RAPORT

SORTOWANIE CIĄGU UMIESZCZONEGO W KOPCU BINARNYM ORAZ DRZEWIE BST

Valery Kunhurau

Uniwersytet im. Adama Mickiewicza w Poznaniu

Technologie Komputerowe, II rok

2023-2024 rok

A) **Definicja operacji dominujących. Definicja struktury danych (Drewo BST, kopiec binarny)**

- **[2]Złożoność obliczeniową algorytmu** definiuje się jako ilość zasobów komputerowych potrzebnych do jego wykonania. Podstawowymi zasobami rozważanymi w analizie algorytmów są czas działania i ilość wykorzystywanej pamięci.

  Zauważmy, że nie jest na ogół możliwe wyznaczenie złożoności obliczeniowej jako funkcji danych wejściowych (takich jak ciągi, tablice, drzewa czy grafy). Zwykle, co naturalne, z zestawem danych wejściowych jest związany jego **rozmiar**, rozumiany - mówiąc ogólnie - jako liczba pojedynczych danych wchodzących w jego skład.

  W problemie sortowania na przykład za rozmiar przyjmuje się zazwyczaj liczbę elementów w ciągu wejściowym, w problemie przejścia drzewa binarnego - liczbę węzłów w drzewie, a w problemie wyznaczenia wartości wielomianu - stopień wielomianu. Rozmiar zestawu danych d będziemy oznaczać przez $| d |$.

  Aby móc wyznaczać złożoność obliczeniową algorytmu, musimy się jeszcze umówić, w jakich jednostkach będziemy ją liczyć. Na złożoność obliczeniową składa się złożoność pamięciowa i złożoność czasowa. W wypadku **złożoności pamięciowej** za jednostkę przyjmuje się zwykle słowo pamięci maszyny. Sytuacja jest nieco bardziej skomplikowana w wypadku **złożoności czasowej**. Złożoność czasowa powinna być własnością samego tylko algorytmu jako metody rozwiązania problemu - niezależnie od komputera, języka programowania czy sposobu jego zakodowania. W tym celu wyróżnia się w algorytmie charakterystyczne dla niego operacje nazywane **operacjami dominującymi** - takie, że łączna ich liczba jest proporcjonalna do liczby wykonań wszystkich operacji jednostkowych w dowolnej komputerowej realizacji algorytmu.

  Dla algorytmów sortowania na przykład za operację dominującą przyjmuje się zwykle porównanie dwóch elementów w ciągu wejściowym, a czasem też przestawienie elementów w ciągu; dla algorytmów przeglądania drzewa binarnego przyjmuje się przejście dowiązania między węzłami w drzewie, a dla algorytmów wyznaczania wartości wielomianu - operacje arytmetyczne +, -, * i /.

  Za **jednostkę złożoności czasowej** przyjmuje się wykonanie jednej operacji dominującej.

  Złożoność obliczeniową algorytmu traktuje się jako funkcję rozmiaru danych n. Wyróżnia się: **złożoność pesymistyczną** - definiowaną jako ilość zasobów komputerowych potrzebnych przy „najgorszych" danych wejściowych rozmiaru n, oraz **złożoność oczekiwaną** - definiowaną jako ilość zasobów komputerowych potrzebnych przy ,,typowych" danych wejściowych rozmiaru n.

  Aby zdefiniować precyzyjnie pojęcia pesymistycznej i oczekiwanej złożoności czasowej, wprowadzimy następujące oznaczenia:
  D_n – zbiór zestawów danych wejściowych rozmiru n;
  $t(d)$ – liczba operacji dominujących dla zestawu danych wejściowych d;
  X_n – zmienna losowa, którejj wartością jest $d \in D_n$;

p_{nk} – rozkład prawdopodobieństwa zamiennej losowej X_n, tzn. prawdopodobieństwo, że dla danych romiaru n algorytm wykona k operacji dominujących ($k \geq 0$).
Rozkład prawdopodobieństwa zmiennej losowej X_n wyznacza się na podstawie informacji o zastosowaniach rozważanego algorytmu. Gdy na przykład zbiór D_n jest skończony, przyjmuje się często model probabilistyczny, w którym każdy zestaw danych rozmiaru *n* może się pojawić na wejściu do algorytmu z jednakowym prawdopodobieństwem.

Przez **pesymistyczną złożoność czasową algorytmu** rozumie się funkcję

$W(n) = \sup\{t(d) : d \in D_n\}$,

gdzie sup oznacza kres górny zbioru.

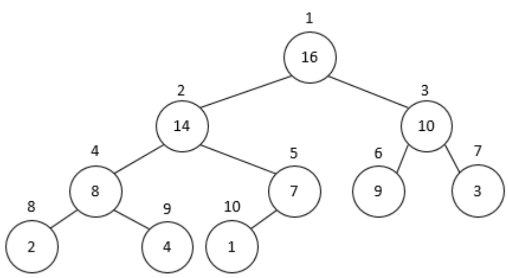
Przez **oczekiwaną złożoność czasową algorytmu** rozumie się funkcję

$A(n) = \sum_{k\geq 0} kp_{nk}$,

tzn. wartość oczekiwaną ave(X_n) zmiennej losowej X_n.

Aby stwierdzić, na ile funkcje $W(n)$ i $A(n)$ są reprezentatywne dla wszystkich danych wejściowych rozmiaru *n*, rozważa się miary wrażliwości algorytmu: **miarę wrażliwości pesymistycznej**, czyli $\Delta(n) = \sup\{t(d_1) - t(d_2) : d_1, d_2 \in D_n\}$, oraz **miarę wrażliwości oczekiwanej**, czyli $\delta(n) = \mathrm{dev}(X_n)$ jest standardowym odchyleniem zmiennej losowej X_n, tzn. $\mathrm{dev}(X_n) = \sqrt{var(X_n)}$ i $var(X_n) = \sum_{k\geq 0}(k - ave(X_n))^2 p_{nk}$ ($var(X_n)$ jest wariancją zmiennej losowej X_n). Im większe są wartości funkcji $\Delta(n)$ i $\delta(n)$, tym algorytm jest bardziej wrażliwy na dane wejściowe i tym bardziej jego zachowanie w przypadku rzeczywistych danych może odbiegać od zachowania opisanego funkcjami $W(n)$ i $A(n)$.

- **[1]Kopiec binarny (binary heap)** jest to tablicowa struktura danych, którą można rozpatrywać jako pełne drzewo binarne, jak to widać na rys. 1. Każdy węzeł drzewa odpowiada elementowi tablicy, w którym jest podana wartość węzła. Drzewo jest pełne na wszystkich poziomach z wyjątkiem być może najniższego, który jest wypełniony od strony lewej do pewnego miejsca. Tablica A reprezentująca kopiec ma dwa atrybuty: *length[A]*, określajęcy liczbę elementów tablicy, i *heap-size[A]*, określający liczbę elementów kopca przechowywanych w tablicy. To znaczy, że żaden element tablicy *A[1..length[A]]* występujący po *A[heap-sizep[A]]*, gdzie *heap-size[A]≤ length[A]*, nie jest elementem kopca. Korzeniem drzewa jest *A[1]*, a mając dany indeks i węzła, można łatwo obliczyć indeksy jego ojca Parent(i), lewego syna Left(i) i prawego syna Right(i).



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

(a) (b)

Rys. 1. Kopiec można rozpatrywać jako (a) drzewo binarne i (b) jako tablicę. Liczba w kółku w każdym węźle drzewa jest wartością przechowywaną w tym węźle. Liczba obok węzła jest odpowiadającym mu indeksem tablicy

Kopce mają również tzw. **własność kopca**: dla każdego węzła i, który nie jest kotzeniem, zachodzi

$$A[Parent(i)] \geq A[i]$$

to znaczy, że wartość w węźle jest nie większa niż wartość przechowywana w jego ojcu. Stąd wynika, że największy element kopca jest umieszczony w korzeniu, a poddrzewa każdego węzła zawierają wartości mniejsze niż wartość przechwywana w tym węźle.

Definiujemy **wysokość** węzła w drzewie jako liczbę krawędzi na najdłuższej prostej ścieżce prowadzącej od tego węzła do liścia, a wysokość drzewa jako wysokość jego korzenia. Ponieważ kopiec mający *n* elementów jest tworzony na podstawie pełnego drzewa binarnego, jego wysokość wynosi *O(lg n)*. Zobaczymy, że podstawowe operacje na kopcach działają w czasie co najwyżej proporcjonalnym do wysokości drzewa, czyli *O(lg n)*. Algorytmy sortowania z użyciem kopca binarnego:

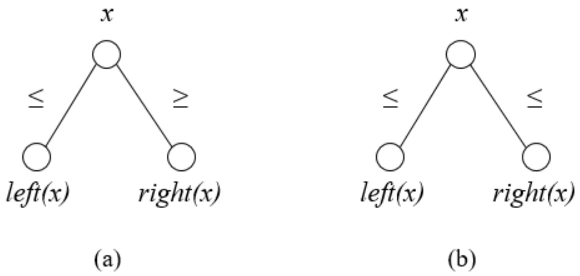
1) Procedura **Heapify**, która działa w czasie *O(lg n)*, służy do przywracania własności kopca.
2) Procedura **Build-Heap**, która działa w czasie *liniowym*, tworzy kopiec z nieuporządkowanej tablicy danych wejściowych.
3) Procedura **Heapsort**, działająca w czasie *O(n lg n)*, sortuje tablicę w miejscu.
4) Procedury **Extract-Max** i **Insert**, które działają w czasie *O(lg n)*, pozwalają na użycie kopca jako kolejki priorytetowej.

- **[2]**Uogólnieniem wyszukiwania elementu w tablicy uporządkowanej jest wyszukiwanie elementu w tzw. **drzewach poszukiwań binarnych, czyli w skrócie BST (od ang. Binary Search Trees)**. Przez wzbogacenie struktury danych uzyskujemy możliwość szybszego wykonywania operacji wstawiania i usuwania elementu ze zbioru.

  Zakładamy, że każdy węzeł x drzewa, będący obiektem typu *node,* ma trzy atrybuty:

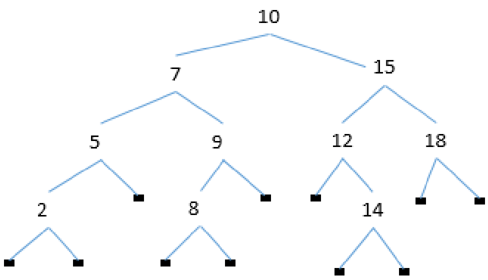
  x: left (x), key (x), right (x)

  Elementy w kopcu są uporządkowane zgodnie z porządkiem kopcowym (rys. 2a). W drzewach BST mamy natomiast do czynienia z porządkiem symetrycznym. W porządku tym dla każdego węzła *x* jest spełniony następujący warunek: jeśli węzeł *y* leży w lewym poddrzewie *x*, to *key (y)* ≤ *key (x)*; jeśli *y* leży w prawym poddrzewie *x*, to *key (x)* ≤ *key (y)* (rys. 2b).



Rys. 2. (a) Porządek kopcowy; (b) porządek symetryczny

Drzewem poszukiwań binarnych (drzewem BST) nazywamy dowolne drzewo binarne, w którym elementy zbioru są wpisane do wierzchołków zgodnie z porządkiem symetrycznym (rys. 3).

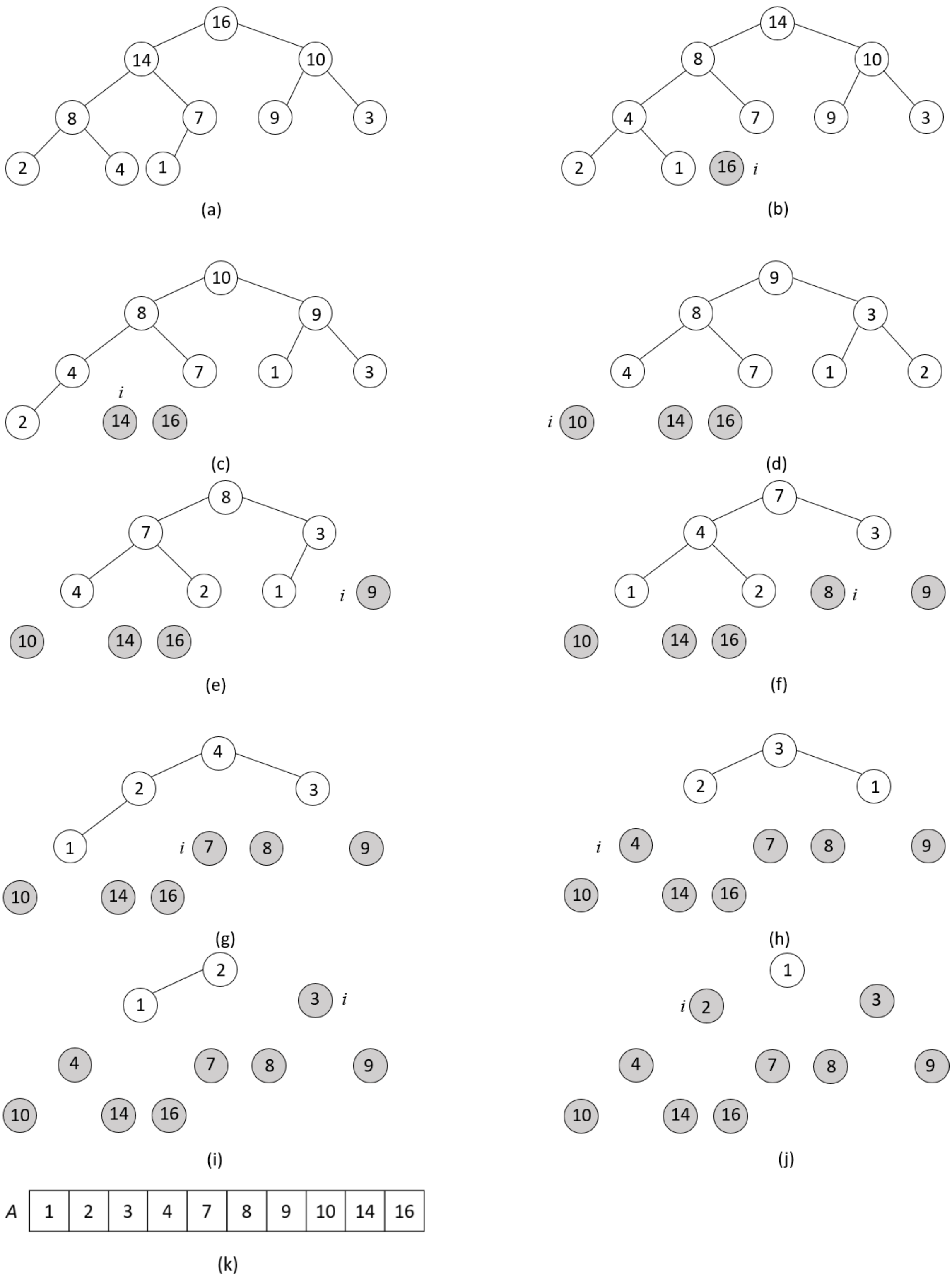


Rys. 3. Drzewo poszukiwań binarnych z zaznaczonymi wierzchołkami zewnętrznymi

Wierzchołki zaznaczone na rysunku 3 przez ▪ to **wierzchołki zewnętrzne**. W strukturze dowiązaniowej są one reprezentowane przez nil. Wierzchołkom zewnętrznym odpowiadają przedziały wartości, na które zostaje podzielona przestrzeń wszystkich kluczy przez klucze obecne w drzewie (czyli elementy znajdujące się w drzewie). Wstawiając nowy element, umieszczamy go w wierzchołku zewnętrznym reprezentującym przedział wartości, do którego należy dany element.

B) Opis algorytma sortowania (Drewo BST, kopiec binarny). Przykład teoretyczny algorytmu sortowania

- **[1]Algorytm sortowania przez kopcowanie (heapsort)**. Algorytm heapsort rozpoczyna działanie, używając procedury Build-Heap do skonstruowania kopca w tablicy wejściowej *A*[1 .. n], gdzie *n* = *length*[*A*]. Skoro największy element tablicy znajduje się w korzeniu *A*[1], może on zostać umieszczony na swoim właściwym miejscu przez zamianę z *A*[n]. Jeśli teraz,,odrzucimy" węzeł *n* z kopca (przez zmniejszenie *heap-size*[*A*]), zauważymy, że tablica *A*[1 .. (*n* − 1)] może łatwo zostać przekształcona w kopiec. Synowie korzenia pozostają kopcami, ale nowy korzeń może naruszać własność kopca (1). Jedyne, co trzeba zrobić, żeby przywrócić własność kopca, to raz wywołać Heapify (*A*, 1), co pozostawi kopiec w *A*[1 .. (*n* − 1)]. Algorytm *heapsort* powtarza ten proces dla kopca o rozmiarze *n* - 1, aż do uzyskania kopca o rozmiarze 2.



Rys. 4. Działanie procedury Heapsort. (a) Struktura kopca zaraz po jego zbudowaniu przez Build-Heap. (b)-(j) Kopiec po każdym wywołaniu Heapify w wierszu 5. Wartość i w tym momencie jest pokazana na rysunku. Tylko jasnoszare węzły pozostaną w kopcu. (k) Posortowana tablica A

Heapsort(A)

1 Build-Heap(A)

2 **for** $i \leftarrow length[A]$ **downto** 2

3 **do** zamień $A[1] \leftrightarrow A[i]$
4 *heap-size*[*A*] ← *heap-size*[*A*] – 1
5 Heapify(A, 1)

Na rysunku 4 widać przykład działania algorytmu heapsort po początkowym zbudowaniu kopca. Każdy kopiec jest pokazany na początku iteracji pętli **for** w wierszu 2.

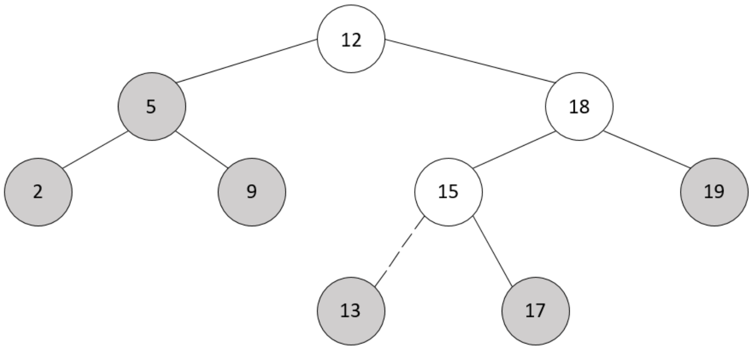
Czas działania procedury Heapsort wynosi $O(n\ lg\ n)$, ponieważ wywołanie Build-Heap zajmuje czas $O(n)$, a każde z n - 1 wywołań Heapify zajmuje czas $O(lg\ n)$.

- [1]Nową wartość *v* można wstawić do drzewa poszukiwań binarnych T za pomocą procedury Tree-Insert. Do procedury przekazujemy jako argument węzeł z, w którym *key*[*z*] = *v*, *left*[*z*] = NIL oraz *right*[*z*] = NIL. W wyniku wykonania procedury drzewo T oraz niektóre pola z są modyfikowane w sposób, który odpowiada wstawieniu *z* we właściwe miejsce w drzewie.

  Tree-Insert(*T*, *z*)

  1 *y* ← NIL
  2 *x* ← *root*[*T*]
  3 **while** *x* ≠ NIL
  4 **do** *y* ← *x*
  5 **if** *key*[*z*] < *key*[*x*]
  6 **then** *x* ← *left*[*x*]
  7 **else** *x* ← *right*[*x*]
  8 *p*[*z*] ← *y*
  9 **if** *y* = NIL
  10 **then** *root*[*T*] ← *z*
  11 **else if** *key*[*z*] < *key*[*y*]
  12 **then** *left*[*y*] ← *z*
  13 **else** *right*[*y*] ← *z*

  Na rysunku 5 jest zilustrowany sposób działania procedury Tree-Insert. Podobnie jak w przypadku procedur Tree-Search i Iterative-Tree-Search, procedura Tree-Insert rozpoczyna przeglądanie w korzeniu, a następnie przebiega po ścieżce w dół drzewa. Wskaźnik *x* przebiega po ścieżce, a zmienna *y* zawiera zawsze wskazanie na ojca *x*. Po zainicjowaniu wartości zmiennych w pętli **while** w wierszach 3-7 wskaźniki *x* i *y* są przesuwane w dół drzewa w lewo lub w prawo, w zależności od wyniku porównania *key*[*z*] z *key*[*x*], aż do chwili, w której zmienna *x* przyjmie wartość NIL. Ta właśnie wartość NIL zajmuje miejsce w drzewie, w którym należy umieścić wskaźnik na węzeł *z*. Wstawienie *z* do drzewa (tzn. wiążące się z tym przypisania właściwych wartości odpowiednim wskaźnikom) odbywa się w wierszach 8-13.



Rys. 5. Wstawienie węzła z kluczem 13 do drzewa poszukiwań binarnych. Jasnoszare węzły wchodzą w skład ścieżki od korzenia do miejsca, w którym węzeł zostaje wstawiony. Przerywaną linią jest oznaczony wskaźnik, który zostaje utworzony w wyniku dodania elementu do drzewa

Procedura Tree-Insert, podobnie jak inne elementarne operacje na drzewach poszukiwań, działa na drzewie o wysokości h w czasie $O(h)$.

C) Sprawdzenie poprawności działania implementacji algorytmów sortowania. Opis w jaki sposób zostały wegenerowane dane do analizy:

- **Funkcja Rand()** jest funkcją wbudowaną w C++ STL, która jest zdefiniowana w pliku nagłówkowym **<cstdlib>** . Rand() służy do generowania serii liczb losowych. Algorytm używany do implementacji funkcji rand() może się różnić w zależności od kompilatora, a co za tym idzie, wyniki również mogą się różnić. Większość implementacji funkcji rand() wykorzystuje metodę **Linear Congruent Method** (w skrócie "LCM"). Funkcja Rand() jest używana w C++ do generowania liczb losowych z zakresu [0, RAND_MAX).

  RAND_MAX: Jest to stała, której wartość domyślna może się różnić w zależności od implementacji, ale może wynosić co najmniej 32767.

  Istotą metody **Linear Congruent Method** (w skrócie "LCM") jest obliczenie ciągu liczb losowych X_n przy założeniu

  $X_{n+1} = (aX_n + c) \; mod \; m,$

  gdzie m – moduł (liczba naturalna, względem której obliczana jest reszta z dzielenia; $m \geq 2$), a – mnożnik ($0 \leq a < m$), X_0 – wartość początkowa ($0 \leq X_0 < m$).

- Poprawność działania implementacji algorytmu sortowania z użyciem kopca binarnego

```
Wprowadź liczbę elementów: 10
Tablica przed algorytmem: 42 68 35 1 70 25 79 59 63 65
Posortowana tablica: 1 25 35 42 59 63 65 68 70 79

C:\Users\Asus\source\repos\ConsoleApplication1\Debug\ ConsoleApplication1.exe
(proces 13188) terminates with code 0.
```

- Poprawność działania implementacji algorytmu sortowania z użyciem drzewa BST

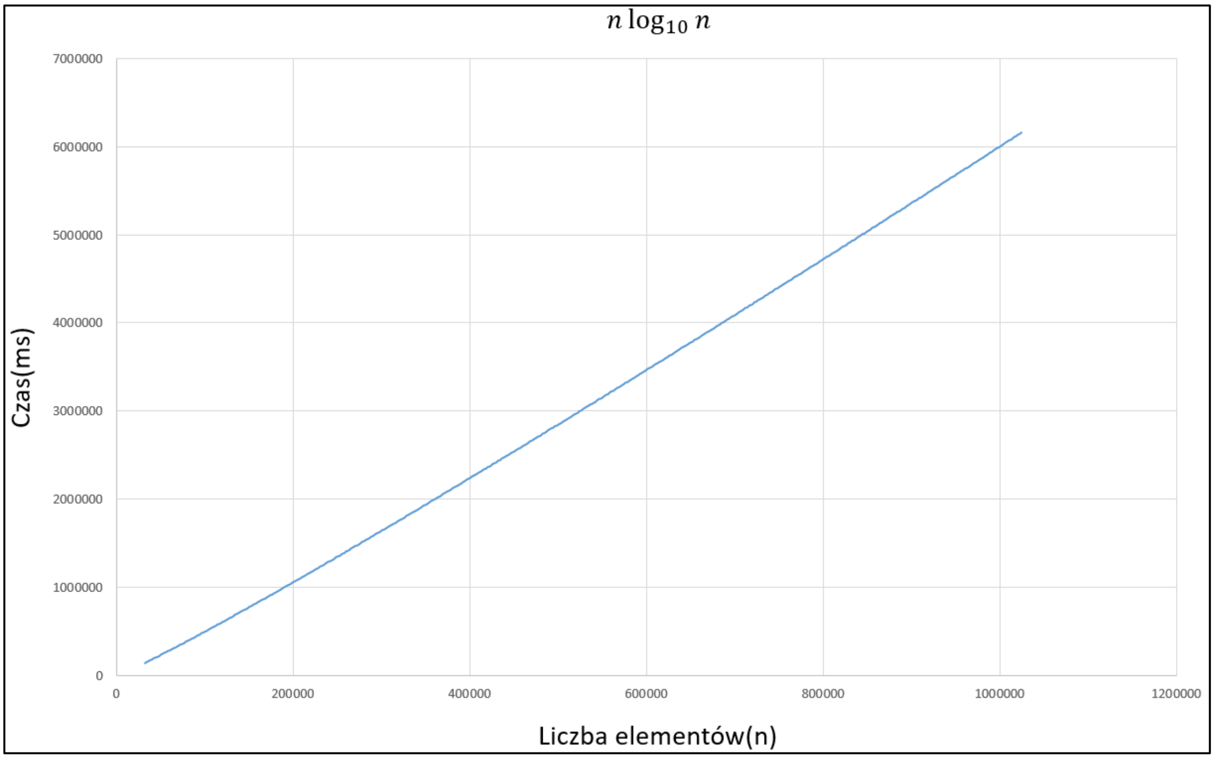
```
Wprowadź liczbę elementów: 10
Tablica przed algorytmem: 97 83 91 68 49 31 41 92 38 6
Posortowana tablica: 6 31 38 41 49 68 83 91 92 97
```

```
C:\Users\Asus\source\repos\ConsoleApplication1\Debug\ ConsoleApplication1.exe
(proces 20476) terminates with code 0.
```

D) Wyniki testów:

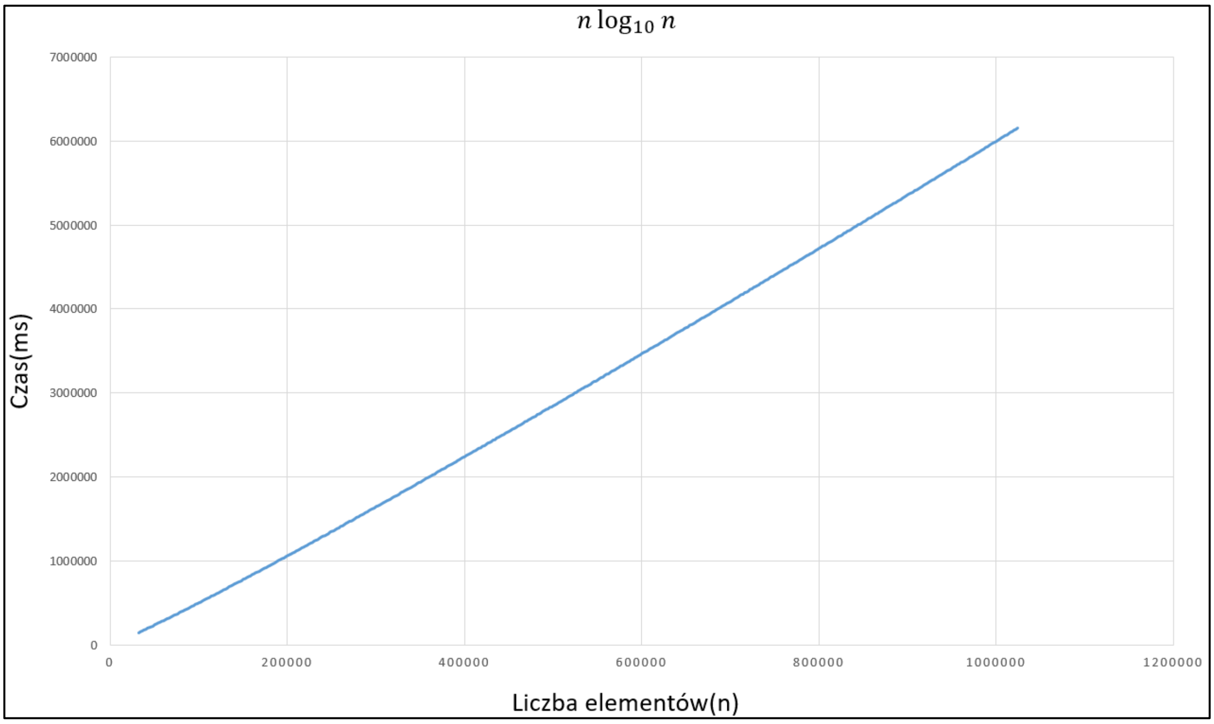
Testy zostały przeprowadzone na komputerze z procesorem Intel® Core^{TM} i5-11300H @ 3.10GHz i obejmowały pomiar czasu sortowania dla zbiorów liczb o liczebności od 32000 do 1024000 elementów w przedziałach od 1 do 1000000. Wyniki testów algorytmu sortowania z użyciem:

- kopca binarnego.



Rys. 6. Wykres złożoności czasowej a liczby elementów algorytmu sortowania z użyciem kopca binarnego

- drzewa BST.



Rys. 7. Wykres złożoności czasowej a liczby elementów algorytmu sortowania z użyciem drzewa BST

E) Wnioski:

- W procesie tym podkreślono następujące kluczowe punkty i wnioski dla algorytmu sortowania z użyciem drzewa BST:

  Wydajność algorytmu:
  Wykorzystanie binarnego drzewa wyszukiwania do sortowania danych wykazuje wysoką wydajność. Złożoność algorytmu wynosi średnio O(n log n), co czyni go atrakcyjnym do przetwarzania dużych ilości danych.

  Zalety i wady:
  Binarne drzewo wyszukiwania ma zalety w postaci łatwości wstawiania i usuwania elementów oraz możliwości obsługi dynamicznej modyfikacji danych. Jednak w najgorszym przypadku złożoność może osiągnąć O(n), co czyni go mniej wydajnym w przypadku wstępnie posortowanych danych.

  Testowanie i stabilność:
  Implementacja została dokładnie przetestowana, w tym na przypadkach z powtarzającymi się i posortowanymi elementami. Wyniki potwierdzają stabilność algorytmu i jego przydatność do obsługi danych wejściowych.

  Wnioski:
  Wykorzystanie binarnego drzewa wyszukiwania do sortowania danych stanowi znaczący krok w kierunku optymalizacji przetwarzania danych. Przy odpowiednim wykorzystaniu i dopracowaniu algorytmu można osiągnąć wysoką wydajność i niezawodność w różnych scenariuszach użytkowania.

- Wnioski z napisania algorytmu sortowania wykorzystującego kopiec binarny:

Wydajność algorytmu:
Praca z kopcem binarnym pozwala osiągnąć wysoką wydajność sortowania. W porównaniu do innych algorytmów, takich jak sortowanie przez scalanie lub szybkie sortowanie, algorytm sortowania z użyciem kopca binarnego zapewnia stabilny czas wykonania w najgorszym przypadku na poziomie O(n log n), co jest ważne w przypadku dużych ilości danych.

Wsparcie dla modyfikowalności danych:
Jedną z zalet kopca binarnego jest możliwość efektywnego dodawania i usuwania elementów podczas procesu sortowania. Jest to szczególnie przydatne, gdy dane dynamicznie się zmieniają lub są przesyłane strumieniowo.

Możliwość zastosowania do różnych typów danych:
Algorytm sortowania z użyciem kopca binarnego jest niezależny od typu danych, co czyni go wszechstronnym i możliwym do zastosowania w różnych scenariuszach. Jest to szczególnie cenne w różnych projektach, w których konieczne jest przetwarzanie danych o różnych formatach.

Stabilność i niezawodność:
Kopiec binarny zapewnia stabilną i niezawodną wydajność sortowania. Algorytm działa poprawnie z różnymi danymi wejściowymi, w tym przypadkami z duplikatami i określonymi rozkładami wartości.

F) Literatura:

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Wprowadzenie do Algorytmów, Wydawnictwo Naukowo-Techniczne, Warszawa 2000

[2] L. Banachowski, K. Diks, W. Rytter, Algorytmy i Struktury Danych, Wydawnictwo Naukowo-Techniczne, Warszawa 1996

[3] Strona wykładowcy, A. Chrobot, na temat implementacji drzewa BST (https://achilles.tu.kielce.pl/portal/Members/84df831b59534bdc88bef09b15e73c99/archive/semestr-ii-2019-2020/pdf/pp2/lecture/pp2_lecture_9.pdf)

[4] Strona wykładowcy I Liceum Ogólnokształcące im. Kazimierza Brodzińskiego w Tarnowie, mgr Jerzy Wałaszek, na temat implementacji kopieca binarnego: Algorytmy i Struktury Danych - Kopiec (Algorytmy i Struktury Danych - Kopiec (eduinf.waw.pl))