Struktury Danych i Złożoność Obliczeniowa

S P R A W O Z D A N I E

Prowadzący	Dr inż. Jarosław Mierzwa
Student	Szymon Wiśniewski, 241269
Termin zajęć	Piątek, 9^{15}-11^{00} TP
Numer zadania	1
Data oddania sprawozdania	22.03.2019

Spis treści

1. Założenia projektu

W ramach projektu należało napisać własną implementację zadanych struktur – tablica, lista dwukierunkowa, kopiec binarny (typu maksimum), drzewo poszukiwań binarnych BST oraz drzewo czerwono-czarne wraz z obsługą operacji, które można na nich wykonywać (np. dodawanie, usuwanie, generowanie). Zrealizowano jedynie pierwsze trzy struktury. Podczas programowania korzystano ze zintegrowanego środowiska graficznego CLion na platformie Linux. Ostateczny program istnieje w wersji konsolowej jako plik wykonywalny .exe i działa na systemie operacyjnym Windows.

Zgodnie z zaleceniami, podstawowym elementem struktur jest 4 bajtowa liczba całkowita ze znakiem, a wszystkie struktury alokowane są dynamicznie i relokowane po dodawaniu oraz usuwaniu elementów.

2. Wstęp teoretyczny

2.1. Złożoność obliczeniowa

Złożoność czasowa charakteryzuje ilość czasu niezbędnego do rozwiązania problemu z zależności od ilości danych wejściowych. Wyrażamy ją w jednostkach czasu albo w liczbie operacji dominujących, które trzeba wykonać dla n danych, aby otrzymać rozwiązanie problemu.

Złożoność pamięciowa informuje o liczbie komórek pamięci, która będzie zajęta przez dane i wyniki pośrednie tworzone w trakcie pracy algorytmu.

W doświadczeniu badana będzie jedynie złożoność czasowa.

Definicje poboczne:

Problem – zbiór danych wejściowych, ich definicje oraz pytanie lub polecenie do wykonania.

Algorytm – procedura działająca w skończonym czasie, która rozwiązuje dany problem.

2.2. Tablica

Tablica jest relatywnie prostą strukturą danych, która składa się z ciągu elementów danego typu. Elementy tablicy w pamięci komputera ułożone są koło siebie, jest to zwarta struktura. Do wybranego elementu uzyskujemy dostęp poprzez jego indeks, który obliczany jest na podstawie rozmiaru struktury oraz rozmiaru pojedynczego elementu. Numeracja zaczyna się od 0, więc indeks ostatniego elementu jest równy *size - 1* (size – ilość elementów w strukturze, to oznaczenie będę stosował również dla pozostałych struktur).

Zaimplementowana tablica składa się z 3 atrybutów:

- wskaźnik na tablicę (int *tab),
- rozmiar tablicy (int size),
- wskaźnik na tymczasową tablicę (int *temp), która konieczna jest do przeprowadzenia niektórych operacji na strukturze.

Lista metod – operacji, które można wykonać na zaimplementowanej strukturze:

- wczytanie struktury z pliku (int loadFromFile(string FileName)),
- sprawdzenie, czy dana wartość występuje w tablicy (bool IsValueInTable(int value)),
- dodanie wartości w wybranym miejscu (void addValue(int index, int value)),
- dodanie wartości na początku (void addValueStart(int value)),
- dodanie wartości na końcu (void addValueEnd(int value)),
- dodanie wartości w losowym miejscu (void addValueRandom(int value)),
- usuniecie elementu z wybranego miejsca (void deleteFromTable(int index)),
- usuniecie z początku (void deleteFromTableStart()),
- usuniecie z końca (void deleteFromTableEnd()),
- usuniecie z losowego miejsca (void deleteFromTableRandom()),
- wyświetlenie tablicy (void display()) w formie
  [0] wartość [1] wartość … [size - 1] wartość,
- wygenerowanie struktury o podanym rozmiarze (void generateTable(int size)).

Złożoność obliczeniowa tablicy:

	Złożoność czasowa		Złożoność pamięciowa
	Średnia	Najgorsza	Najgorsza
Dostęp	O(1)	O(1)	O(n)
Wyszukiwanie	O(n)	O(n)	
Dodawanie	O(n)	O(n)	
Usuwanie	O(n)	O(n)	

2.3. Lista dwukierunkowa

Drugą zaimplementowaną strukturą jest lista dwukierunkowa. W przeciwieństwie do tablicy, lista nie musi być zwartą w pamięci strukturą (mimo to, jeśli jest taka możliwość, rezerwowane są miejsca bezpośrednio koło siebie). Lista składa się z węzłów (node), które związane są ze sobą poprzez wskaźniki. Każdy węzeł zawiera określoną wartość (int value), wskaźnik na następny węzeł (Node *next) oraz poprzedni (Node *previous). Wyjątkami są węzeł początkowy, tzw. głowa (head) oraz ostatni, ogon (tail). W przypadku pierwszego węzła, wskaźnik previous wskazuje na NULL, natomiast w ogonie next na NULL. W porównaniu z tablicą, dostęp do elementów jest wolniejszy, bo należy przejść przez każdy poprzedzający element. Wygrywa jednak na polu dodawania oraz usuwania elementów – wystarczy zmienić wskazania wskaźników next oraz previous na inne elementy.

Zaimplementowana lista składa się z następujących atrybutów:

- głowa (Node *head),
- ogon (Node *tail),
- ilość elementów w liście (int size).

Lista metod:

- wczytanie struktury z pliku (int loadFromFile(string FileName)),

- sprawdzenie, czy dana wartość występuje w liście (bool IsValueInList (int value)),
- dodanie wartości w wybranym miejscu (void addValue(int index, int value)),
- dodanie wartości na początku (void addValueStart(int value)),
- dodanie wartości na końcu (void addValueEnd(int value)),
- dodanie wartości w losowym miejscu (void addValueRandom(int value)),
- usuniecie elementu o wybranej wartości (void deleteFromList (int value)),
- usuniecie z początku (void deleteFromListStart()),
- usuniecie z końca (void deleteFromListEnd()),
- usuniecie z losowego miejsca (void deleteFromListRandom()),
- wyświetlenie listy (void display()) w formie

  głowa … węzezły … ogon
- wygenerowanie struktury o podanym rozmiarze (void generateList(int size)),
- czyszczenie listy (void clearList()) – wymagane na przykład przed wygenerowaniem nowej.

Złożoność obliczeniowa listy:

	Złożoność czasowa		Złożoność pamięciowa
	Średnia	Najgorsza	Najgorsza
Dostęp	O(n)	O(n)	O(n)
Wyszukiwanie	O(n)	O(n)	
Dodawanie	O(1)	O(1)	
Usuwanie	O(n)	O(n)	

2.4. Kopiec (maksymalny)

Kopiec jest strukturą danych opartą na drzewie, w której wartości potomków pozostają w stałej relacji z rodzicem. Charakteryzuje się tak zwaną własnością kopca. W przypadku kopca maksymalnego, w korzeniu (element na samej górze, który nie ma rodzica) umiejscowiony jest element o największej wartości. Poza tym, każdy węzeł nadrzędny – tzw.

rodzic, jest większy lub równy od swojego potomka – dziecka (w przypadku kopca minimalnego występuje relacja odwrotna). Kopiec można przedstawić za pomocą indeksowanej tablicy (w mojej implementacji od zera).

W przypadku indeksowania od zera, indeksy poszczególnych rodziców oraz potomków można obliczyć ze wzorów:

$$parentIndex = \frac{(childIndex - 1)}{2}$$

$$leftChildIndex = parentIndex \cdot 2 + 1$$

$$rightChildIndex = parentIndex \cdot 2 + 2$$

Zaimplementowany kopiec składa się z następujących atrybutów:

- rozmiar kopca (int size),
- maksymalny rozmiar (int max_size)
- wskaźnik na tablicę przechowującą elementy kopca (*arr).

Lista metod:

- wczytanie struktury z pliku (int loadFromFile(string FileName)),
- sprawdzenie, czy dana wartość występuje w kopcu (bool IsValueInList (int value)),
- przywracanie własności kopca w górę (void heapifyUp(int index)),
- przywracanie własności kopca w dół (void heapifyDown(int index)),
- dodanie elementu o wybranej wartości (void addValue (int value)),
- usuniecie elementu o wybranej wartości (void deleteFromHeap (int value)),
- pomocnicza funkcja swap służąca do zmiany położenia elementów w tablicy (void swap(int a, int b)),
- wyświetlenie kopca (void display()) w formie drzewa oraz tablicy,
- wygenerowanie struktury o podanym rozmiarze (void generateHeap(int size)),
- czyszczenie kopca (void clearHeap()) – wymagane na przykład przed wygenerowaniem nowej,

- funkcja pomocnicza do pobierania indeksu rodzica danego elementu (int getParentIndex(int childIndex)),
- funkcja pomocnicza do pobierania indeksu lewego potomka danego elementu (int getLeftChildIndex(int parentIndex)
- funkcja pomocnicza do pobierania indeksu prawego potomka danego elementu (int getRightChildIndex(int parentIndex)),

Przywracanie własności kopca w górę:

- określenie indeksu rodzica względem danego indeksu,
- jeśli element o podanym indeksie jest większy od rodzica, następuje zamiana miejscami
- rekurencyjne wywołanie funkcji z indeksem rodzica.

Przywracanie własności kopca w dół:

- znalezienie indeksów lewego i prawego potomka względem danego indeksu,
- porównanie lewego dziecka z rodzicem, jeśli jest większe, zostaje ono tymczasowo największym elementem w badanym obrębie,
- porównanie prawego dziecka z aktualnie największym elementem w badanym obszarze,
- wypchnięcie potomka do góry - jeśli któryś z potomków jest większy od rodzica, następuje zamiana miejscami potomka z rodzicem (jeśli oba są większe, zamienia się największy), jeśli rodzic był większy od potomków, zamiana nie dochodzi do skutku,
- rekurencyjne wywołanie funkcji z indeksem największego elementu w danym obszarze.

Dodanie elementu do kopca:

- inkrementacja rozmiaru tablicy,,
- wstawienie danej wartości do ostatniego elementu tablicy

- przywrócenie własności kopca w górę.

Usuniecie elementu z kopca:

- znalezienie elementu w tablicy, który chcemy usunąć,
- przesunięcie ostatniego elementu z tablicy na wybraną pozycję i zmniejszenie rozmiaru tablicy,
- przywrócenie własności kopca w dół.

Złożoność obliczeniowa kopca:

	Złożoność czasowa		Złożoność pamięciowa
	Średnia	Najgorsza	Najgorsza
Dostęp	O(1)	O(1)	O(n)
Wyszukiwanie	$O(n\log_2 n)$	$O(n\log_2 n)$	
Dodawanie	$O(\log_2 n)$	$O(\log_2 n)$	
Usuwanie	$O(\log_2 n)$	$O(\log_2 n)$	

3. Plan eksperymentu

Do dokładnego pomiaru czasu skorzystano z wbudowanej biblioteki C++ *chrono*. Pobierano cykl zegara z procesora przed wykonaniem operacji i tuż po. Ich różnica dała dokładny czas z dokładnością do ± 1 nanosekundy.

Ostateczne wyniki testów są średnią arytmetyczną 1000 pomiarów wybranej przez użytkownika ilości elementów, na których wykonywane będą operacje, w tym przypadku kolejno: 100, 1 000, 10 000, 20 000 oraz 30 000. Dane do testów zostały wygenerowane pseudo losowo. Instrukcja *srand(time(NULL))* gwarantuje, że ziarno za każdym uruchomieniem programu będzie inne, więc funkcja *rand* zawsze będzie generowała inną sekwencję liczb.

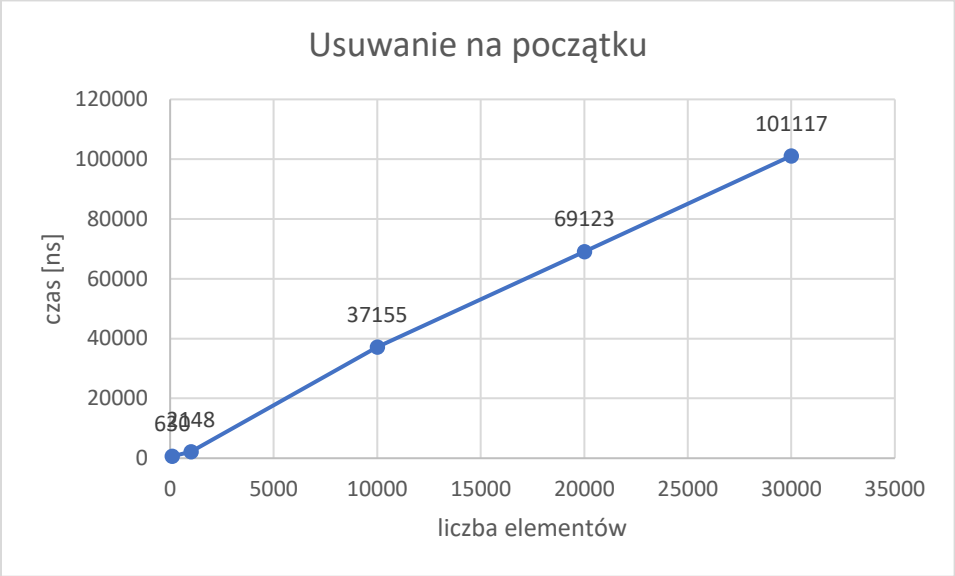
Wyniki czasowe dla konkretnych operacji zostały zestawione w arkuszu kalkulacyjnym Excel i na ich podstawie wygenerowano poniższe wykresy.

Na czas testowania wyłączono wypisywanie w konsoli informacji zwrotnych dotyczących wykonywanych operacji (np. „Nie znaleziono elementu o wartości …”) aby nie fałszować wyników.

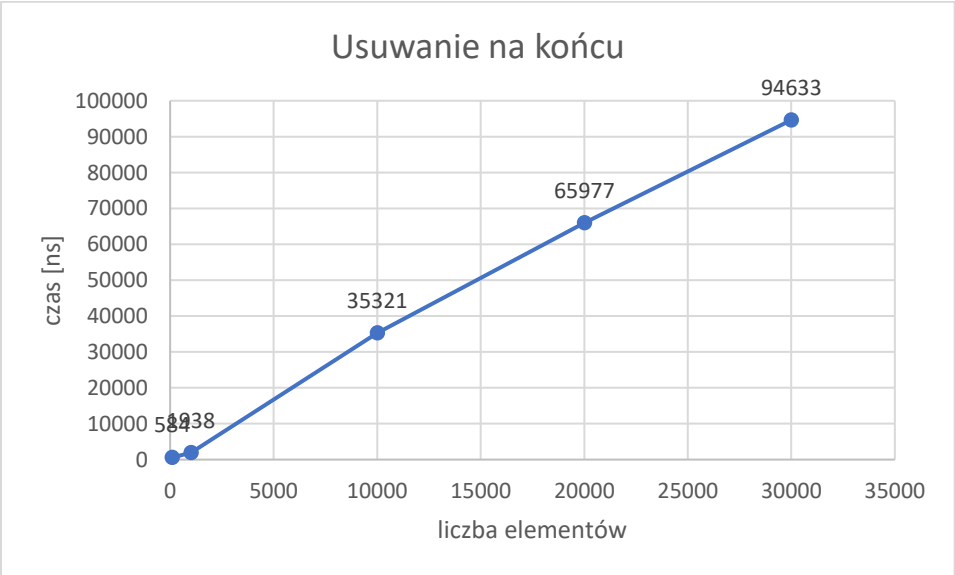
4. Zestawienie wyników

4.1. Tablica

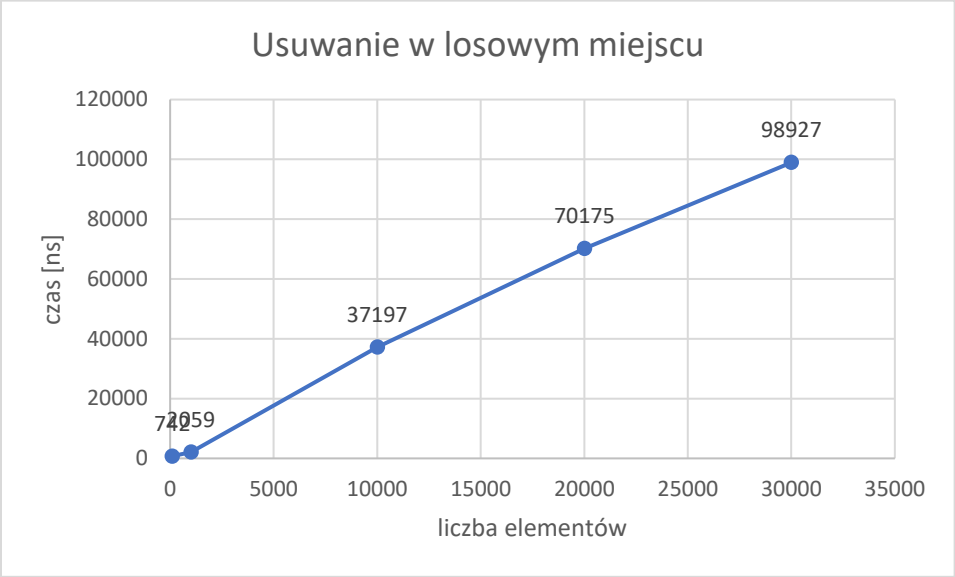
4.1.1. Usuwanie na początku



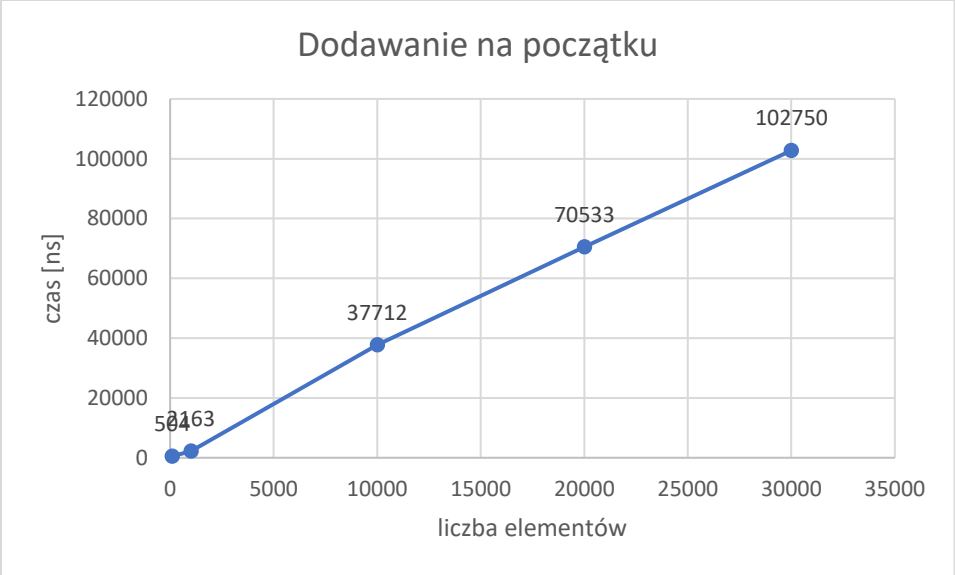
4.1.2. Usuwanie na końcu



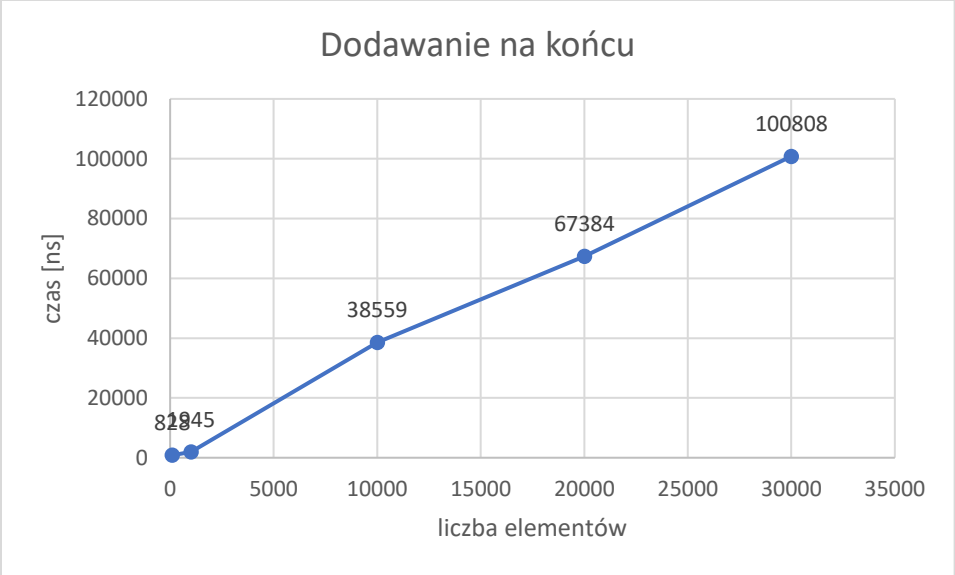
4.1.3. Usuwanie w losowym miejscu



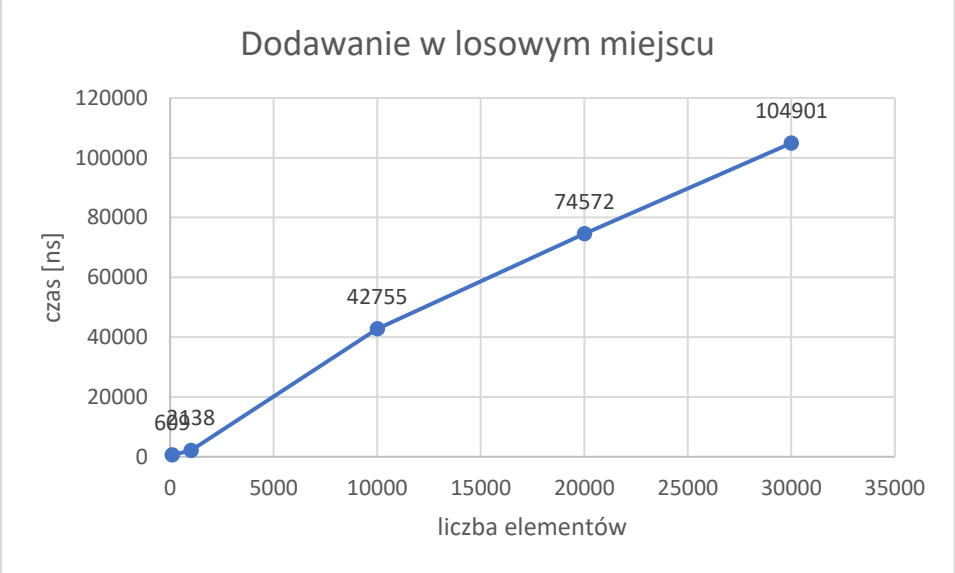
4.1.4. Dodawanie na początku



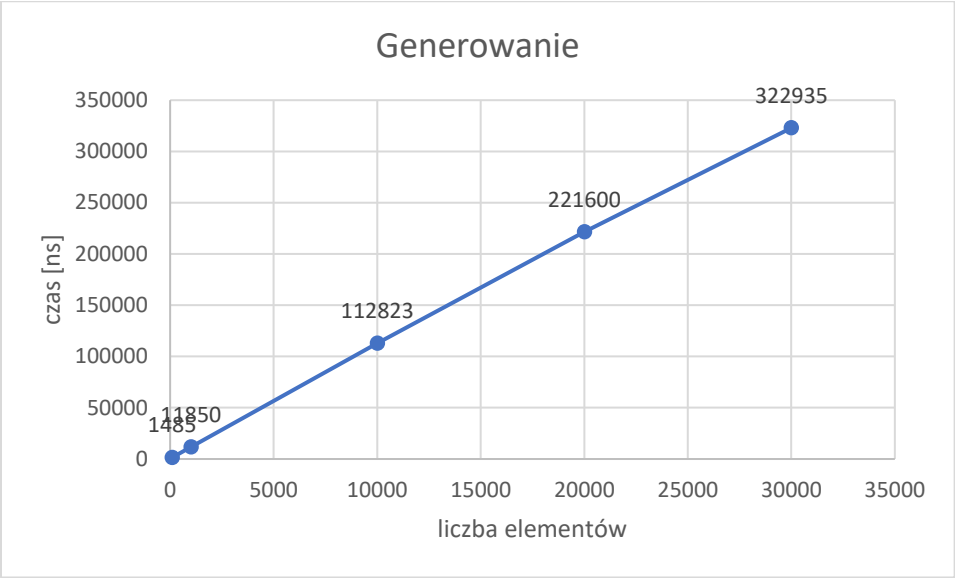
4.1.5. Dodawanie na końcu



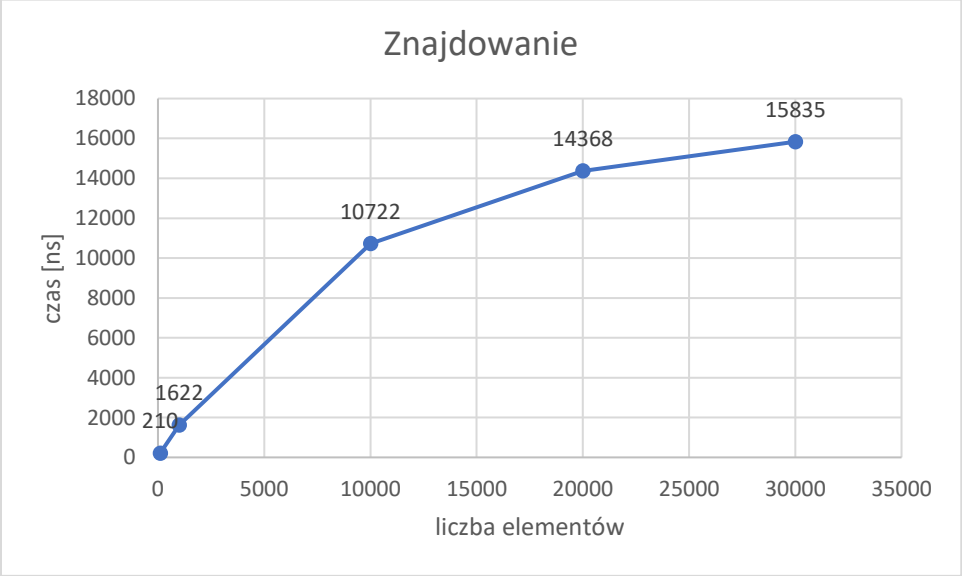
4.1.6. Dodawanie w losowym miejscu



4.1.7. Generowanie struktury o określonym rozmiarze z elementami o losowych wartościach

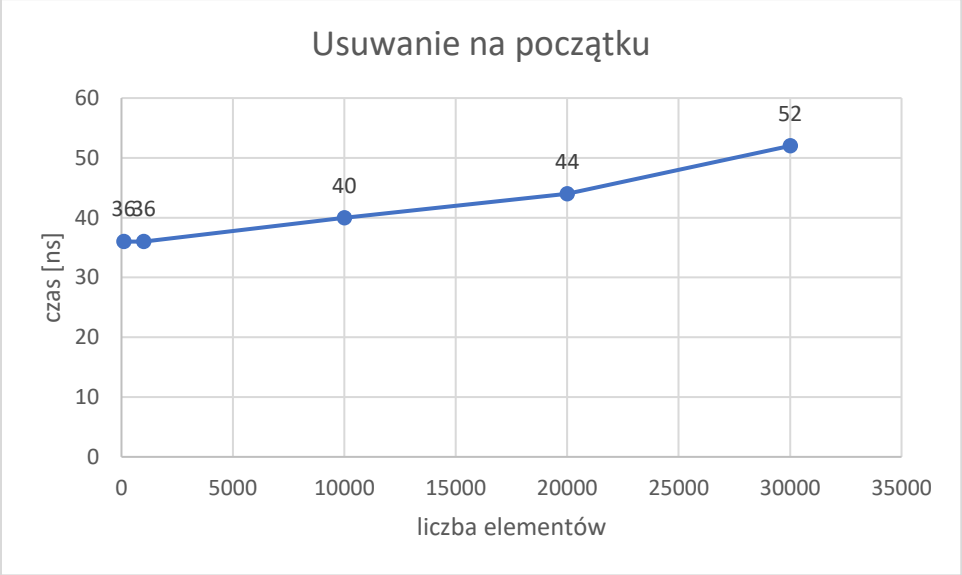


4.1.8. Znajdowanie elementu

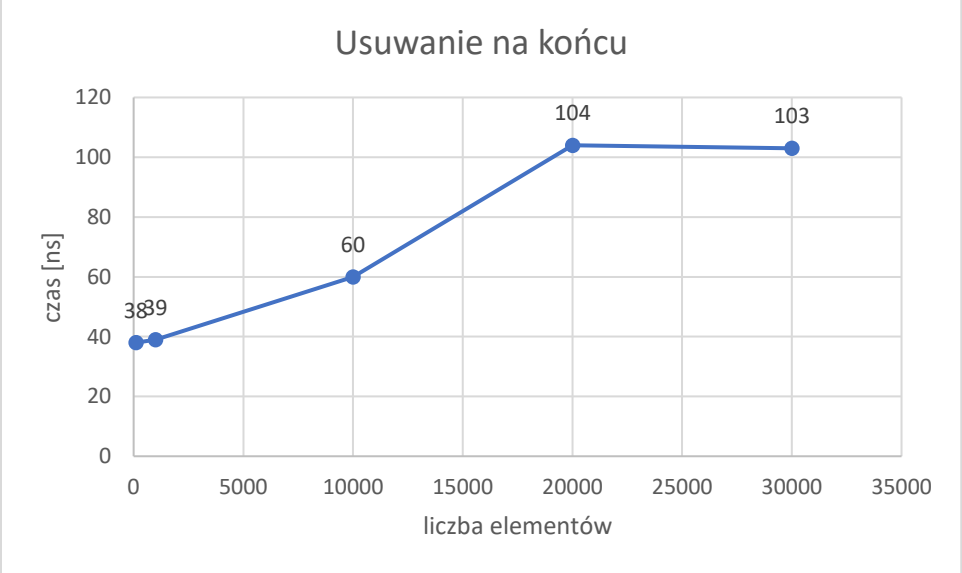


4.2. Lista dwukierunkowa

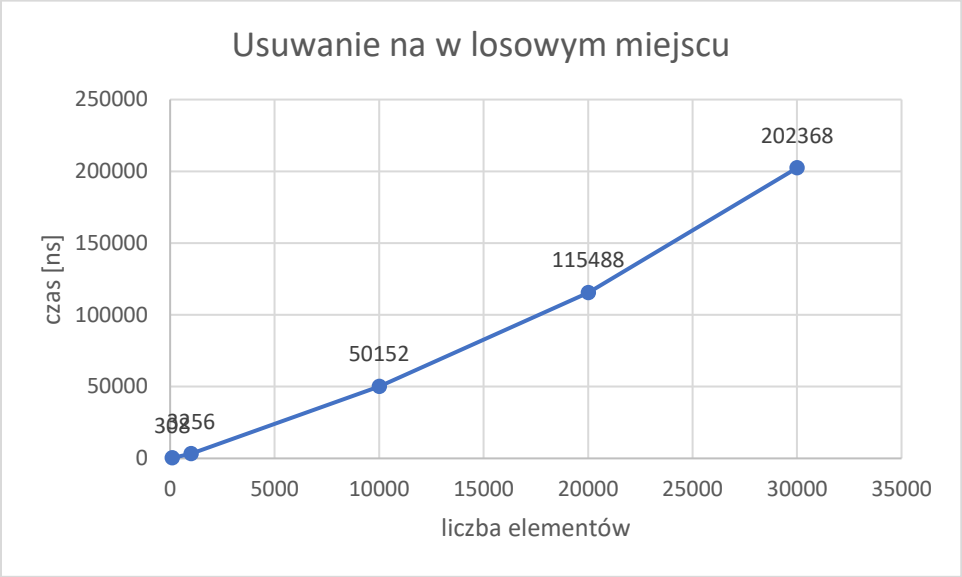
4.2.1. Usuwanie na początku



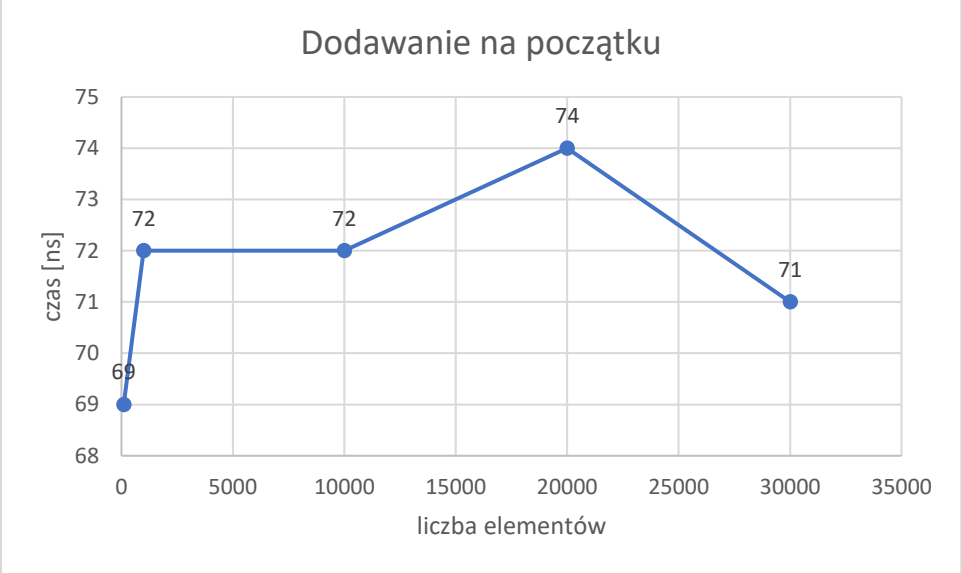
4.2.2. Usuwanie na końcu



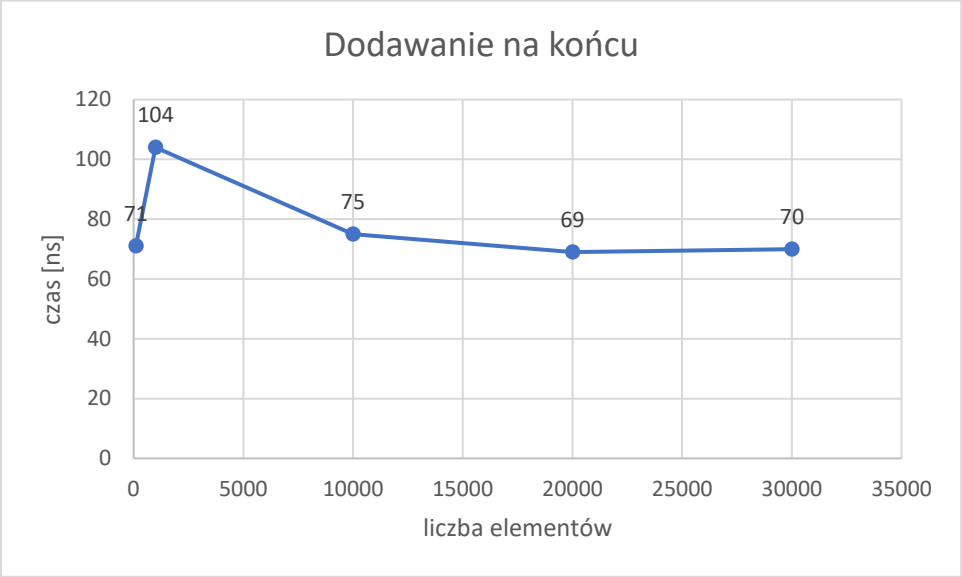
4.2.3. Usuwanie w losowym miejscu



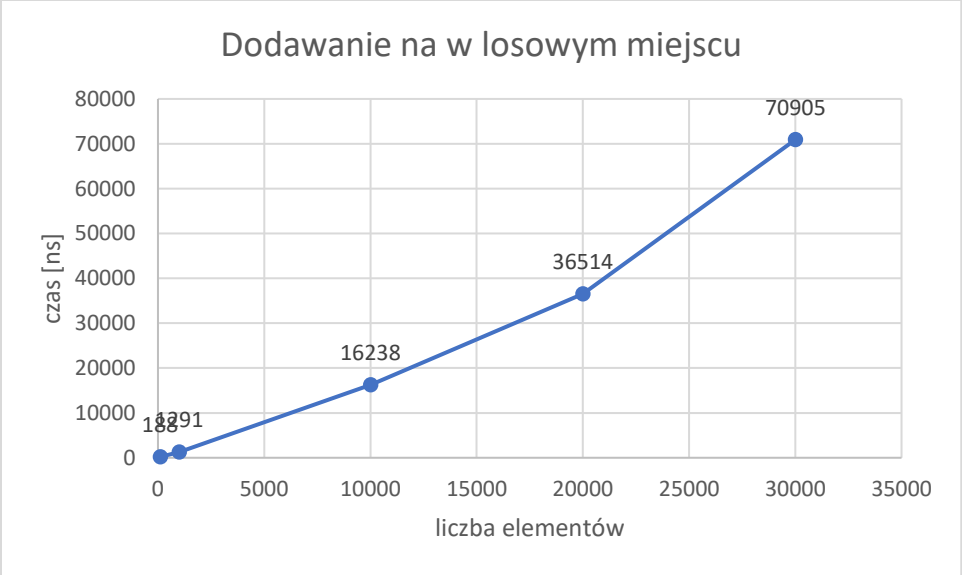
4.2.4. Dodawanie na początku



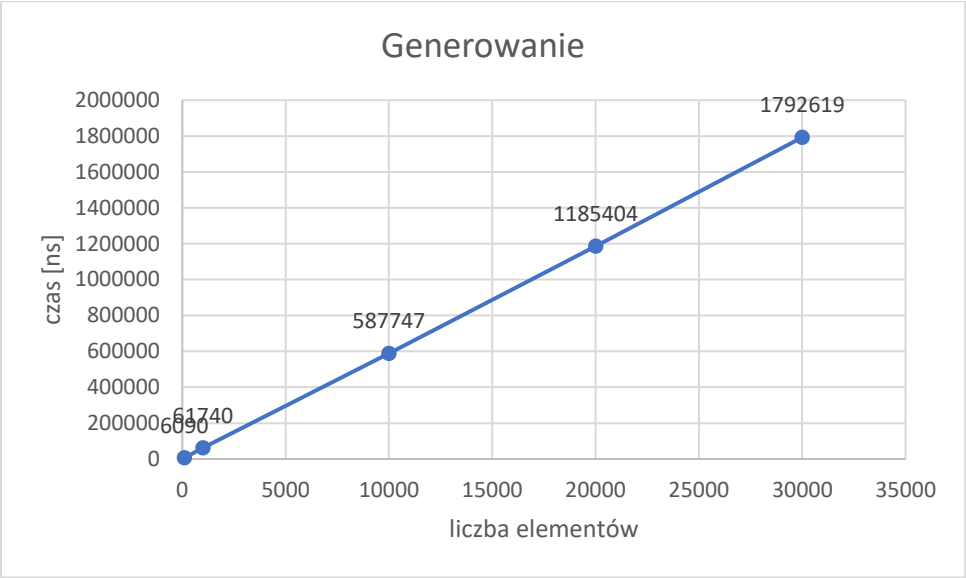
4.2.5. Dodawanie na końcu



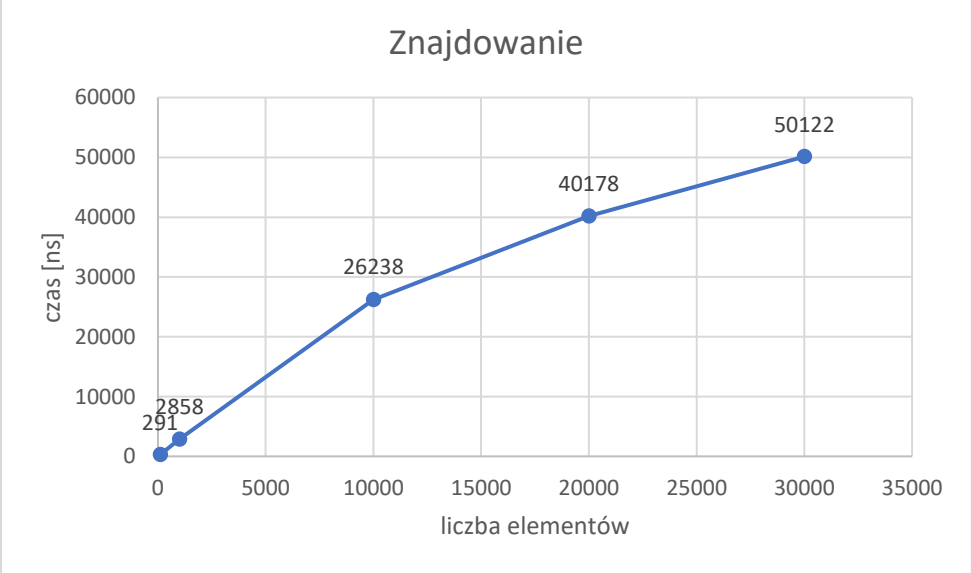
4.2.6. Dodawanie w losowym miejscu



4.2.7. Generowanie struktury o określonym rozmiarze z elementami o losowych wartościach

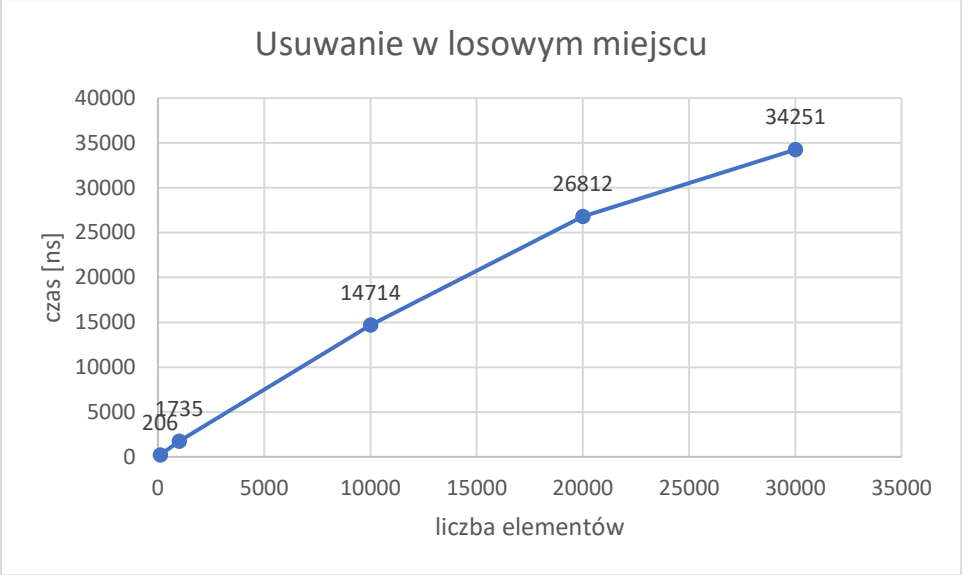


4.2.8. Znajdowanie

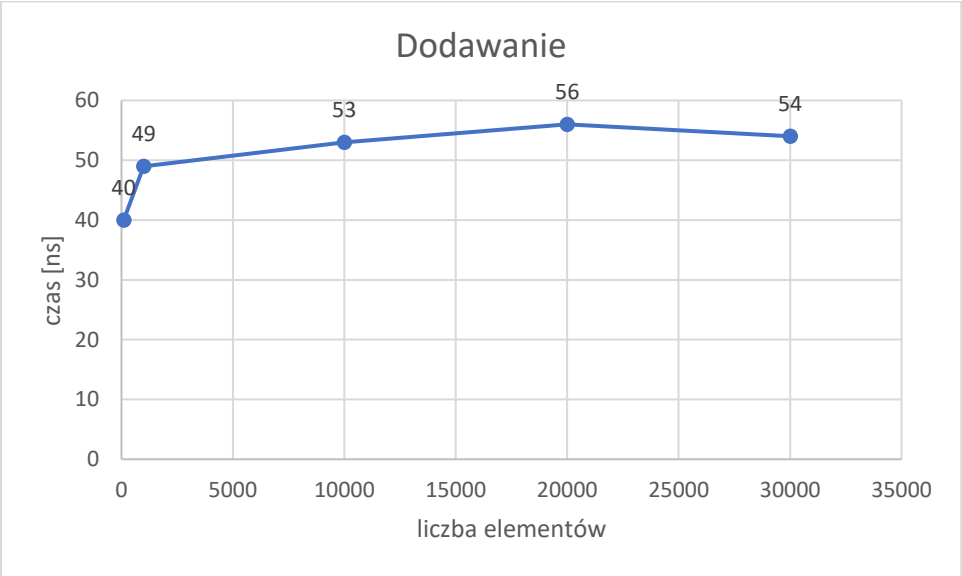


4.3. Kopiec

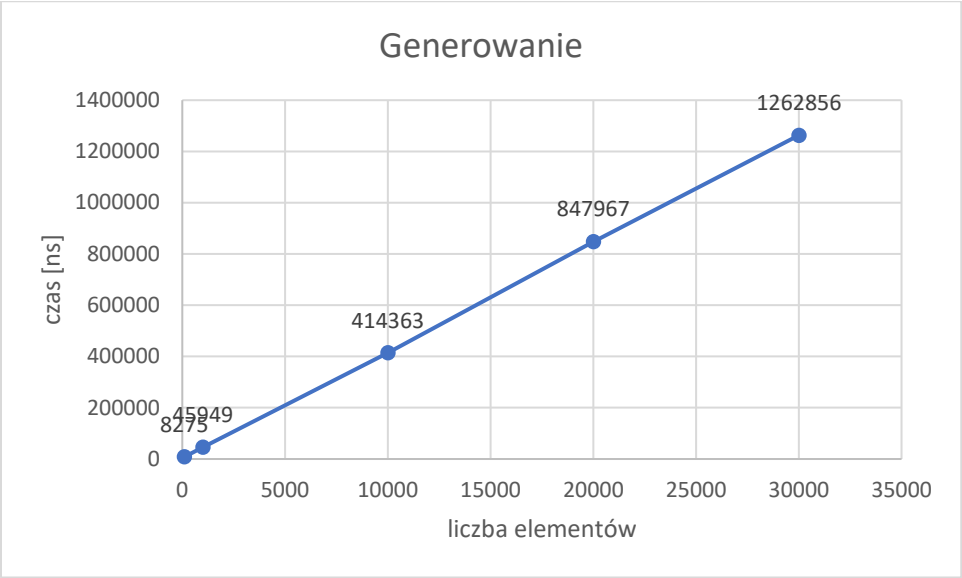
4.3.1. Usuwanie w losowym miejscu



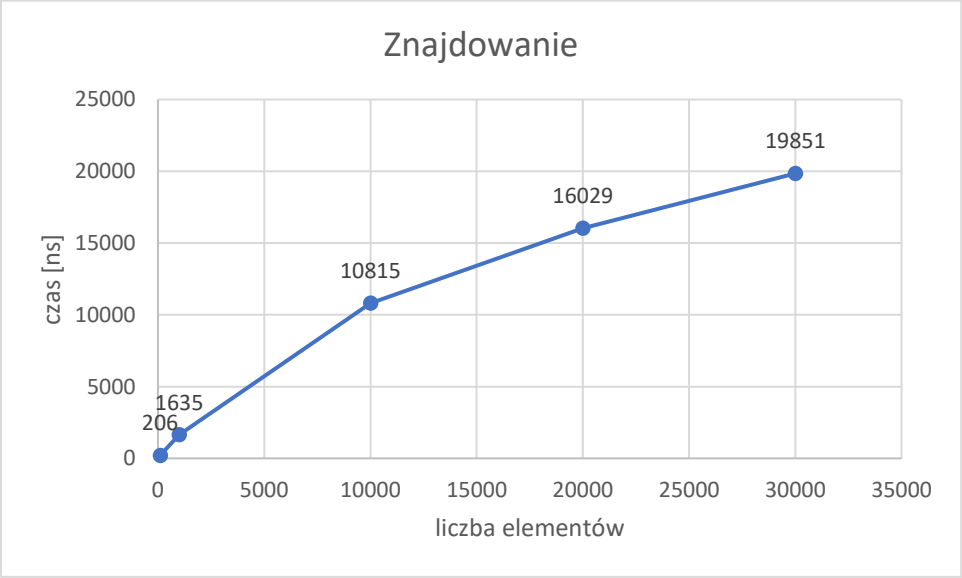
4.3.2. Dodawanie



4.3.3. Generowanie struktury o określonym rozmiarze z elementami o losowych wartościach



4.3.4. Znajdowanie



5. Wnioski dotyczące efektywności poszczególnych struktur

5.1. Tablica

Wyniki w dużej mierze pokrywają się z teoretycznymi założeniami. Dla dodawania elementów oraz ich usuwania zawsze wychodzi O(n). Zaskakujące jest wyszukiwanie, jego wykres zbliżony jest do funkcji logarytmicznej.

5.2. Lista

Usuwanie i dodawanie na początku oraz na końcu zbliżone są do funkcji stałej O(1) zgodnie z założeniami. Usuwanie oraz dodawanie węzłów w losowym miejscu mają złożoność zbliżoną do O(n). Wynika to z faktu, że jeśli węzeł nie jest dodawany/usuwany na początku albo końcu listy, należy najpierw wyszukać miejsce, w którym ma być dodany/usunięty – przeiterować po wszystkich poprzedzających. Znacząco obciąża to całą procedurę. Sam proces, bez konieczności wyszukiwania miejsca, faktycznie miałby złożoność O(1). Złożoność znajdowania wybranego elementu, podobnie jak w tablicy, przypomina funkcję logarytmiczną.

5.3. Kopiec

Złożoność usuwania oraz znajdowania elementów w zaimplementowanej strukturze mają tendencję logarytmiczną. Wbrew oczekiwaniom, złożoność dodawania elementów do zaimplementowanego kopca jest funkcją stałą - O(1).

Złożoność czasowa generowania każdej z zadanych struktur o określonej wielkości rośnie liniowo w zależności od ilości elementów.

5.4. Podsumowanie

Tablica najlepiej sprawdzi się w sytuacjach, kiedy konieczny jest błyskawiczny dostęp do danego elementu. Lista dwukierunkowa króluje pod względem dodawania początkowych i końcowych elementów, dostęp do poszczególnego elementu jest wolniejszy niż w tablicy, gdyż trzeba przejść przez wszystkie poprzedzające węzły. Kopiec, ze względu na swoją

własność, ustawianie elementów względem wartości, będzie idealnie pasował kiedy elementy danej struktury muszą być stale posortowane.

Bibliografia

Big-O Cheat Sheet. (brak daty). Pobrano z lokalizacji http://bigocheatsheet.com/

Cormen, T. (2001). *Wprowadzenie do algorytmów Wydanie czwarte.* Warszawa: Wydawnictwa Naukowo-Techniczne.