Algorytmy i struktury danych – podstawowe informacje

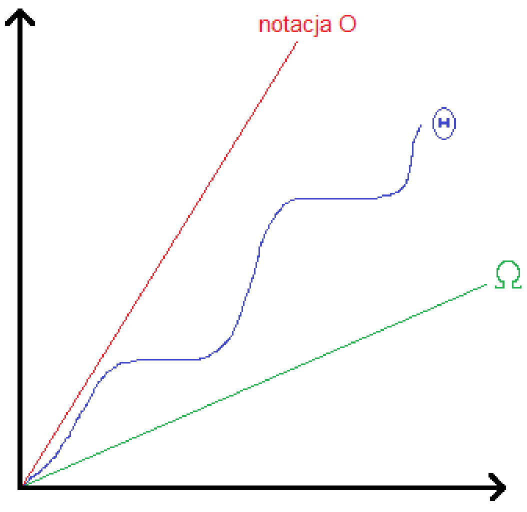
Zawartość

Ważne!

$$\boldsymbol{gddy} <=> <=>$$

Złożoność obliczeniowa

Szacowanie wydajności



Rys. 1 szacowanie wydajności

Wzór do szacowania Θ [teta]

$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

$$n_0 > 0 - ilość\ danych\ \in N\ uint$$

$$c_1, c_2 > 0 - paramtry\ \in R\ float$$

Przykładowe zadanie

Udowodnij używając definicji, że $\frac{1}{2}n^2 - 3n = \Theta(\mathrm{n}^2)$

$$f(n) = \frac{1}{2}n^2 - 3n$$

$$g(n) = n^2$$

$$0 \leq c_1 * n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 * n^2 \ \ | : n^2$$

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

Liczenie ograniczenia górnego:

$$c_2 = \lim_{n \to \infty} \left(\frac{1}{2} - \frac{3}{n}\right) = \frac{1}{2}$$

Liczenie ograniczenia dolnego:

Szukamy najmniejszego naturalnego n, dla którego równanie jest większe od 0.

1) metoda: podstawianie po kolei

$$n_1 = \frac{1}{2} - \frac{3}{1} < 0$$

$$\ldots$$

$$n_7 = \frac{1}{2} - \frac{3}{7} > 0$$

Więc $n_0 = 7$

2) metoda: przyrównanie do zera

$$\frac{1}{2} - \frac{3}{n} = 0$$

$$\frac{1}{2} = \frac{3}{n}$$

$$n = 6$$

Zawsze dodajemy do wyniku 1. Więc $n_0 = 7$

$$c_1 = \frac{1}{2} - \frac{3}{7}$$

$$c_1 = \frac{7}{14} - \frac{6}{14}$$

$$c_1 = \frac{1}{14}$$

Odp.: Udowodniłem, że funkcja f(n) jest ograniczona funkcją $\Theta(n^2)$ dla $c_1 = \frac{1}{14}$ $c_2 = \frac{1}{2}$ $n_0 = 7$

Algorytmy sortowania

Sortowanie bąbelkowe (bubble sort)

Zasada działania

Porównuje sąsiadujące ze sobą elementy, aż cała tablica nie będzie posortowana.

- Złożoność obliczeniowa: $O(n^2)$
- Złożoność pamięciowa: $O(1)$
- Zalety: sortowanie w miejscu, stabilny, łatwy w implementacji
- Wady: powolny

Implementacja:

```
void bubbleSort(int *A, int n)
{
    do
    {
        for(int i = 0; i < n - 1; i++)
        {
            if(A[i] > A[i + 1])
            {
                swap(&A[i], &A[i + 1]);
            }
        }
        --n;
    }
    while(n > 1)
}
```

Rys. 2 sortowanie bąbelkowe

Rozrysowanie

Dla danych 8 4 5 60 22 30 66 50

0) 8 4 5 60 22 30 66 50

1) 4 8 5 22 60 30 66 50

2) 4 5 8 22 60 30 66 50

3) 4 5 8 22 60 30 66 50

4) 4 5 8 22 60 30 66 50

5) 4 5 8 22 30 60 66 50

6) 4 5 8 22 30 60 66 50

7) 4 5 8 22 30 60 50 66 – po pierwszym przejściu przez wszystkie elementy

0) 4 5 8 22 30 60 50 66

1) 4 5 8 22 30 60 50 66

2) 4 5 8 22 30 60 50 66

3) 4 5 8 22 30 60 50 66

4) 4 5 8 22 30 60 50 66

5) 4 5 8 22 30 60 50 66

6) 4 5 8 22 30 50 60 66 – po drugim przejściu (wersja którą mamy się nauczyć w tym miejscu by się nie zatrzymała tylko sprawdziła wszystkie kolejne przestawienia pomimo tego, że już widzimy iż jest posortowany)

Sortowanie przez wstawianie (karciane, insert sort)

Zasada działania

Łap po kolei elementy i przesuwaj je tak długo w lewo aż nie znajdziesz elementu od niej większego (mniejszego).

- Złożoność obliczeniowa: $O(n^2)$
- Złożoność pamięciowa: $O(1)$
- Zalety: stabilny, szybki dla małej ilości danych, szybki dla wstępnie posortowanych danych
- Wady: powolny dla rzeczywistych danych (średniego przypadku)

Implementacja

```
void InsertSort(int *A, int n)
{
    int klucz;
    for(int i = 1; i < n; ++i)
    {
        klucz = A[i];
        int j = i - 1;
        while(j >= 0 && A[j]>klucz)
        {
            A[j + 1] = A[j];
            --j;
            A[j + 1] = klucz;
        }
    }
}
```

Rys. 3 sortowanie przez wstawianie

Rozrysowanie

Dla danych 6 5 3 1 8 7 2 4

0) 6 5 3 1 8 7 2 4

1) 5 6 3 1 8 7 2 4

2) 3 5 6 1 8 7 2 4

3) 1 3 5 6 8 7 2 4

4) 1 3 5 6 8 7 2 4

5) 1 3 5 6 7 8 2 4

6) 1 2 3 5 6 7 8 4

7) 1 2 3 4 5 6 7 8

Sortowanie przez wybieranie (selection sort)

Zasada działania

Znajdź najmniejszy element i zamień go z tym na początku (ważne zamień a nie przestawiaj), a potem wyszukuj od kolejnego elementu który jest traktowany jako pierwszy. Często mylony z sortowaniem przez wstawianie można sobie skojarzyć, że wybieramy sobie najmniejszy lub największy element żeby go przestawić na początek.

- Złożoność obliczeniowa: $O(n^2)$
- Złożoność pamięciowa: $O(1)$
- Zalety: sortowanie w miejscu, łatwy w implementacji
- Wady: niestabilny, powolny

Implementacja

```
void SelectionSort(int n, int *A)
{
    int key;
    for(int i = 0; i < n; i++)
    {
        key = i;
        for(int j = i + 1; j<n; j++) //miniumum
        {
            if(A[j] < A[key]) key = j;
        }
        swap(&A[key], &A[i]);
    }
}
```

Rys. 4 sortowanie przez wybieranie

Sortowanie przez zliczanie (counting sort)

Zasada działania

Zapisuje do dodatkowej tablicy ilość wystąpień poszczególnych wartości. Np. pod indeksem 2 jest ilość wystąpień liczby 2.

- Złożoność obliczeniowa: $O(n+k)$
- Złożoność pamięciowa: $O(n+k)$
  gdzie k – rozpiętość danych (różnica pomiędzy najmniejszą a największą wartością danych)
- Zalety: szybki, dobry dla danych z dużą ilością powtórzeń, stabilny
- Wady: wykorzystywana dodatkowa pamięć (nie jest w miejscu), słaby dla zróżnicowanych danych

Implementacja

```
void CountingSort(int *A, int *B, int n)
{
    int k; //ilość różnych elementów
    int *temp; //tablica z ilością elementów o danej wartości

    for(int i = 0; i < k; ++i) temp[i] = 0; //zerowanie tablicy
    for(int i = 0; i < n; ++i) ++temp[A[i]]; //zliczanie
    for(int i = 1; i < k; ++i) temp[i] += temp[i - 1]; //zapisanie pozycji
    for(int i = n - 1; i >= 0; --i)B[--temp[A[i]]] = A[i];
}
```

Rys. 5 sortowanie przez zliczanie

UWAGA: powyższa implementacja nie zawiera funkcji alokujących pamięć.

Sortowanie przez scalanie (merge sort)

Zasada działania

Dzieli tablice na pół tak długo aż nie zostanie jeden element i potem łączy od dołu sortując poszczególne podtablice używając funkcji do scalania posortowanych ciągów.

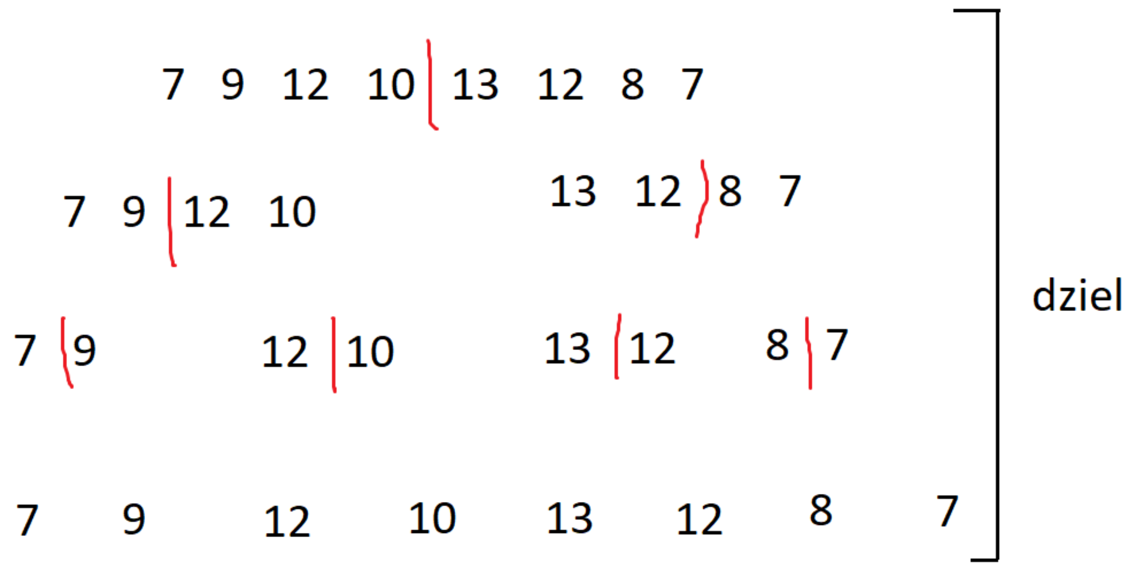
- Złożoność obliczeniowa: $O(n * log\ n)$
- Złożoność pamięciowa: $O(n)$
- Zalety: stabilny, prosty, szybki także w praktyce
- Wady: potrzeba dodatkowej pamięci

Implementacja

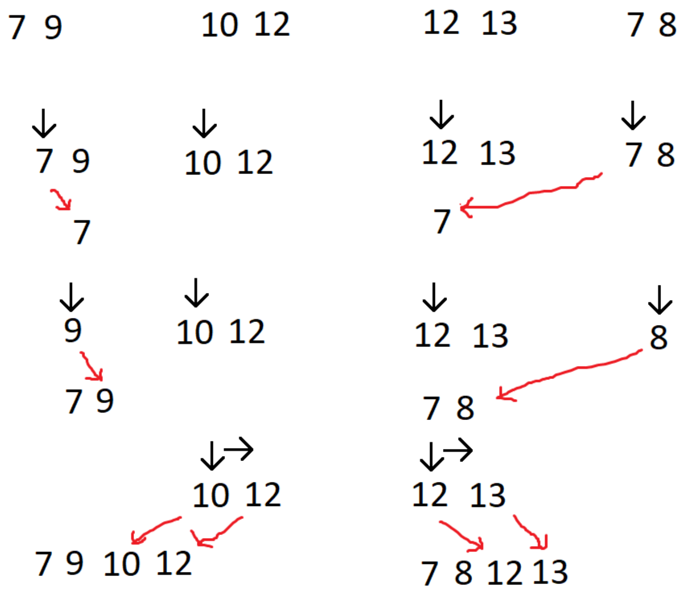
```
void mergesort(int pocz, int kon)
{
    int sr;
    if(pocz < kon)
    {
        sr = (pocz + kon) / 2;
        mergesort(pocz, sr); // Dzielenie lewej czesci
        mergesort(sr + 1, kon); // Dzielenie prawej czesci
        merge(pocz, sr, kon); /* laczenie czesci lewej i prawej*/
    }
}
```

Rys. 6 sortowanie przez łączenie

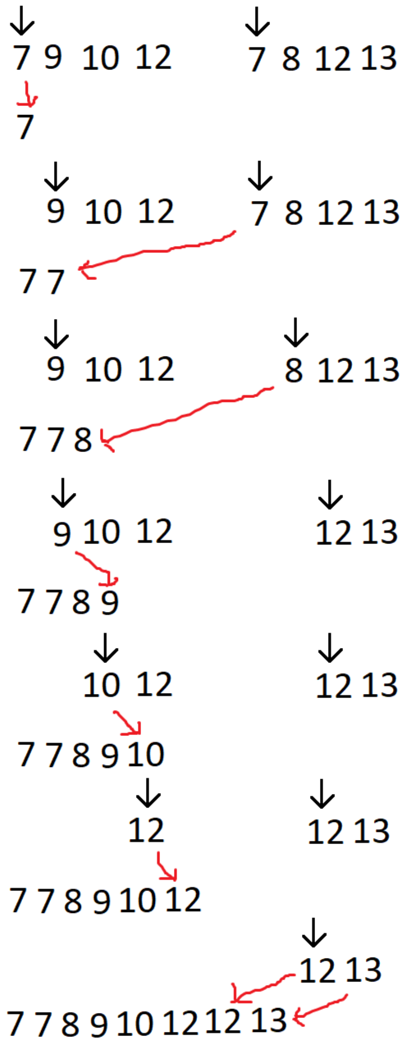
Rozrysowanie



Rys. 7 faza dzielenia w algorytmie sortowania przez scalanie



Rys. 8 faza łączenia w algorytmie przez scalanie



Rys. 9 faza łączenia w algorytmie przez scalanie ostatnie łączenie

Szybkie sortowanie (quick sort)

Zasada działania

Wybieramy arbitralnie jakiś punkt i przerzucamy na lewo mniejsze, a na prawo większe bądź równe od danego elementu. Potem wchodzimy do każdej z podzielonych części i powtarzamy aż nie zostanie tylko jeden element.

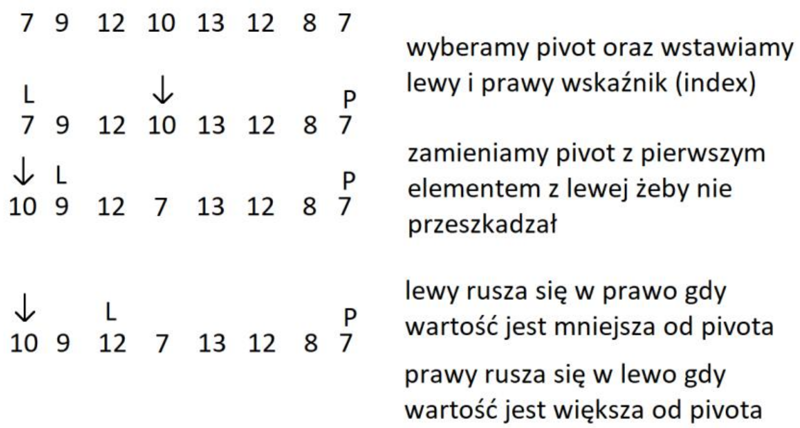
- Złożoność obliczeniowa: $O(n * log\ n)$
- Złożoność pamięciowa: $O(1)$
- Zalety: szybki także w praktyce, sortuje w miejscu
- Wady: w pesymistycznych sytuacjach złożoność obliczeniowa $\Theta(n^2)$, niestabilny

Implementacja

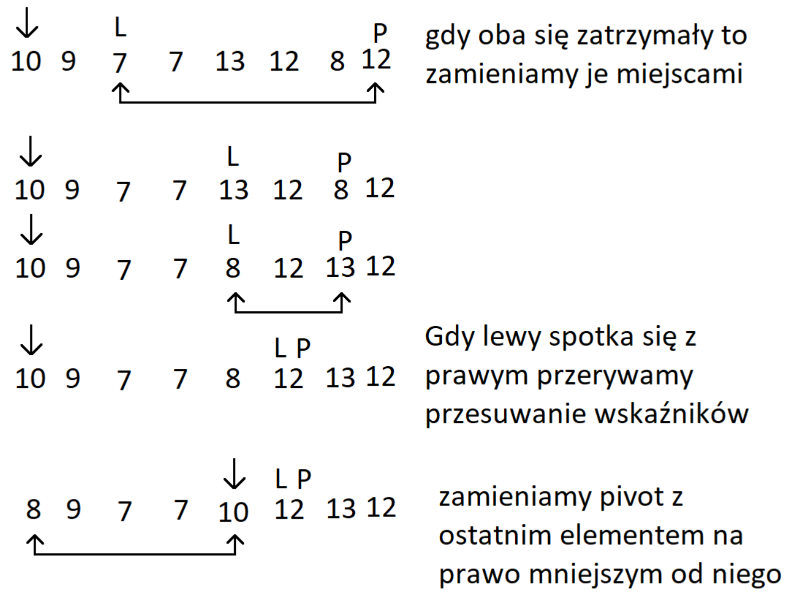
```
void quicksort(int tablica[], int p, int r)
{
    int q;
    if(p < r)
    {
        q = partition(tablica, p, r);
        //dzielimy tablice na dwie czesci, q oznacza punkt podzialu
        quicksort(tablica, p, q);
        //wywolujemy rekurencyjnie quicksort dla pierwszej czesci tablicy
        quicksort(tablica, q + 1, r);
        // wywolujemy rekurencyjnie quicksort dla drugiej czesci tablicy
    }
}
```

Rys. 10 implementacja sortowania szybkiego

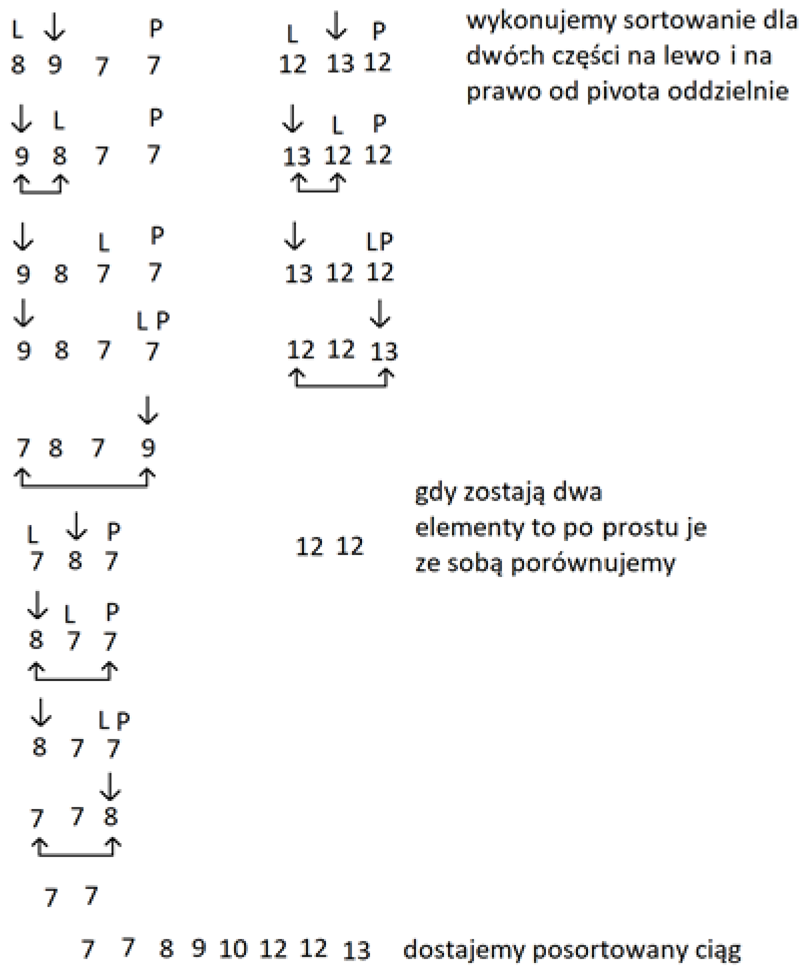
Rozrysowanie



Rys. 11 sortowanie szybki graficzne rozwiązanie cz1



Rys. 12 sortowanie szybki graficzne rozwiązanie cz2



Rys. 13 sortowanie szybki graficzne rozwiązanie cz3

Podsumowanie

Tabela złożoności (szeregowanie algorytmów)

Nazwa	optymistyczna	typowa	pesymistyczna	stabilność	w miejscu	Pamięć
Bąbelkowe	$O(n^2)$	$O(n^2)$	$O(n^2)$	TAK	TAK	O(1)
Wstawianie	O(n)	$O(n^2)$	$O(n^2)$	TAK	TAK	O(1)
Wybieranie	$O(n^2)$	$O(n^2)$	$O(n^2)$	NIE	TAK	O(1)
Zliczanie	O(k + n)	O(k + n)	O(k + n)	TAK	NIE	O(n + k)
Scalanie	O(n*log n)	O(n*log n)	O(n*log n)	TAK	NIE	O(n)
Szybkie	O(n*log n)	O(n*log n)	$O(n^2)$	NIE	TAK	O(1)

Algorytmy stabilne

- bąbelkowe
- przez wstawianie
- przez scalanie
- przez zliczanie
- kubełkowe
- pozycyjne

Algorytmy niestabilne

- przez wybieranie
- Shella
- szybkie
- przez kopcowanie

Definicje

Sortowanie w miejscu

Algorytm nie potrzebuje dodatkowej pamięci na obliczenia. Ewentualnie niewielką stałą niezależną od ilości danych.

Sortowanie stabilne

Gdy wśród elementów do sortowania występują takie, które się powtarzają to po sortowaniu jest zachowana ich pierwotna kolejność. Czyli ta która wystąpiła jako pierwsza nadal będzie pierwsza.

Złożoności obliczeniowe

Każdy kolejny algorytm jest słabszy od poprzedniego. Należy czytać od góry do dołu.

- 1
- log log n
- $\sqrt{\log n}$
- $\frac{\log n}{\log\log n}$
- log n
- $log^2 n$
- $log^3 n$
- $n^{0,01}$
- $n^{0,1}$
- $n^{0,1}\log n$
- $\frac{n}{logn}$
- $\frac{n}{\log\log n}$
- n
- n log log n
- n log n
- $n^{\frac{3}{2}}$
- $\frac{n^2}{\log n}$
- n^2
- n^3
- $n^{\log n}$
- 2^n
- $16^{\frac{n}{2}}$
- 10^n
- $n!$
- n^n
- $n^{n!}$

Stos

Stos jest interfejsem, w którym mamy określone funkcję. Możemy podglądać tylko element, który położyliśmy, jako ostatni.

Operacje:

- push() – dokładamy nowy element do stosu.
- pop() - która pozwala na zdjęcie i odczytanie ostatnio położonego elementu.
- isEmpty() – sprawdza czy stos jest pusty
- pick() – pobranie wartości bez ściągania jej ze stosu. (opcjonalnie)
- size() – zwraca rozmiar stosu. (opcjonalnie)

Nie mylić stosu z listą czy kolejką. Stos może być zrealizowany na dowolnej strukturze danych (tablica, tablica dynamiczna, lista dwukierunkowa) i może być dla dowolnego typu (int, float char).

Odwrotna notacja polska na przykładach

1) AB*CD*E/-

(A*B)-((C*D)/E)

2) 6 5*3 7+8 4-2/*+

Gdy natrafiamy na liczbę dokładamy ją do stosu, gdy na trafimy na działanie ściągamy dwie liczby i wykonujemy na nich działanie, a wynik wrzucamy na stos.

Lp.	wejście	stos
1	6	6
2	5	6 5
3	*	30
4	3	30 3
5	7	30 3 7
6	+	30 10
7	8	30 10 8
8	4	30 10 8 4
9	-	30 10 4
10	2	30 10 4 2
11	/	30 10 2
12	*	30 20
13	+	50

(6 * 5) + ((3 + 7) * ((8 – 4) / 2)) = 50

Kolejka

Kolejka podobnie jak stos nie jest strukturą danych tylko interfejsem. Z kolejki możemy, jeśli jest jednostronna to zabierać i odczytywać element, który został położony, jako pierwszy, a dokładamy elementy na sam koniec. W dwustronnej możemy zabierać elementy jeszcze z góry, ale nie ze środka.

Operacje:

- Init(q) – opróżnia kolejkę i przygotowuje pamięć
- isEmpty(q) – sprawdza czy kolejka jest pusta
- Enqueue(q, x) – dokłada element **x** na koniec kolejki **q**
- Dequeue(q) – zabiera element z początku kolejki

Haszowanie

Haszowanie liniowe i obsługa kolizji

Mamy tablicę 8-elementową:

13	0	7	6	8	10	12	5

Zakładamy że funkcja haszująca to $f(n) = (2n + 1)\,\%\,10$. Zwraca ona wartości od 0 do 9, więc tworzymy 10-elementową tablicę haszującą. Zwrócona wartość to nr docelowego indeksu w tablicy.

$$f(13) = (2 \cdot 13 + 1)\,\%\,10 = 27\,\%\,10 = 7$$

							13		
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

$$f(0) = (2 \cdot 0 + 1)\,\%\,10 = 1\,\%\,10 = 1$$

	0						**13**		
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

$$f(7) = 15\,\%\,10 = 5$$

	0				**7**		**13**		
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

$$f(6) = 13\,\%\,10 = 3$$

	0		**6**		**7**		**13**		
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

$$f(8) = 17\,\%\,10 = 7$$

Występuje konflikt, więc przesuwamy się w prawo tak długo aż znajdziemy wolne miejsce.

	0		**6**		**7**		**13**	**8**	
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

$$f(10) = 21\,\%\,10 = 1$$

Występuje konflikt, więc przesuwamy się w prawo tak długo aż znajdziemy wolne miejsce.

	0	**10**	**6**		**7**		**13**	**8**	
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

$$f(12) = 25\,\%\,10 = 5$$

Występuje konflikt, więc przesuwamy się w prawo tak długo aż znajdziemy wolne miejsce.

	0	**10**	**6**		**7**	**12**	**13**	**8**	
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

$$f(5) = 11 \ \% \ 10 = 1$$

Występuje konflikt, więc przesuwamy się w prawo tak długo aż znajdziemy wolne miejsce.

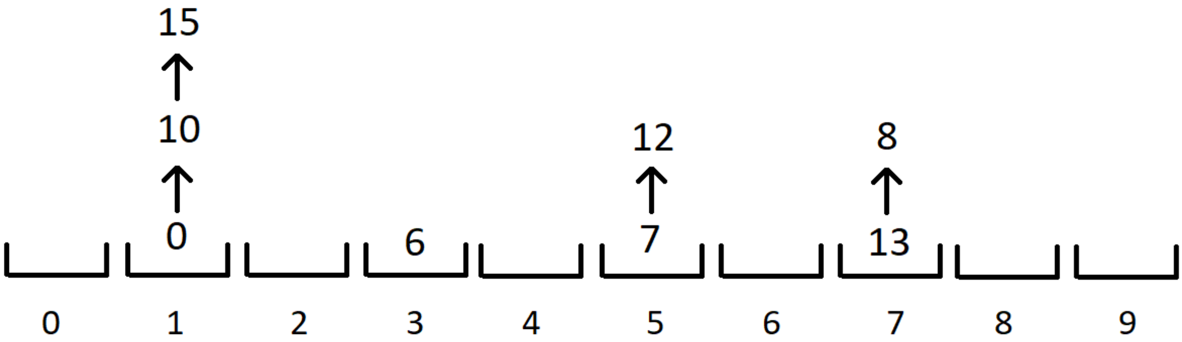
	0	**10**	**6**	**5**	**7**	**12**	**13**	**8**	
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Haszowanie łańcuchowe

Dla ułatwienia i pokazania różnic użyjemy tablicy z poprzedniego zadania

13	0	7	6	8	10	12	15
7	1	5	3	7	1	5	1

Drugi rząd to już obliczone pozycje do tablicy za pomocą tego samego wzoru co w poprzednim przykładzie $f(n) = (2n + 1)\ \%\ 10$



Haszowanie łańcuchowe zamiast nie jest tablicą przechowującą dane wartości tylko tablicą list.

Te które były jako pierwsze są niżej na liście. Nie ma różnicy czy narysujemy to pionowo (tak jak tu) czy poziomo (jak było na prezentacji).

Haszowanie kwadratowe

Jeszcze raz rozważmy te same dane dla tej samej funkcji haszującej $f(n) = (2n + 1)\ \%\ 10$.

13	0	7	6	8	10	12	15
7	1	5	3	7	1	5	1

Rozważmy liczby 0, 10 oraz 15 ponieważ mają kolizje na pozycji 1.

Zastąpmy naszą liczbę 1 w funkcji przez i: $f(n) = (2n + i)\ \%\ 10$

Przy każdej kolizji używamy następującego wzoru $f(n) = (2n + i^2)\ \%\ 10$ poczym wykonujemy ++i

Każda następna kolizja jest przesuwana o kwadrat danej liczby, a nie o jeden jak w liniowym

$$(2n + 1^2)\ \%\ 10$$

$$(2n + 2^2)\ \%\ 10$$

$$(2n + 3^2)\ \%\ 10$$

	0		**6**		**7**		**13**		

0	1	2	3	4	5	6	7	8	9

Pierwsza kolizja to 8 bo chcemy ją postawić na indeks 7 który jest zajęty przez liczbę 10.

$$(2 * 8 \ + \ 2^2)\% \ 10 = (16 + 4)\%10 = 0$$

8	**0**		**6**		**7**		**13**		
0	1	2	3	4	5	6	7	8	9

Druga kolizja to 10.

$$(2 * 10 \ + \ 2^2)\% \ 10 = (20 + 4)\%10 = 4$$

8	**0**		**6**	**10**	**7**		**13**		
0	1	2	3	4	5	6	7	8	9

Trzecia kolizja to 12.

$$(2 * 12 \ + \ 2^2)\% \ 10 = (24 + 4)\%10 = 8$$

8	0		6	10	7		13	12	
0	1	2	3	4	5	6	7	8	9

Czwarta kolizja to 15.

$$(2 * 15 \ + \ 2^2)\% \ 10 = (30 + 4)\%10 = 4$$

Tu następuje kolejna kolizja więc dodajemy do i jeszcze 1.

$$(2 * 15 \ + \ 3^2)\% \ 10 = (30 + 9)\%10 = 9$$

8	**0**		**6**	**10**	**7**		**13**	**12**	**15**
0	1	2	3	4	5	6	7	8	9

Haszowanie podwójne

Mamy dwie funkcje haszujące h(k) oraz h’(k)

Używamy pierwszej funkcji haszującej, a w przypadku kolizji wykorzystujemy wzór:

$$h(k) + i * h'(k)$$

Gdzie **i** to ilość kolizji.

Haszowanie kukułcze

Mamy dwie tablice równej wielkości oraz oddzielną funkcję haszującą do każdej

Jak wygląda wrzucanie elementów do tablicy: (za każdym razem używamy odpowiedniej funkcji haszującej)

1) Sprawdzamy czy mamy wolne miejsce w pierwszej tablicy
2) Jeżeli nie, to wstawiamy do drugiej tablicy
3) Jeśli w drugiej też jest zajęte to, to co było na miejscu, które chce zająć wrzucany element to go zabieramy i próbujemy wstawić do pierwszej
4) Jeśli w pierwszej jest zajęte to wstawiamy element na jego miejsce na pałę, a ten wypchany element próbujemy wrzucić do drugiej
5) Jeśli wystąpi jakiś cykl czy blokada to masz pecha chyba, że zrobisz jakiś limit przerzuceń to wtedy np. może ci coś nie wejść do którejś z tablic

Pojęcia związane z haszowaniem

- O(1) wyszukiwanie w optymistycznym przypadku
- O(n/m + 1) wyszukiwanie w średnim przypadku
- **n** – ilość elementów w ciągu **m** – długość haszowanej tablicy
- O(n) wyszukiwanie w pesymistycznym przypadku
- $\alpha = \frac{n}{m}$ współczynnik przeładowania (load factor)
- $\frac{1}{2}(1 + 1/(1-\alpha)^2)$ oczekiwana liczba prób do znalezienia oczekiwanej wartości

Drzewo

- Drzewo (wolne): graf spójny acykliczny nieskierowany. Jeśli graf jest nieskierowany i acykliczny, ale niekoniecznie spójny, to nazywamy go lasem.
- Drzewo z korzeniem (drzewo ukorzenione): drzewo wolne z jednym wyróżnionym wierzchołkiem, tzw. korzeniem (oznaczamy go zwykle przez *r*).
- Drzewo uporządkowane: drzewo ukorzenione, w którym dzieci każdego węzła są uporządkowane.
- Regularne drzewo binarne: każdy węzeł ma 0 lub 2 dzieci.
- Pełne drzewo binarne: wszystkie liście są na tej samej głębokości *h*(dla pewnego h), a wszystkie węzły wewnętrzne mają stopień 2.(Innymi słowy, pełne drzewo binarne jest regularne.)

Przeszukiwanie

INORDER	**PREORDER**	**POSTORDER**
• Lewe dziecko • Węzeł • Prawe dziecko	• Węzeł • Lewe dziecko • Prawe dziecko	• Lewe dziecko • Prawe dziecko • Węzeł

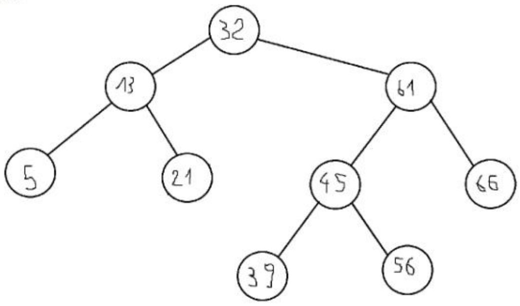
Podstawowe pojęcia

- **Liść -** Węzeł drzewa, z którego nie wychodzi żadna krawędź. Nie ma dzieci
- **Korzeń -** Węzeł początkowy drzewa znajdujący się, jako jedyny na wysokości 0 i nie ma rodzica
- **Bracia –** Węzły wychodzące z tego samego rodzica

- **Wysokość -** Odległość między korzeniem, a najodleglejszym liściem. Do wysokości drzewa nie wlicza się korzenia (jest na głębokości 0).
- **Głębokość -** Odległość miedzy korzeniem, a konkretnym wierzchołkiem
- **Drzewo binarne -** Drzewo, w którym stopień każdego wierzchołka jest nie większy od 3.
- **Drzewo BST (przeszukiwań binarnych) -** To takie drzewo, w którym prawe dzieci są większe od rodzica, a lewe mniejsze od niego.

Następnik węzła o jakimś kluczu to następny węzeł odwiedzony w kolejności InOrder
Następnik można wyznaczyć nie wykonujac porównań.

następnik(5) = 13
następnik(13) = 21
następnik(21) = 32
następnik(32) = 39
następnik(39) = 45
następnik(45) = 56
następnik(56) = 61
następnik(61) = 66
następnik(66) = NULL



Minimum w drzewie BST znajdujemy odwiedzając lewych synów aż napotkamy wartość NULL (dojdziemy do liścia maksymalnie po lewo.

Szukanie rozpoczynamy zaczynając w korzeniu!

Pseudokod:

```
MINIMUM (struct tree* tree)
  while (tree->Left != NULL)
  {
    tree = tree->Left;
  }
  return tree;
```

Analogicznie jest ze znajdywaniem Maksimum z tą różnicą, że idziemy w prawo

```
MAKSIMUM (struct tree* tree)
  while (tree->Right != NULL)
  {
    tree = tree->Right;
  }
  return tree;
```

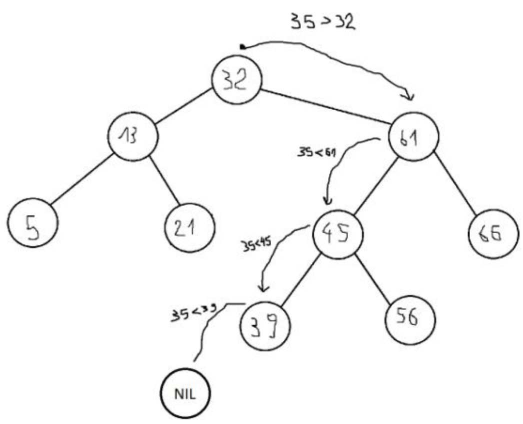
Dodawanie nowego węzła o podanym przez nas kluczu:

Schemat:

1. Utwórz element i przypisz mu wartość klucza.
2. Szukaj miejsca w drzewie:
   a) Zaczynamy w korzeniu i porównujemy wartość klucza z kluczem w węźle w którym aktualnie się znajdujemy.
   b) Jeśli klucz w dodawanym węźle ma wartość większą, poruszamy się w prawo, jeśli mniejszą - w lewo.
   c) Powtarzamy kroki 'a' i 'b' aż dojdziemy do liścia (wskaźnik na potomka ma wartość NULL)
3. Ustaw wartość wskaźnika który wywołał zatrzymanie przeszukiwania tak, aby wskazywał na nowo dodany element.

DODAJ(drzewo, 35)

punkt 1. ze schematu 



punkt 3: ustaw wartośc wskaźnika
drzewo->Lewy = nowy_element

Kopce

Kopiec jest strukturą opartą na drzewie.

Wszystkie liście są umieszczone maksymalnie z lewej strony drzewa, a różnica między poziomami liści nie jest większa niż 1. Liście w tablicy są umieszczone na końcu. (Dla każdego i > floor(n/2))

Cechuje się tym że zachowana jest relacja między węzłem i jego dziećmi (zwykle wartość węzła jest większa/mniejsza niż wartości obu jego dzieci). Dzięki temu dobrze nadaje się do implementacji kolejki priorytetowej.

Wiemy, że element największy (najmniejszy) jest zawsze na górze kopca, więc zdejmowanie kolejnych elementów z góry kopca zawsze da ciąg posortowany. Jest to heapsort.

Tworzenie kopca

Kopiec jest strukturą bezwskaźnikową. Oznacza to że informacja o relacjach między elementami jest zawarta w kolejności tych elementów w tablicy.

(indeksy liczymy od 1) (floor: zaokrąglenie w dół) (H: tablica zawierająca kopiec)

funkcje ustalające pozycje elementów związanych z elementem na pozycji i

parent

    return floor(i/2)

left

    return 2*i

right

    return 2*i + 1

Funkcja heapify.

Służy do naprawy własności kopca względem jednego elementu. Po heapify zostaje utworzony podkopiec z rootem w węźle i.

```
heapify(H, a)
{
    largest = a;

    if(2 * a <= size && H[2 * a] > H[largest])
    {
        largest = 2 * a;
    }
    if(2 * a + 1 <= size && H[2 * a + 1] > H[largest])
    {
        largest = 2 * a + 1;
    }
    if(largest != a)
    {
        swap(H[a], H[largest]);
        heapify(H, largest);
    }
}
```

Funkcja build heap

Służy do przemieszczenia elementów w tablicy w taki sposób, aby stanowiła ona kopiec. Z własności kopca polegającej na zachowaniu relacji między ojcem i synami wynika wniosek że każde poddrzewo kopca jest kopcem. Należy więc wykonać funkcję heapify dla każdego elementu który ma dzieci (nie jest liściem)

```
build_heap(H)
{
    int i;
    for(i = heapSize(H)/2; i >= 1; --i)
    {
        heapify(H, i);
    }
}
```

Heapsort

Skoro wiemy że największy(najmniejszy) element jest na samej górze kopca można wykorzystać ten fakt do sortowania.

Procedura heapsort

1. zamień element ostatni (najbardziej prawy liść) z pierwszym (korzeń)
2. zmniejsz rozmiar kopca o jeden. (końcówka tablicy nie należy teraz do kopca i jest posortowanym ciągiem)

3. napraw kopiec (build heap)
4. jeśli rozmiar kopca jest większy niż 0 wróć do kroku 1

```
heapsort(H)
{
    while(heapSize(H) > 0)
    {
        swap(H[0], H[heapSize(H)]);
        heapSize(H)--;
        build_heap(H);
    }
}
```

Dodawanie elementu

1. zwiększ rozmiar kopca o 1
2. Dodaj element na koniec tablicy
3. napraw kopiec

```
addElement(H, el)
{
    ++heapSize(H);
    H[heapSize] = el;
    build_heap(H);
}
```

Usuwanie elementu na określonym indeksie

Analogicznie do dodawania elementu

1. zamień element usuwany z ostatnim
2. zmniejsz rozmiar kopca o jeden
3. napraw kopiec

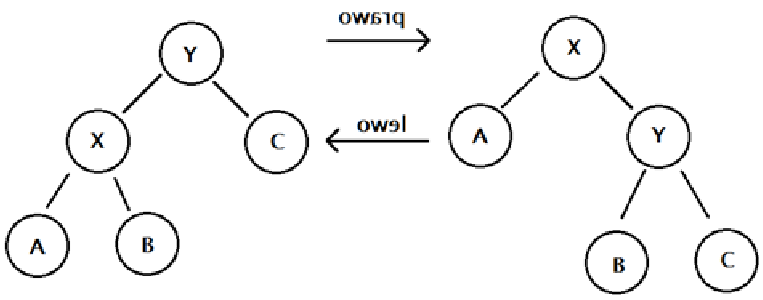
```
deleteElement(H, i)
{
    swap(H[i], H[heapSize(H)]);
    --heapSize(H);
    build_heap(H);
}
```

Drzepiec aka drzewiec

Jednostki 5 TR’u w bastionie są bezwartościowe, powolne i potrafią unieruchomić przeciwników, ale wychodzi im to tylko wtedy, gdy przeciwnik stoi przy twoich strzelcach. Niektórzy stosują do obrony bramy zamku, ale zanim do niej dotrą w murze jest już inna dziura. Ogólnie nie polecam.

W informatyce jest to struktura łącząca cechy drzewa i kopca. Wierzchołek oprócz samej wartości ma również wagi. Klucze zachowują się jak drzewo przeszukiwań binarnych zaś wagi jak kopiec.

Rotacje



Rotacje są robione względem wagi

Dodawanie do drzepca

Krok1: Dodajemy element dokonując porównań według klucza tak jak w drzewie BST

Krok2: Dokonujemy rotacji tak, by wartości wag rodzica były zawsze większe od wag dzieci

Grafy

Graf skierowany (zorientowane)

To para G = (V, E) zbiorów skończonych gdzie V to zbiór węzłów (wierzchołków), a E to zbiór uporządkowanych par wierzchołków (krawędzi, łuków, pętli).

Graf nieskierowaniy (niezorientowane)

To para G = (V, E) zbiorów skończonych gdzie V to zbiór węzłów (wierzchołków), a E to zbiór dwuelementowych zbiorów wierzchołków (krawędzi, łuków, pętli).

Macierze sąsiedztwa

Dwuwymiarowa (V^2), jest jedna dla całego grafu. Wypełniona 0 (brak połączenia) lub 1 (połączenie).

- Dla grafów skierowanych: diagonalna nie musi być z samych 0.
- Dla grafów nieskierowanych: diagonalna jest z samych 0, oraz jest symetryczna względem diagonalnej.

Diagonalna to ta przekątna od góry po lewo do dołu po prawo.

Listy sąsiedztwa

Jednowymiarowa (V), przechowuje informację o tym, do których wierzchołków prowadzi droga z danego wierzchołka. Jest listą wskaźników.

Macierze incydencji

Dwuwymiarowa (V*E) prostokątna, niesymetryczna i przyjmuje wartości -1, 0, 1 oraz 2.

-1 oznacza krawędź w grafie skierowanym w przeciwnym kierunku do wierzchołka (wychodzi z niego), 0 to brak połączenia, 1 to krawędź wchodząca do danego wierzchołka, a 2 to pętla.

Przeszukiwanie BFS (w szerz)

- wrzucamy wierzchołek startowy do kolejki
- sprawdzamy wszystkich sąsiadów i wrzucamy do kolejki każdego, który nie został jeszcze odwiedzony.
- po dorzuceniu wierzchołków wierzchołka na dole kolejki wyrzucamy go

Lub

- Na początku wszystkie wierzchołki są białe.
- Wierzchołki szare są przechowywane w kolejce FIFO.
- Po jej opuszczeniu kolorujemy je na czarno.

Przeszukiwanie DFS (w głąb)

- badamy wszystkie krawędzie ostatnio odwiedzonego wierzchołka v
- gdy wszystkie krawędzie v są zbadane przechodzimy wierzchołka v, z którego przeszliśmy do obecnego
- kontynuujemy do momentu, gdy wszystkie wierzchołki osiągalne z wierzchołka startowego zostaną odwiedzone

Złożoność pamięciowa grafu

Złożoność pamięciowa grafu jest $\leq V^2$

- **Graf spójny -** gdy do każdego wierzchołka prowadzi krawędź i da się dojść od niego do każdego innego. Graf spójny może być słabo lub silnie spójny.
- **Graf słabo spójny -** gdy do każdego wierzchołka prowadzi krawędź i da się dojść od niego do każdego innego z tym, że gdy graf jest skierowany to by trzeba było iść pod prąd.
- **Graf silnie spójny -** gdy zawsze da się dojść z każdego do dowolnego innego wierzchołka uwzględniając kierunki dróg.

Kompresja danych

Podstawowe dane

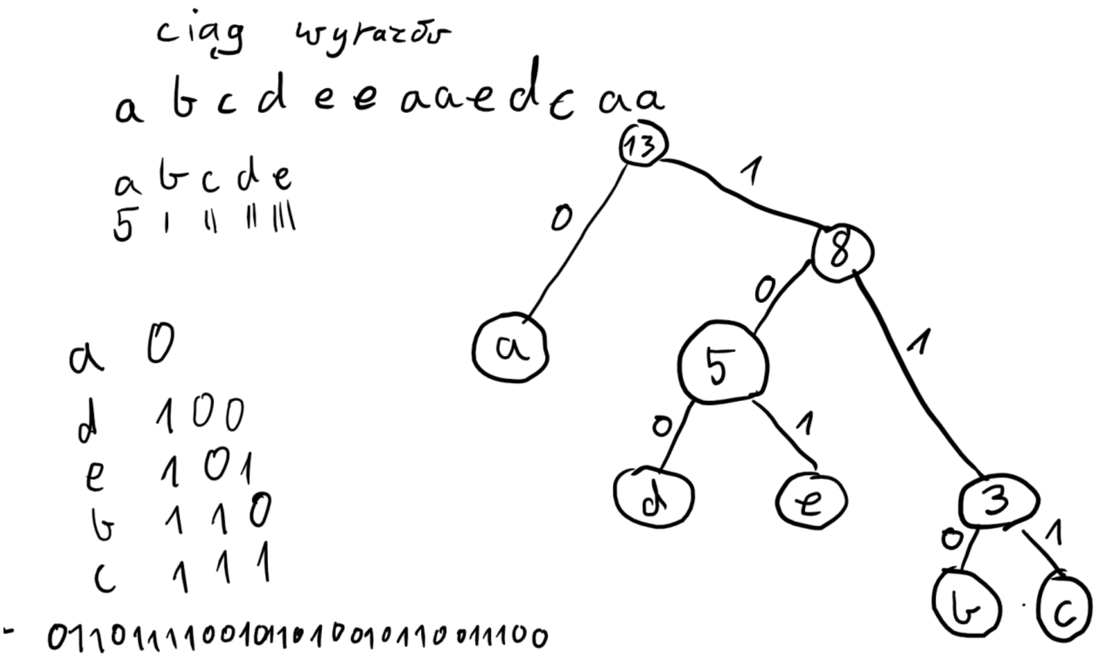
KOMPRESJA BEZSTRATNA (dane są w pełni zgodne z początkowymi)	**KOMPRESJA STRATNA** (część informacji jest tracona)
• Tekst	• Muzyka

• Programy • Bazy danych	• Filmy • Zdjęcia

Algorytm Huffmana

Jest to algorytm kompresji danych, który polega na jak największym skróceniu danych które najczęściej występują.

W dowolnym ciągu liczymy ilość wystąpienia poszczególnych znaków i budujemy na ich podstawie drzewo tak by na samym dole znajdowały się najrzadziej występujące, a im wyżej tym częściej występujące.



Typowa litera zajmuje pamięć 8 bitów czyli 1 bajt. Zauważmy, że tu poszczególne litery zajmują od 1 do 3 bitów, a kod jest jednoznacznie odczytywalny, np. jeśli litera zacznie się od **0** to od razu wiemy, że to **a**, zaś gdy od **11** to będzie to **b** lub **c.**

Algorytm bardzo dobrze działa dla danych o **małej ilości różnych znaków** w tym przypadku było tylko 5 różnych liter. Do algorytmu jest potrzebna dodatkowa pamięć przetrzymująca drzewo, oprócz samej sekwencji.

Średnia długość słowa

ciąg wyrazów

$T[n]=\{a, b, c, d, e, e, a, a, e, d, c, a, a\}$ $\quad n=13$

a	b	c	d	e
5	|	||	||	|||

$P(a)=\frac{5}{13}$

$P(b)=\frac{1}{13}$

$P(c)=\frac{2}{13}$

$P(d)=\frac{2}{13}$

$P(e)=\frac{3}{13}$

	C_j	
a	0	$\rightarrow 1$
d	100	$\rightarrow 3$
e	101	$\rightarrow 3$
b	110	$\rightarrow 3$
c	111	$\rightarrow 3$

\- 01101111001011010010110011100

Średnia dł. słowa

$$\sum_{i=1}^{n} P(i)\,|C_j| = \frac{5}{13}\cdot 1 + \frac{1}{13}\cdot 3 + \frac{2}{13}\cdot 3 + \frac{2}{13}\cdot 3 + \frac{3}{13}\cdot 3 = \frac{1}{13}(5+3+6+6+9) = 2\frac{3}{13}$$

Entropia

entropia

a b a a b c a d $\quad n=8$

a	b	c	d
4	2	1	1

$P(a)=\frac{4}{8}$

$P(b)=\frac{2}{8}$

$P(c)=\frac{1}{8}$

$P(d)=\frac{1}{8}$

$$H(S) = -\sum_{i=0}^{S-1} P(i)\cdot \log_2 P(i) = \sum_{i=0}^{S-1} P(i)\cdot \log_2\left(\frac{1}{P(i)}\right) =$$

$$= \frac{4}{8}\cdot \log_2 \frac{1}{\frac{4}{8}} + \frac{2}{8}\cdot \log_2 \frac{1}{\frac{2}{8}} + \frac{1}{8}\cdot \log_2 \frac{1}{\frac{1}{8}} + \frac{1}{8}\cdot \log_2 \frac{1}{\frac{1}{8}} = \frac{1}{2}\cdot 1 + \frac{1}{4}\cdot 2 + \left(\frac{1}{8}\cdot 3\right)\cdot 2 =$$

$$= 1\frac{3}{4}$$

Entropia jest miarą rozrzutu danych. Określa ona czy warto kompresować badane dane. Im mniejsza entropia, tym lepiej dane powinny się skompresować.

Redundancja

Redundancja

$$\underbrace{\sum_{i=1}^{m} P(i)|c_i|}_{\text{średnia dł. słowa}} - \underbrace{\sum_{i=1}^{m} P(i)\cdot \log_2 \frac{1}{P(i)}}_{\text{entropia}} \geq 0$$

Redundancja to średnia nadwyżka ponad entropię. Jest miarą przeciwną do entropii jest to wartość tego jak dużo jest nadmiarowych danych. Im wyższa redundancja, tym lepiej dane powinny się skompresować.

Kompresja RLE

aaaaaabaaabcaaaa

RLE z flagą

#6ab#3abc#4a

\# - specjalny dowolny znak
po # piszemy ilość wystąpień słowa, a następnie to słowo. np. #6a

#ilość-wystąpień słowo

RLE bez flag

aa4baa1bcaa2

jeżeli litera się powtarza 2 razy, to cyfra za nią oznacza ile jeszcze takich samych znaków powinno wystąpić.

może przyjmować wartości od 0 do n

np. aaba => aa0ba

Run-length encoding to algorytm kompresji bezstratnej gdzie znaki zapisujemy zliczając ich ilość i pisząc np. 10g zamiast gggggggggg.

W praktyce mało przydatny. Dobry dla danych, w których jest dużo powtarzających się **znaków następujących po sobie.**

LZ77

Algorytm kompresji, w którym wykorzystujemy powtarzający się ciąg, i zamiast go pisać drugi raz wskazujemy gdzie on wystąpił i jakiej jest długości.

Przykład

- Wczoraj wieczorem Hamming garał w golfa z Hammingway'em
- Wczoraj wieczorem Hamming garał w golfa z <24, 7>way'em

Oznacza to, że 24 znaki wcześniej jest siedmioliterowe powtórzenie.

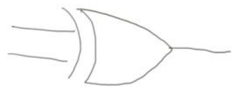
Dobry dla długich ciągów znaków, które się powtarzają. Przydatny także w praktyce.

Stringologia:

Długość Hamminga

odległość Hamminga

1° 00000000
00000001 = 1



2° 10101011011
11001011100 = 5

- bity, które się różnią

ilość bitów, które się różnią w słowach tej samej długości.

3° 110011101
000111010 = 6

Bit parzystości

By sprawdzić czy dane są poprawne często dodaje się do nich sztuczne dane które pokazują, że Informacje są zgodne z tym co się przesłało.

Na początku lub na końcu dodaje się jeden bit którego wartość oznacza ilość jedynek w całym pozostałym ciągu.

0 to nieparzysta ilość jedynek, 1 to parzysta ilość jedynek.

Kontrola parzystości

kontrola parzystości wg. kodu Hamming

Pozycja bitu		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Bit parzystości (p), danych (d)		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15	
Sekwencja sprawdzanych bitów	p1	X		X		X		X		X		X		X		X		X		X		
	p2		X	X			X	X			X	X			X	X			X	X		...
	p4				X	X	X	X					X	X	X	X					X	
	p8								X	X	X	X	X	X	X	X						
	p16																X	X	X	X	X	

111101111010001

p1 1
p2 1
p4 1
p8 1

101101101110001

p1 1
p2 0
p4 1
p8 0

Gdy wysyłamy dane przez sieć i nie chcemy by nam jakieś dane się zgubiły lub co gorsze przekłamały.

Do policzenia ilości bitów potrzebnych do korekcji bierzemy ilość wartości, które chcemy wysłać. Liczbę 7 można zapisać jako $2^r - 1 = 7$ gdy r = 3, bo $2^3 - 1 = 8 - 1 = 7$

$$7 - r = 3$$

Więc potrzeba 3 bitów, by można było skorygować.

Liczbę 9 można zapisać jako $2^r - 1 = 9$ gdy r = 5, bo $2^5 - 1 = 16 - 1 = 15$

$$9 - r = 4$$

Więc potrzeba 4 bitów, by można było skorygować. W tym przypadku trzeba dać trochę nadmiaru.

LCS – Longest common subsequence

Najdłuższa wspólna podsekwencja. Algorytm jest stosowany np.: w repozytoriach do sprawdzania różnic między poszczególnym wersjami plików.

LCS

		-1	0 G	1 R	2 A	3 T	4 K	5 A
	-1	0	0	0	0	0	0	0
	0 K	0	0	0	0	0	1	1
	1 R	0	0	1	1	1	1	1
y	2 A	0	0	1	2	2	2	2
	3 T	0	0	1	2	3	3	3
	4 K	0	0	1	2	3	4	4
	5 I	0	0	1	2	3	4	4

x

```
if y[j]==x[i]
then T[i][j]=T[i-1][j-1]+1

else
    MAX(T[i-1][j], T[i][j-1])
```

4 - wynik - ilość wspólnych znaków w odpowiedniej kolejności

nie muszą być obok siebie!!!

	-1	0 M	1 A	2 L	3 A	4 R	5 Z
-1	0	0	0	0	0	0	0
0 M	0	1	1	1	1	1	1
1 U	0	1	1	1	1	1	1
2 R	0	1	1	1	1	2	2
3 A	0	1	2	2	2	2	2
4 R	0	1	2	2	2	3	3
5 Z	0	1	2	2	2	3	4

		A	M	L	A	R	Z
	0	0	0	0	0	0	0
M	0	0	1	1	1	1	1
U	0	0	1	1	1	1	1
R	0	0	1	1	1	2	2
A	0	1	1	1	2	2	2
R	0	1	1	1	2	3	3
Z	0	1	1	1	2	3	4

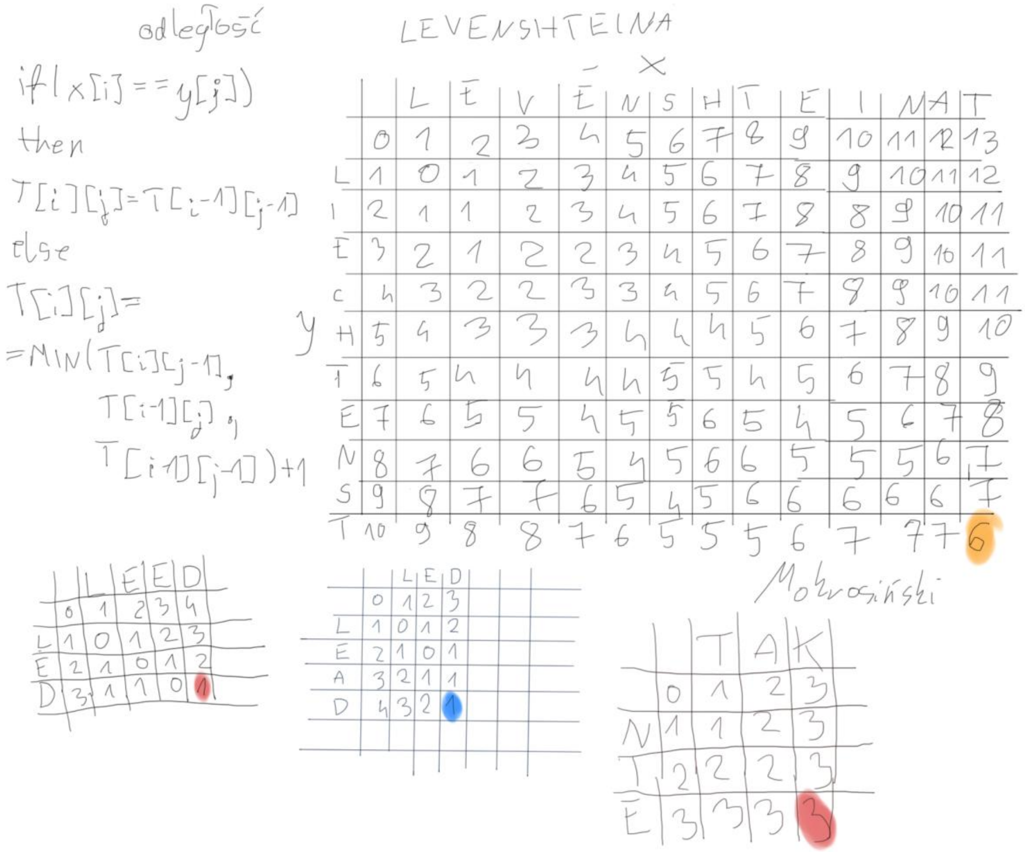
wszystkie możliwości ciągów po 2 kolejne znaki (niekoniecznie obok siebie)

1° 2° 3° 4°

		A	L	A	Z	R	M
	0	0	0	0	0	0	0
M	0	0	0	0	0	0	1
U	0	0	0	0	0	0	1
R	0	0	0	0	0	1	1
A	0	1	1	1	1	1	1
R	0	1	1	1	1	2	2
Z	0	1	1	1	2	2	2

		A	L	A	R	Z	M
	0	0	0	0	0	0	0
M	0	0	0	0	0	0	1
U	0	0	0	0	0	0	1
R	0	0	0	0	1	1	1
A	0	1	1	1	1	1	1
R	0	1	1	1	2	2	2
Z	0	1	1	1	2	3	3

Odległość Levenshteina (edycyjna)



Służy do wyznaczania różnic między poszczególnymi wyrazami

Mówi nam o tym ile minimalnie znaków musimy **usunąć, przestawić** lub **zastąpić,** aby jeden wyraz zamienić w drugi.

Algorytm ten może służyć jako podpowiedzi w słownikach np.: gdy zrobimy jakąś literówkę zjemy jakąś jedną literę możemy sobie obliczyć do jakich wyrazów ze słownika jest podobny ten, w którym się pomyliliśmy.

Geometria obliczeniowa

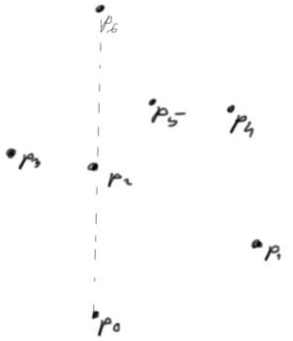
Otoczka wypukła

Najmniejszy wielokąt wypukły otaczający zbiór punktów.

Algorytm Jarvisa

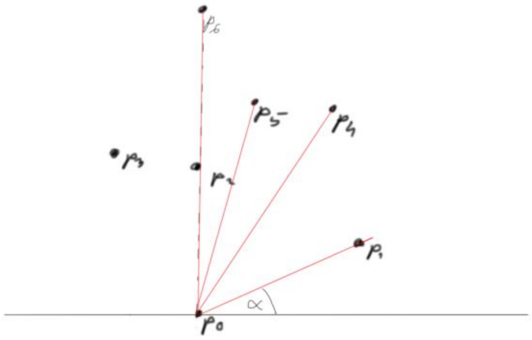
Złożoność obliczeniowa O(n)+O(nh) ogólnie O(nh)

W najgorszym przypadku O(n^2) gdy h jest bliskie n

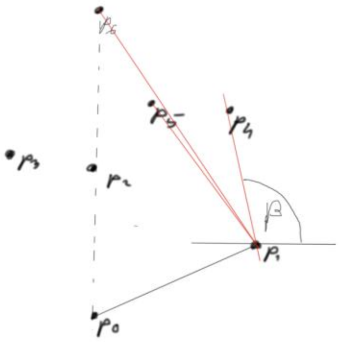


krok 1 – szukamy najniżej położonego punktu (p_0) jeżeli ta sama wysoko 2 pkt, to bierzemy tego najbardziej po lewo
krok 2 – 〃 najwyżej 〃 (p_6) 〃 〃 prawo

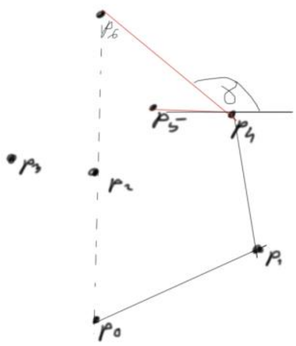
krok 3.



krok 4.

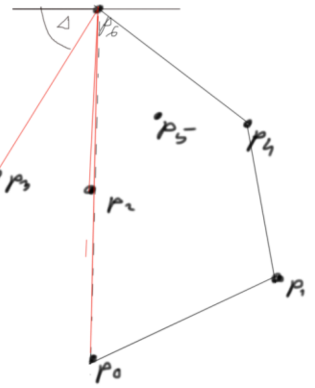


krok 5

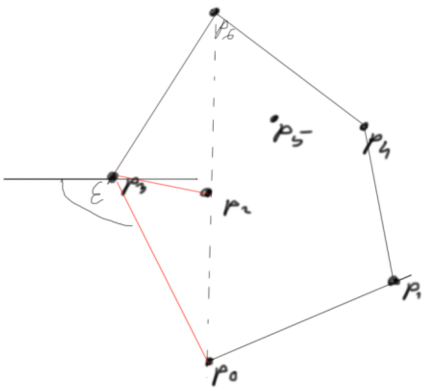


czy $\Delta < 0$?!?

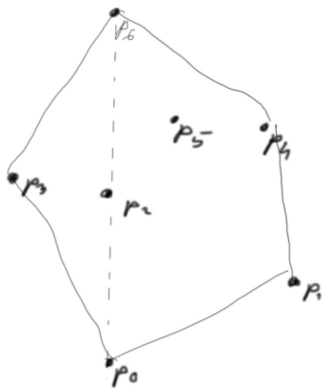
krok 6



krok 7



krok kolejny



Krok 1 i 2: Wyszukujemy najniższy i najwyższy punkt i dzielimy punkty na prawe i lewe.

Krok 3 – 5: Porównujemy punkty z nie niższymi punktami i szukamy te które są pod najmniejszym kątem

Krok 6: Gdy dojdziemy na samą górę zaczynamy patrzeć kąt w drugą stronę.

Algorytm Grahama

Wyszukujemy najniższy punkt.

Sortujemy punkty względem kąta od najniższego punktu odwrotnie do wskazówek zegara.

Wrzucamy pierwsze trzy na stos.

Jeśli kolejny punkt jest na lewo względem dwóch najwyższych punktów ze stosu to dorzucamy do stosu.

Jeśli kolejny punkt jest na prawo względem dwóch najwyższych to wywalamy ten, który był na górze i sprawdzamy ten warunek aż nie będzie skrętu w lewo.

Jeśli będzie już skręcało w lewo to dorzucamy do stosu.

Złożoność obliczeniowa O(n) + O(n log n) + O(1) + O(n)

Wyszukanie najniższego + sortowanie + dorzucenie do stosu + przejście po wszystkich elementach

W praktyce O(n log n)

Rysunki na prezentacji 12 z algorytmów dość dobrze pokazują działanie tego algorytmu.