowiedzialny jest programista.

Operacje wykonywane na tablicy to:

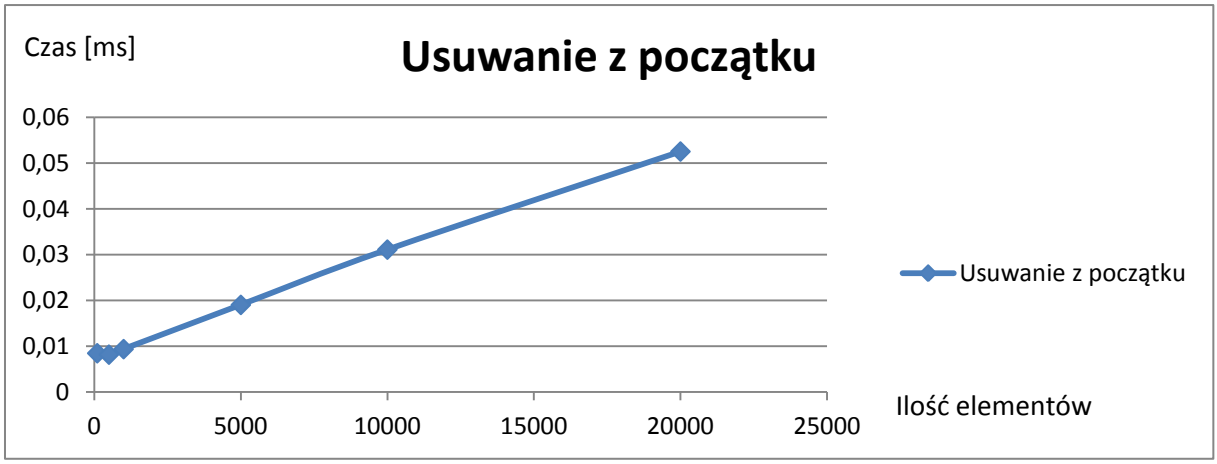
- Zbudowanie tablicy
  Tablica::ZbudujZPliku(); - Otwierany jest dedykowany plik. Pierwszy wiersz pliku określa rozmiar tablicy. Deklarowana jest tablica. Do tablicy przypisywane są wartości z pliku za pomocą pętli for.
- Usunięcie na początku
  Tablica::UsunElementPoczatek(); - Deklarowana jest nowa tablica o rozmiarze pomniejszonym o jeden. Wartości są kopiowane, do nowej tablicy ze starej tablicy o indeksie o jeden większy, za pomocą pętli. Następnie zwalniamy pamięć starej tablicy a wskaźnik nowej tablicy przypisujemy do starej tablicy. Zmniejszamy rozmiar o jeden;
- Usunięcie na końcu

Tablica::UsunElementKoniec(); - Podobnie jak wyżej. Element końcowy jest usuwany przez kopiowanie wartości do nowej tablicy do przedostatniego elementu.

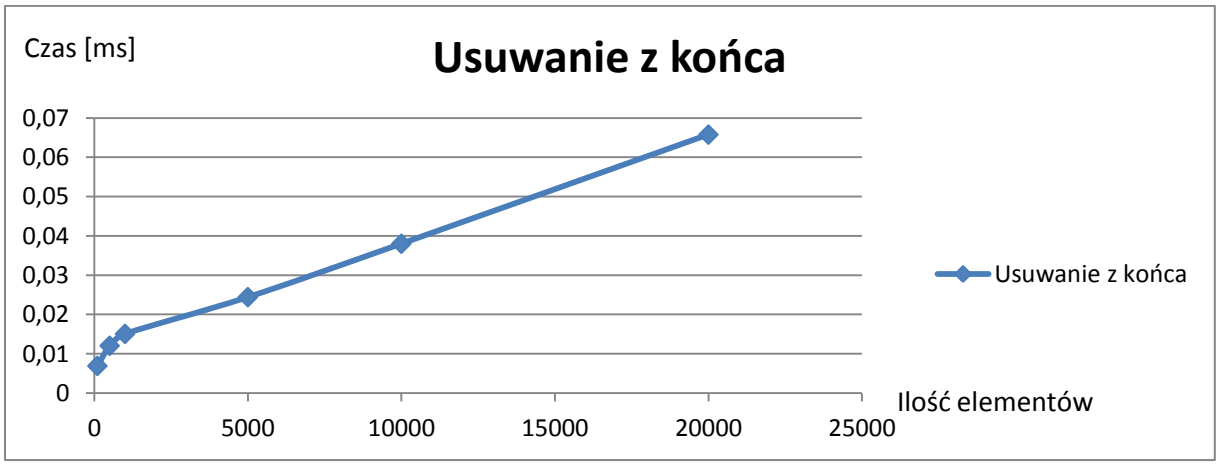
- Usunięcie na losowej pozycji
  Tablica::UsunElementLosowo(); - Metoda wykorzystuje metodę UsunElementNaPozycji(), która nie jest przewidziana w projekcie. Metoda otrzymuje jako argument zrandomizowaną wartość z przedziału rozmiaru. Następnie usuwa ją przez przekopiowanie wartości.
- Dodanie na początku
  Tablica::WstawElementPoczatkowy(int x); - Metoda alokuje pamięć dla nowej tablicy o rozmiarze o jeden większej. Na pierwszej pozycji zapisywana jest wartość x na następnych wartości ze starej tablicy.
- Dodanie na końcu
  Tablica::WstawElementKoncowy(int x); - Podobnie jak wyżej; Wartość x jest przypisywana ostatniemu elementowi nowej tablicy.
- Dodanie na losowej pozycji
  Tablica::WstawElementLosowo(int x, int k); - Metoda otrzymuje dwa argumenty x –wartość, k – zrandomizowaną wartość z zakresu rozmiaru. Używa metody WstawElementNaPozycji() by znaleźć odpowiednie miejsce i przekopiować dane.
- Wyświetlenie tablicy
  Tablica::WypiszTablice(); - Wypisuje zawartość tablicy jedna wartość po drugiej.
- Wyszukanie elementu
  Tablica::Wyszukaj(int x); - Metoda w pętli przeszukuje tablice zwraca true jeśli wartość elementu jest równa x. W przeciwnym przypadku zwraca false.

5. Pomiary funkcji tablicy

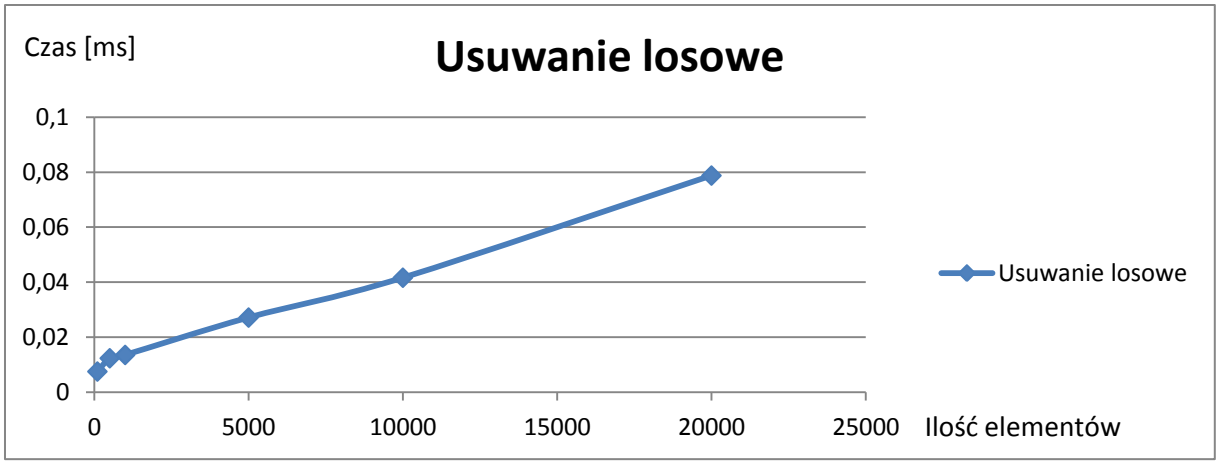
5.1. Usunięcie na początku



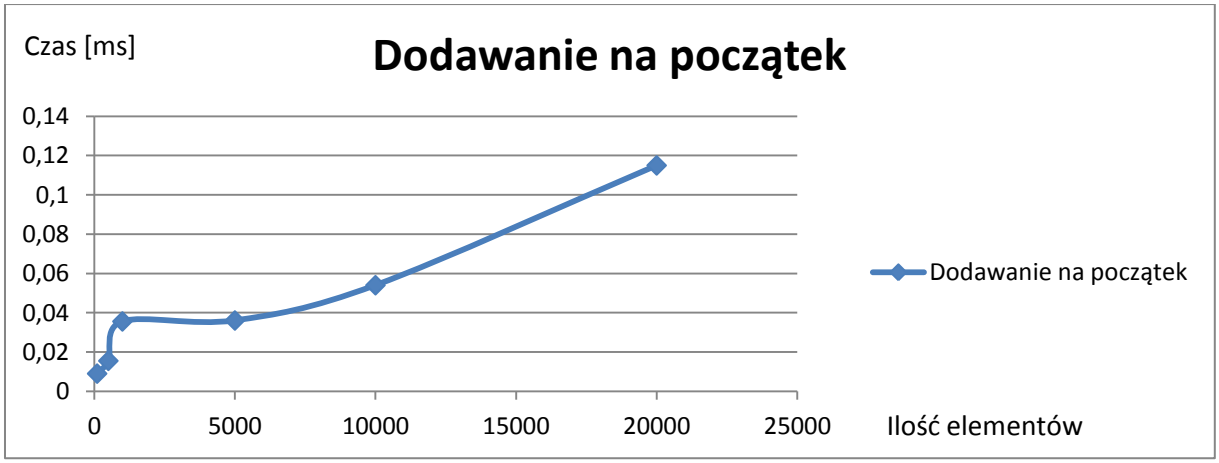
5.2.Usunięcie na końcu



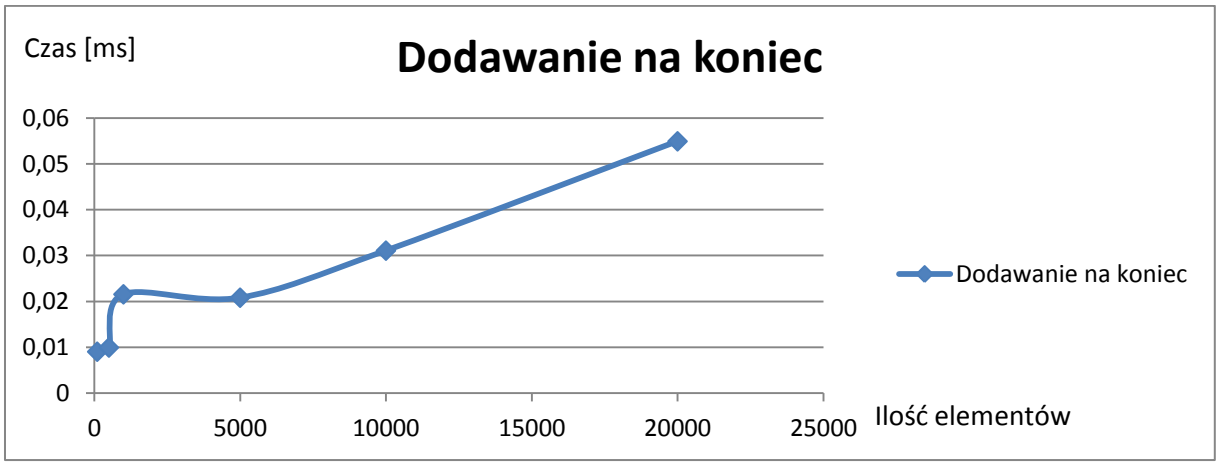
5.3.Usunięcie na losowej pozycji



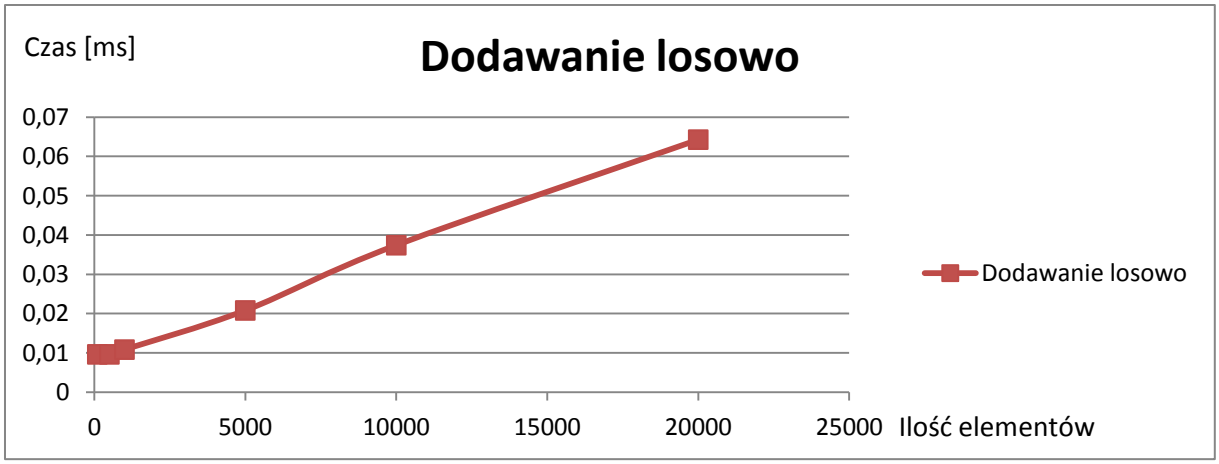
5.4.Dodanie na początku



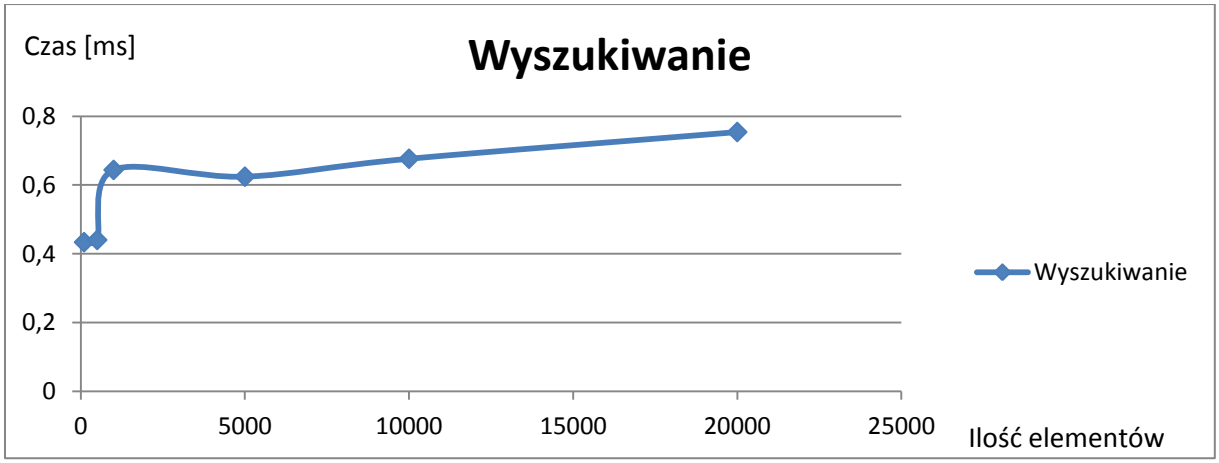
5.5.Dodanie na końcu



5.6.Dodanie na losowej pozycji



5.7.Wyszukanie elementu



6. Implementacja listy

Lista jest dynamiczną strukturą danych co oznacza, że jej rozmiar można dowolnie zmieniać. Lista składa się z połączonych elementów. Każdy element lisy połączony jest z następnym oraz poprzednim. Do obsługi listy potrzebne są dwa elementy: pierwszy element i ostatni element. Pierwszy nazywany jest głową, jego wskaźnik na element poprzedni wynosi NULL. Drugi nazywany jest ogonem, jego wskaźnik na następny element jest równy NULL. Potrzebna jest też zmienna przechowująca rozmiar listy.

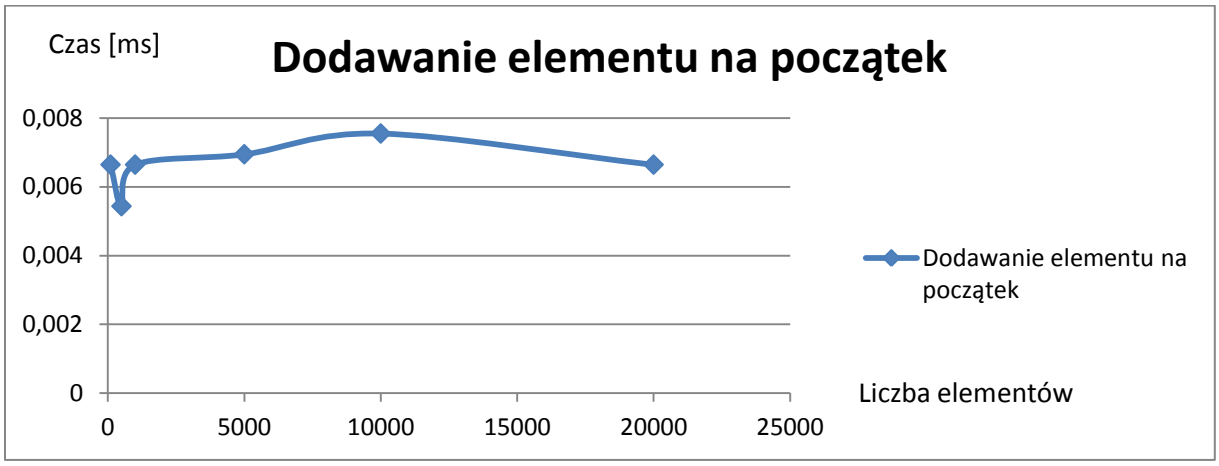
Operacje wykonywane na liście to:

- Dodaj na początek
  Metoda sprawdza czy istnieje element początkowy. Jeśli nie, to zostanie utworzony nowy element listy, który będzie jednocześnie pierwszym i ostatnim elementem listy. Jeśli istnieje pierwszy element, to tworzymy nowy element ustawiamy jego wartość i ustawiamy wskaźnik poprzedniego elementu jako pierwszego elementu. Zwiększamy rozmiar listy.
- Dodaj na koniec
  Dodawanie elementu na koniec listy jest symetryczne z dodawaniem na początek. Tworzymy nowy element i umieszczamy w nim dane. We wskaźniku na element następny wstawiamy adres zerowy (NULL) – ostatni element nie posiada następnika, w polu wskaźnika na poprzedni element wstawiamy adres obecnego ogona. Nowemu elementowi nadajemy status ogona i zwiększamy ilość elementów o jeden oraz do pola następnego elementu w poprzednim ogonie wstawiamy adres dodanego elementu.
- Dodaj na pozycje
  Do metody przekazywane są dwa argumenty wartość i pozycja. Sprawdzane jest czy pozycja jest poprawna. Jeśli pozycja jest równa zero to wywołujemy funkcję dodaj na początek.
  Jeśli pozycja jest równa ostatniej pozycji to wywołujemy funkcję dodaj na koniec.
  Następnie przeszukujemy tylko połowę listy by znaleźć element na odpowiedniej pozycji. Po znalezieniu przepisujemy wskaźniki na następny i poprzedni element dla dodawanego elementu i zwiększamy rozmiar o jeden;
- Dodaj na losową pozycje
  Metoda otrzymuje jako argumenty wartość. Pozycja jest randomizowana. Wartość i pozycja są przekazywane jako argumenty do metody dodaj na pozycje.
- Usuń pierwszy
  Drugiemu elementowi na liście ustawiamy jako pierwszy element. Usuwamy pierwszy element. Sprawdzane jest czy w liście są jeszcze jakieś elementy. Jeśli tak, to aktualny element jest ustawiany jako pierwszy. Jeśli nie, lista jest zerowana. Zmniejszony zostaje rozmiar o jeden.
- Usuń ostatni
  Analogicznie jak w przypadku początku listy – elementowy poprzedzającemu ogon nadajemy status nowego ‘ogona’ a do jego adresu następcy wstawiamy NULL. Usuwamy niepotrzebny element i zmniejszamy licznik wskazujący liczbę elementów.

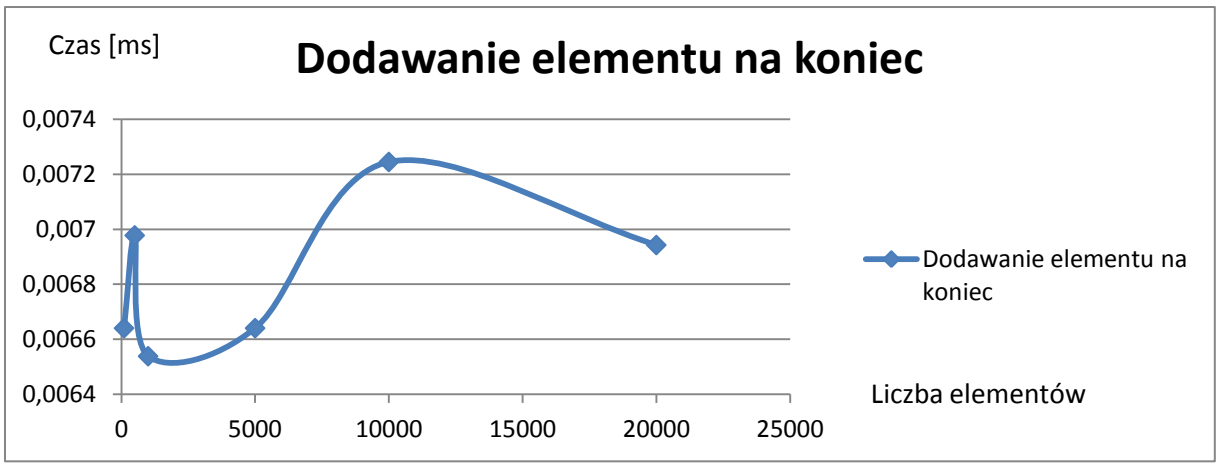
- Usuń element
  Jeśli wybrany element to głowa lub ogon wywoływana jest odpowiednia funkcja.
  Przeszukiwana jest połowa listy w celu odnalezienia wartości. Tworzony jest nowy element.
  Przekazywane są wskaźniki na poprzedni i następny element.
- Wyszukaj
  Sprawdzany jest warunek rozmiaru listy. Jeśli jest pusta zwracany jest false.
  Jeśli nie jest pusta, lista jest przeszukiwana w pętli for pod względem wartości. Po znalezieniu wartości wyświetlana jest informacja o indeksie/pozycji elementu oraz zwracane jest true.
- Wypisz listę
  Ustawiany jest pierwszy element jako aktualny. W pętli for następuje wypisywanie pozycji i wartości elementów listy oraz zmieniany jest element na następny element.

7. Pomiary funkcji listy

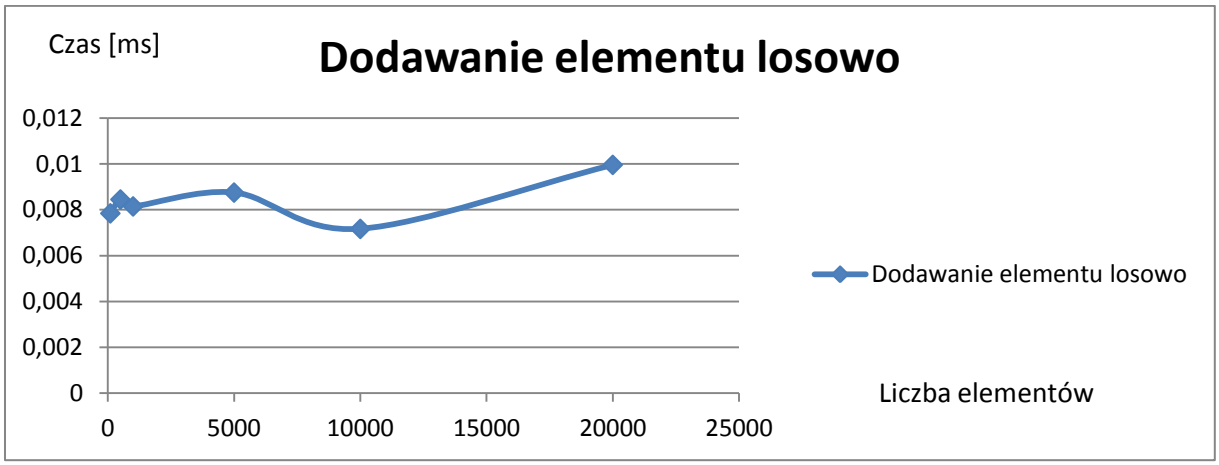
7.1. Dodaj na początek



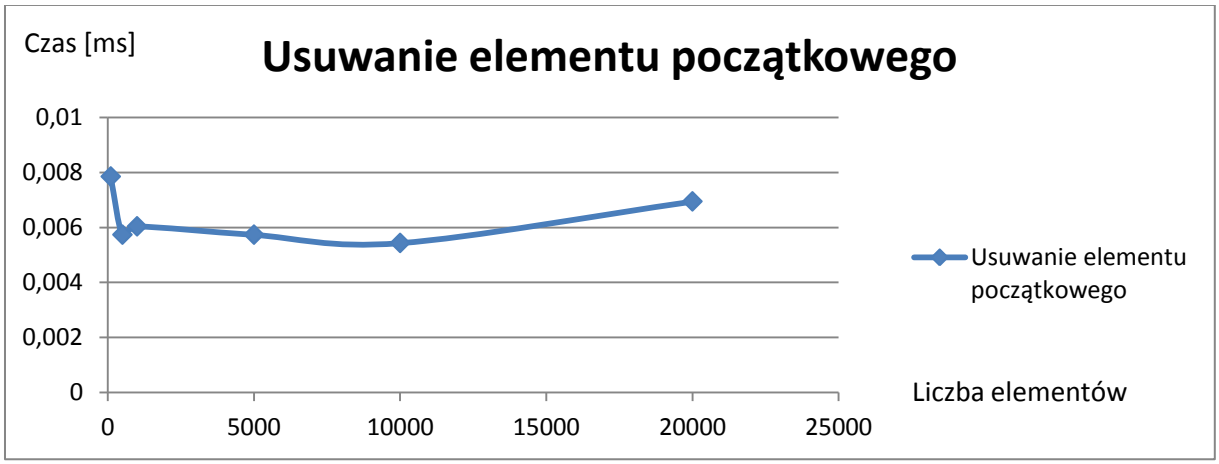
7.2.Dodaj na koniec



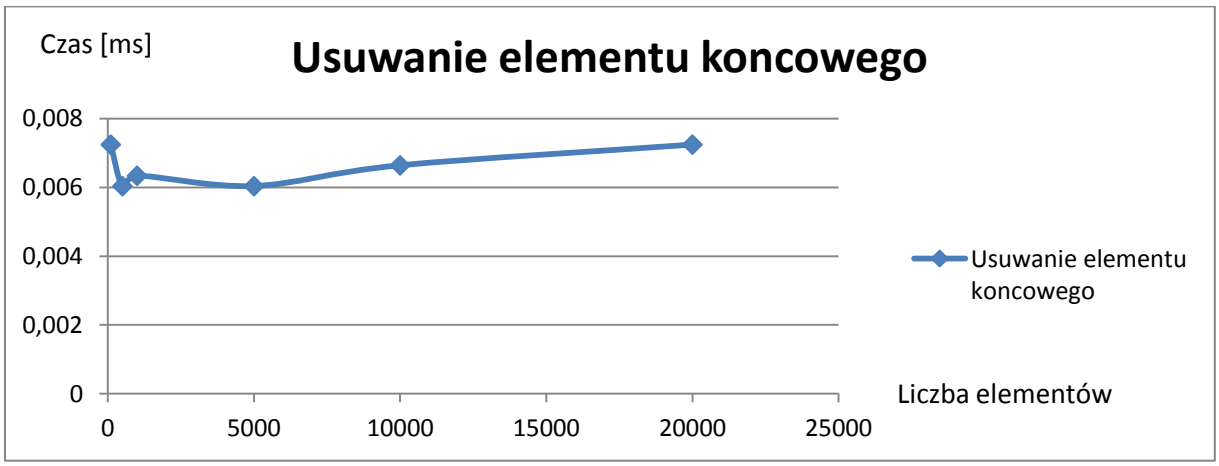
7.3.Dodaj na losową pozycje



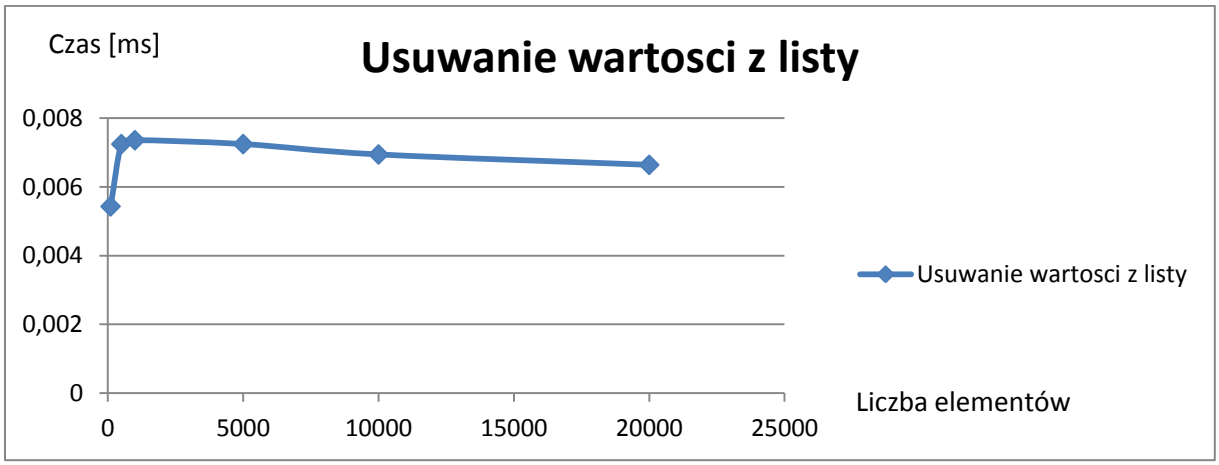
7.4.Usuń pierwszy



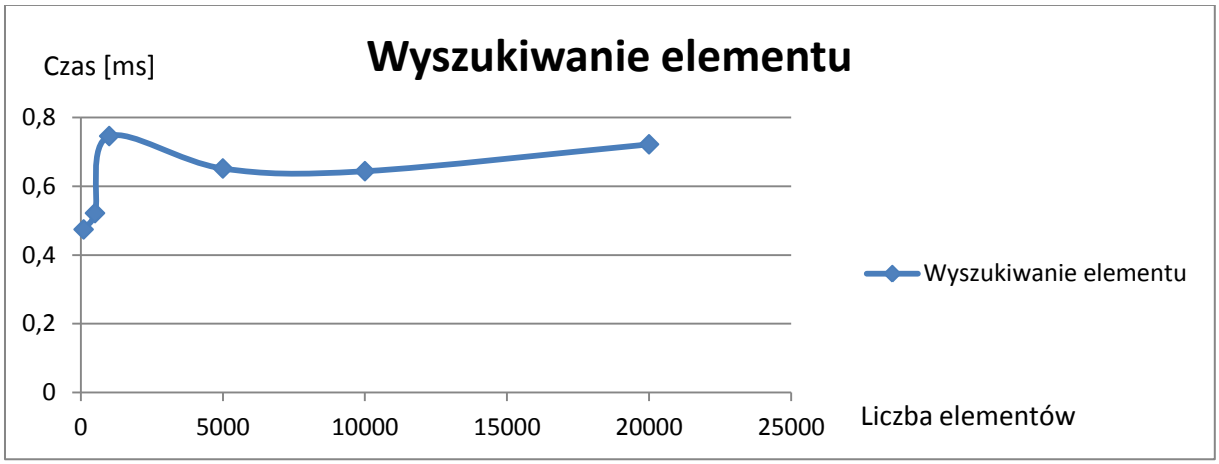
7.5.Usuń ostatni



7.6.Usuń element



7.7.Wyszukaj



8. Implementacja kopca

Kopiec binarny jest zupełnym drzewem binarnym. Kopiec to tablicowa struktura danych, którą można rozpatrywać jako pełne drzewo binarne. Tablica A reprezentująca kopiec ma dwa atrybuty rozmiar – określający liczbę elementów tablicy i heap-size[A] – określający liczbę elementów kopca przechowywanych w tablicy. Korzeniem drzewa jest A[1]. Mając indeks i węzła, można łatwo obliczyć indeksy jego ojca PARENT(i){(i-2)/2}, lewego syna LEFT(i){2*i+1} i prawego syna RIGHT(i){2i+2}. Dla każdego węzła i, który nie jest korzeniem zachodzi równość A[PARENT(i)]≥A[i]. Wysokość węzła to liczba krawędzi na najdłuższej prostej ścieżce prowadzącej od węzła do liścia. Wysokość drzewa to wysokość jego korzenia.

Operacje wykonywane na kopcu:

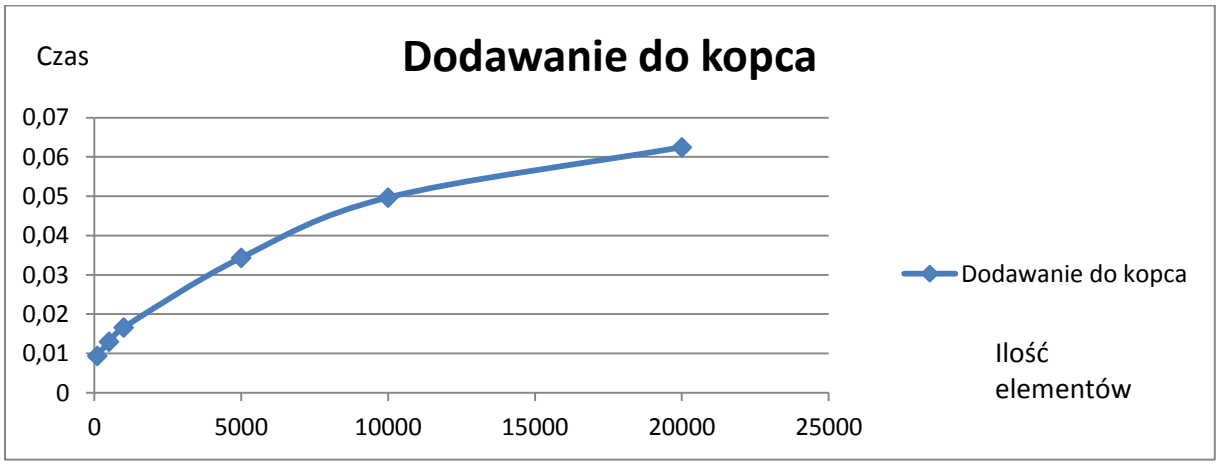
- Dodawanie elementu
  Dodawanie elementu rozpoczyna się od sprawdzenia czy podana wartość nie znajduje się już w kopcu. Działanie ma to na celu utrzymanie poprawności struktury. Sprawdzenia tego dokonuje osobna metoda. Tworzona jest nowa tablica o rozmiarze o jeden większy.

Wartości ze starej tablicy są przepisywane do nowej w pętli for. Na ostatniej pozycji przypisuje się wartość. Zwalniana jest pamięć starej tablicy. Wskaźnikowi tablicy jest przypisywany adres nowej tablicy. Wywoływana jest metoda przywracająca właściwości kopca. Rozmiar zwiększany jest o jeden.

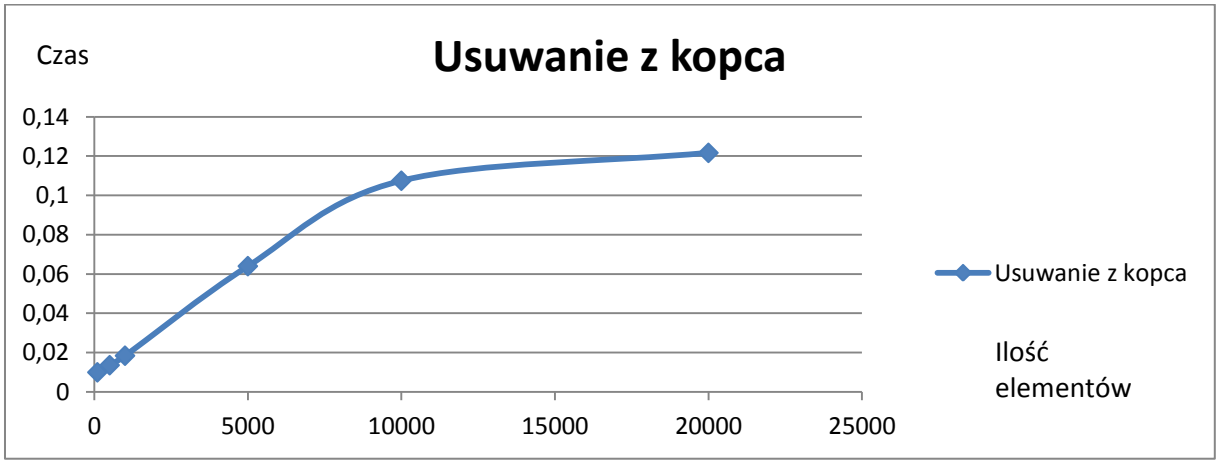
- Usuwanie elementu
  Sprawdzany jest warunek czy wartość znajduje się w kopcu. Jeśli tak, tworzona jest nowa tablica o rozmiarze o jeden mniejszym. Przekopiowywane są elementy do wskazanej wartości oraz po wskazanej wartości. Zwalniana jest pamięć starej tablicy. Wskaźnikowi starej tablicy jest przypisywany adres nowej tablicy. Rozmiar zmniejszany jest o jeden. Wywoływana jest metoda przywracająca poprawność struktury.
- Wyszukiwanie wartości
  Przeszukiwana jest tablica. Jeśli element znajduje się w tablicy metoda zwraca true. Jeśli nie, metoda zwraca false.

9. Pomiary funkcji kopca

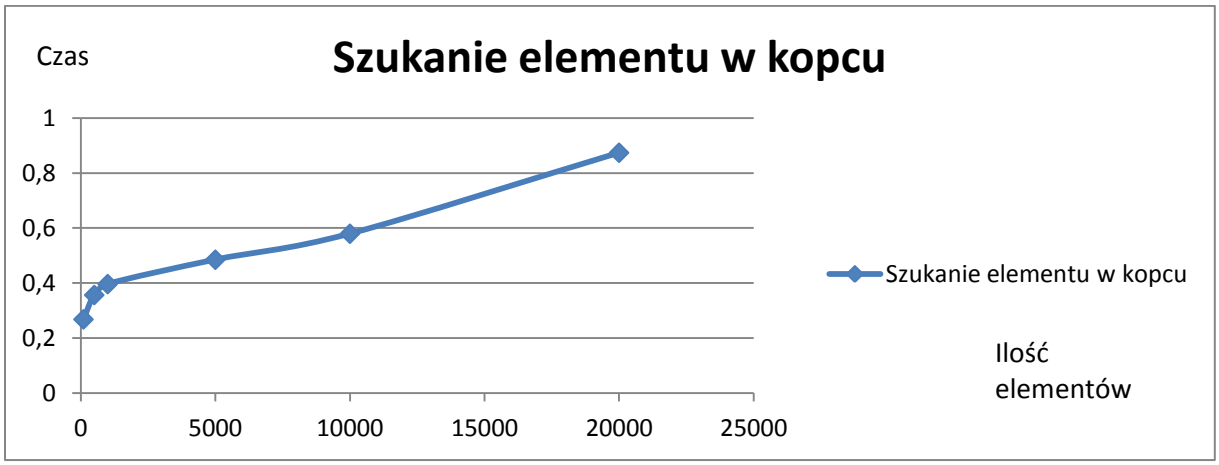
9.1. Dodawanie elementu



9.2. Usuwanie elementu



9.3. Wyszukiwanie wartości



10. Wnioski

Czas wykonywania operacji może zależeć od kilku czynników. Może zależeć od wielkości liczby. Choć nie jest to czynnik dominujący, ponieważ wszystkie liczby są 4 bajtowe różnica pomiędzy kodowaniem liczb może być znacząca. Kolejnym elementem wpływającym na czasu operacji mogą być procesy komputera, które zajmują pamięć RAM przez co kompilacja i działanie programu/algorytmów mogło być zakłócane. Rozwiązaniem tego problemu mogłoby być testowanie algorytmów w wyznaczonej przestrzeni jak na przykład maszynie wirtualnej. Istotnym czynnikiem w czasie wykonywania operacji może być sposób implementacji algorytmów. Te same problemy można rozwiązać na kilka sposobów na przykład struktury takie jak kolejki FIFO, LIFO oraz stosy można zrealizować za pomocą listy lub implementacji tablicowej.
Najbardziej efektywną strukturą okazuje się lista indeksowy dostęp do danych w tablicy przegrywa pod kątem częstej relokacji pamięci przez co tablice zajmują relatywnie więcej czasu na obliczenia. Choć trzeba zaznaczyć, że w pesymistycznym przypadku to listy są gorszymi strukturami od tablic. Każda z przedstawionych struktur ma swoje wady i zalety. Tablica posiada wartości i indeksy, które mogą być łatwo i intuicyjnie zmieniane. Lista posiada wskaźniki na element następny i poprzedni co ułatwia operacje na sąsiednich elementach. Kopiec ustawia wartości względem wielkości.

11. Bibliografia

- „Wprowadzenie do algorymtów” – Thomas H. Cormen
- „Algorytmy struktury danych i techniki programowania” – Piotr Wróblewski
- „Data Structures and Program Design In C++” – Robert L. Kruse