Algorytmy i struktury danych

Algorytm, stos, kolejka, ONP

Krzysztof M. Ocetkiewicz
Krzysztof.Ocetkiewicz@eti.pg.edu.pl

Katedra Algorytmów i Modelowania Systemów, WETI, PG

Algorytm

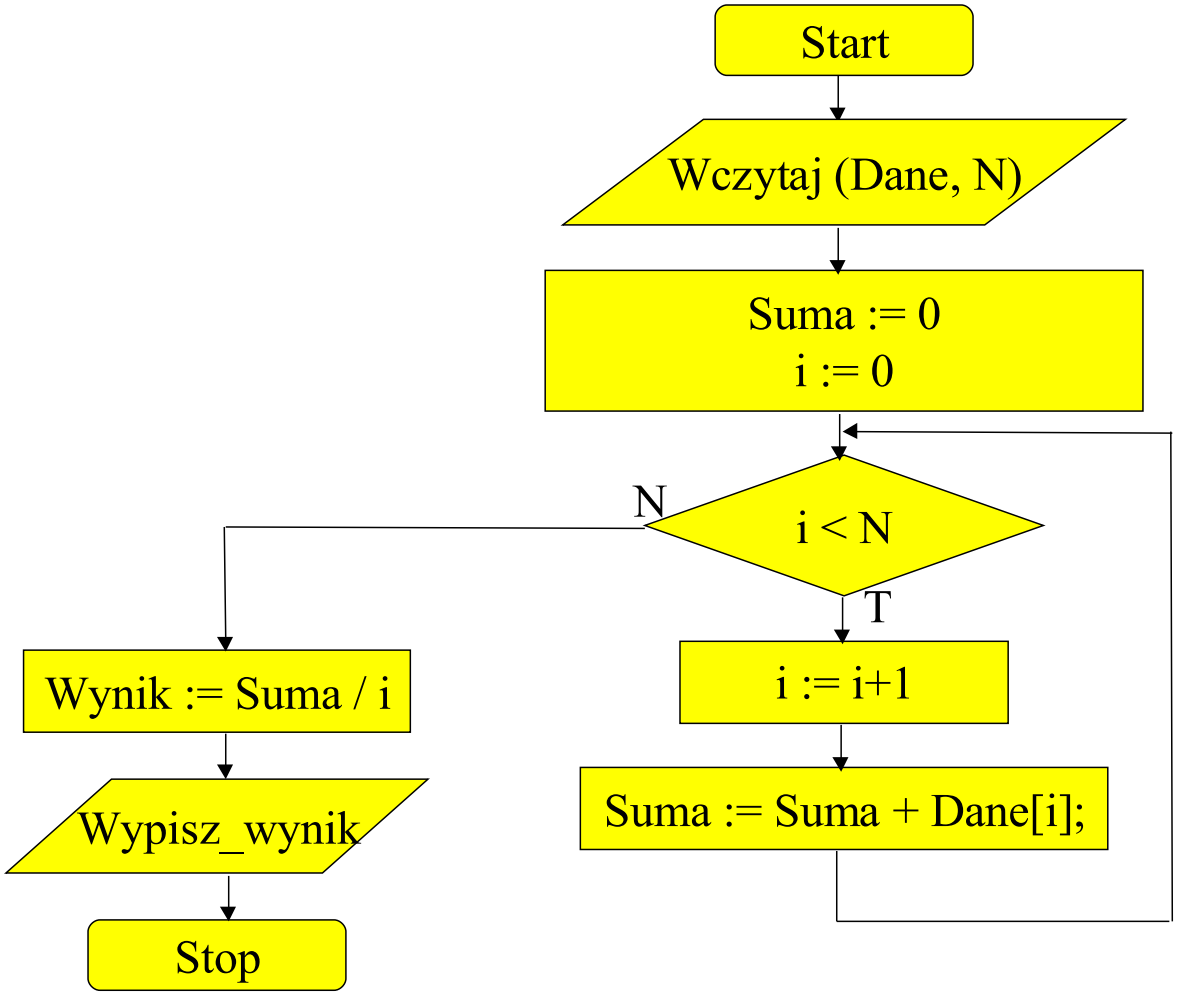
- przepis postępowania prowadzący do rozwiązania ustalonego problemu, określający ciąg czynności elementarnych, które należy w tym celu wykonać [Encyklopedia PWN]
- dany problem (wejście)
- sekwencja czynności
- rozwiązanie problemu (wyjście)

Zapis algorytmu

- język naturalny
  - aby obliczyć obwód koła, weź jego promień, przemnóż przez 2 a następnie przemnóż przez π
  - aby trafić na dworzec kolejowy wsiądź do tramwaju numer 6, jedź nim aż zobaczysz duży zielony budynek, wysiądź i przejdź na drugą stronę ulicy

Zapis algorytmu

- diagram



Zapis algorytmu

- pseudokod:

```
1: NWD(a, b):
2: while b ≠ 0 do
3:         c = a mod b
4:         a = b
5:         b = c
6: end while
7: return a
```

Pseudokod

- umowny język programowania zbliżony składnią do ulubionego języka autora
- posiada typowe struktury sterujące (for, while, if), jednak część instrukcji wyrażona jest językiem naturalnym
- np.:

```
1: while b ma dzielniki różne od siebie i 1 do
2:         podziel b przez najmniejszy dzielnik większy od 1
3: end while
```

Algorytm

- co nas interesuje w algorytmie:
  - czy się kończy?
  - jeżeli tak, to jak długo trzeba czekać?
  - czy zawsze daje wynik?
  - czy daje poprawny wynik?
  - o ile się może pomylić?

Czy algorytm się kończy?

- problem Collatza:

1: a — liczba naturalna
2: **while** $a > 1$ **do**
3: **if** a podzielne przez 2 **then**
4: $a = a/2$
5: **else**
6: $a = 3 * a + 1$
7: **end if**
8: **end while**

Algorytmy

- różne algorytmy rozwiązujące ten sam problem mogą się różnić:
  - czasem działania
  - ilością wymaganej pamięci
  - dokładnością rozwiązania
  - zakresem stosowalności (dziedziną)
  - komplikacją
  - podatnością na zrównoleglanie
  - stabilnością numeryczną
  - ...

Złożoność algorytmu

- chcemy móc porównywać efektywność algorytmów
- najczęściej interesuje nas czas działania
  (czasem również ilość zajętej pamięci)
- ale, w zależności od:
  - komputera
  - implementacji
  - kompilatora
  - ...

  wykonanie tego samego algorytmu zajmie inną ilość czasu

Złożoność czasowa algorytmu

- zamiast samego czasu działania algorytmu badamy zależność czasu potrzebnego do wykonania algorytmu od rozmiaru danych wejściowych
- efektywność algorytmu charakteryzujemy funkcją, której parametrem jest rozmiar danych wejściowych, a wynikiem — czas, przez jaki algorytm będzie działał

Złożoność czasowa algorytmu

- np. sumowanie elementów tablicy:

1: $suma = 0$
2: **for** $i = 1, \ldots, n$ **do**
3: $\quad\quad suma = suma + a[n]$
4: **end for**

wymaga wykonania $6n + 2$ instrukcji

Rozmiar danych

- zamiast dokładnie obliczać liczbę bitów, którą zajmują dane wejściowe, dążymy do wyrażenia rozmiaru danych jednym lub dwoma parametrami
- zazwyczaj łatwo wskazać taki parametr, np.:
  - w przypadku wyszukiwania liczby w tablicy — jest to rozmiar tablicy
  - w przypadku sortowania — ilość elementów do uporządkowania
  - w przypadku wyszukiwania napisu w napisie — ilość znaków w jednym i drugim napisie
  - itd.
- jeżeli jednak mamy z tym problem, zawsze możemy zastosować liczbę bitów (i nikt nie będzie miał prawa się przyczepić)

Złożoność czasowa algorytmu

- dokładna postać funkcji (a zwłaszcza stałe współczynniki) zależy od wielu czynników (komputer, implementacja itd.)
- różnica pomiędzy $10n^3 + n$ a $10n^3 + 2n$ jest mało istotna
- zamiast konkretnej funkcji, badamy tempo jej wzrostu i przedstawiamy je za pomocą tzw. notacji wielkiego O
- wynikiem takiej funkcji jest nie czas, lecz liczba umownie zdefiniowanych operacji podstawowych
- np. złożoność $O(n^2)$ oznacza, że funkcja opisująca czas działania algorytmu rośnie nie szybciej niż funkcja n^2

Złożoność czasowa algorytmu

- formalnie $f(x) = O(g(x))$ wtedy i tylko wtedy, gdy istnieją N i C takie, że dla każdego $x > N$ zachodzi $|f(x)| < C|g(x)|$
- w programowaniu liczba kroków rzadko jest ujemna, więc wartości bezwzględne możemy pominąć:
  $f(x) = O(g(x)) \Leftrightarrow \exists_{C,N} \forall_{x>N} f(x) < Cg(x)$
- np. gdy $f = O(n^2)$ mówimy: "f rośnie nie szybciej niż n^2" lub, mniej formalnie: "f ma kwadratowe tempo wzrostu"

Typowe złożoności

- $O(1)$ — stała
- $O(\log n)$ — logarytmiczna
- $O(n)$ — liniowa
- $O(n \log n)$ — liniowo-logarytmiczna
- $O(n^2)$ — kwadratowa
- $O(n^3)$ — sześcienna
- w ogólnym przypadku, algorytmy o złożoności wyrażonej wielomianem uznawane są za "efektywne" zaś te o wyższej (np. wykładniczej) za "nieefektywne"

Własności

- współczynniki są nieistotne: $O(n^2)$ to to samo co $O(100n^2)$ $(100n^2 = O(n^2))$
- istotny jest tylko najszybciej rosnący składnik sumy: $O(n^3)$ opisuje takie samo tempo wzrostu jak $O(n^3 + n^2 + n + \log n)$, czyli $n^3 + n^2 + n \log n = O(n^3)$
- dla dowolnego $a > 0$ i $b > 0$ $\log^b n = O(n^a)$
  tzn. logarytm z n w dowolnej dodatniej potędze
  rośnie nie szybciej niż n w dowolnej dodatniej potędze

Złożoność czasowa algorytmu

- złożoność pesymistyczna — liczba kroków potrzebna do wykonania algorytmu w najgorszym możliwym przypadku (będzie nas najczęściej interesowała)
- złożoność optymistyczna — liczba kroków potrzebna do wykonania algorytmu w najlepszym możliwym przypadku (zazwyczaj nieistotna)
- złożoność oczekiwana (lub średnia) — liczba kroków wykonanych w "średnim" przypadku
- koszt zamortyzowany — średnia liczba kroków wykonanych w określonym scenariuszu użycia

Szacowanie złożoności

- pętla:

  1: **for** $i = 1, \ldots, n$ **do**
  2: ...
  3: **end for**

  ma złożoność $O(n)$

- pętla:

  1: $i = 1$
  2: **while** $i < n$ **do**
  3: ...
  4: $i = i * 2$
  5: **end while**

  ma złożoność $O(\log n)$

Szacowanie złożoności

- złożoność dwóch zagnieżdżonych pętli jest iloczynem ich złożoności, np.:

```
1: for i = 1, ..., n do
2:         for j = 1, ..., n do
3:                 ...
4:         end for
5: end for
```

ma złożoność $O(n^2)$ (każda z pętli ma złożoność $O(n)$)

Szacowanie złożoności

- złożoność sekwencji kroków jest sumą ich złożoności, np.:

  1: krok o złożoności liniowej
  2: krok o złożoności sześciennej
  3: krok o złożoności kwadratowej

  ma złożoność $O(n + n^3 + n^2) = O(n^3)$
  (w sekwencji liczy się tylko krok o największej złożoności)

Szacowanie złożoności

- co gdy:

```
1: for i = 1, ..., n do
2:         for j = i, ..., n do
3:                 ...
4:         end for
5: end for
```

  ?

- wykonamy $1 + 2 + 3 + \ldots + n = \frac{(1+n)n}{2}$ kroków, czyli złożoność wynosi $O(n^2)$

Złożoność algorytmu

- znając złożoność algorytmu możemy oszacować czas potrzebny na rozwiązanie zadania
- znając zakres danych możemy oszacować, jak szybki musi być nasz algorytm
- obecnie procesory działają z częstotliwością 1–4GHz — wykonują ok. 1000000000 najbardziej podstawowych kroków na sekundę
- jeżeli rozmiar danych (n) jest rzędu 500–1000, to najgorszy algorytm na jaki możemy sobie pozwolić to $O(n^2 \log n)$ — $O(n^3)$

Złożoność czasowa algorytmu

- jeżeli rozmiar danych (n) jest rzędu 10000, możemy pozwolić sobie na algorytm o złożoności $O(n^2)$
- jeżeli rozmiar danych (n) jest większy niż 100000, to algorytm powinien mieć złożoność nie większą niż $O(n \log n)$

Złożoność pamięciowa algorytmu

- funkcja opisująca tempo wzrostu zależności rozmiaru użytej pamięci od rozmiaru danych wejściowych
- także tu możemy wyróżnić złożoność pesymistyczną, optymistyczną, średnią
- czy złożoność pamięciowa algorytmu może być większa od obliczeniowej?

Iteracja vs rekurencja

- iteracja — powtarzanie określonej czynności (np. w pętli)
- np.:

1: $s = 0$
2: **for** $i = 1, \ldots, n$ **do**
3: $\quad\quad s = s + tab[i]$
4: **end for**

Iteracja vs rekurencja

- rekurencja — odwołanie do samego siebie
- np.:
  $Fib(0) = 0$
  $Fib(1) = 1$
  $Fib(n) = Fib(n-1) + Fib(n-2)$

Rekurencja

- głębokość rekurencji — liczba "poziomów zagłębień" — ile maksymalnie razy funkcja wywoła samą siebie
- liczba ta jest zazwyczaj ograniczona przez rozmiar stosu (im więcej funkcja ma parametrów i zmiennych lokalnych, tym szybciej zużywa stos)
- w systemie Windows domyślny rozmiar stosu to 1MB (można to zmienić odpowiednio kompilując program)
- w systemie Linux domyślny rozmiar stosu to 10MB (można to zmienić mając uprawnienia root-a)

Iteracja vs rekurencja

- iteracja i rekurencja są równoważne — wszystko co da się zrobić iteracyjnie można zrobić rekurencyjnie i na odwrót (nie znaczy to jednak, że oba rozwiązania będą tak samo skomplikowane)
- rozwiązanie rekurencyjne jest zazwyczaj krótsze, bardziej zwarte
- rozwiązanie iteracyjne jest zazwyczaj nieco szybsze (brak ciągłego wołania funkcji i przekazywania parametrów)

Iteracja vs rekurencja

- np. :

```
int FibRek(int n) {
      if(n < 2) return n;
      return FibRek(n-1) + FibRek(n-2);
      };
```

Iteracja vs rekurencja

- np. :

```
int FibIter(int n) {
      int i, fb, fp, s;
      fp = 0;
      fb = 1;
      for(i = 2; i <= n; i++) {
            s = fp + fb;
            fp = fb;
            fb = s;
            };
      return fb;
      };
```

Iteracja vs rekurencja

- np. :

```
int _FibRek2(int n,int i,int fc,int fp) {
      if(i==n) return fc;
      return _FibRek2(n,i+1,fc+fp,fc);
      };
int FibRek2(int n) {
      return _FibRek2(n,1,1,0);
      };
```

Iteracja vs rekurencja

- np. :

```
for(i = a; i < b; i++) {
      ...
      };

void Petla(int i, int b) {
      if(i >= b) return;
      ...
      Petla(i + 1, b);
      };
Petla(a, b);
```

Struktura danych

- sposób uporządkowania danych
- struktura danych:
  - przechowuje określone dane
  - umożliwia wykonanie określonych operacji na tych danych (znajdź największy, sprawdź czy istnieje, przejrzyj wszystkie itp.) z określonymi złożonościami
- z pierwszego wynika drugie, ale czasem oddzielamy pierwszy aspekt od drugiego
- np. stos (struktura danych w drugim znaczeniu: pozwalająca szybko dodać element i szybko pobierać elementy w odwrotnej kolejności dodawania) możemy zaimplementować np. na tablicy czy liście (struktury danych w pierwszym znaczeniu)

Czym się różnią struktury danych

- czasem dostępu do elementu
- czasem wyszukiwania elementu
- czasem usuwania elementu
- sposobem przechowywania elementów (np. uporządkowane lub nie)
- dodatkową wymaganą pamięcią
- możliwością dynamicznego zwiększenia rozmiaru
- trwałością wskaźników na elementy
- trudnością implementacji
- ...

Tablica

- najbardziej podstawowa struktura danych
- przechowujemy nieuporządkowane, ponumerowane obiekty w pierszych n komórkach tablicy oraz liczbę elementów
- z góry określony rozmiar (pojemność)

Tablica

- operacje:
  - dostęp do obiektu o podanym numerze: $O(1)$
  - dodanie elementu do tablicy (o ile jest miejsce): $O(1)$
  - usunięcie elementu o podanym numerze z tablicy: $O(1)$
  - usunięcie elementu o podanej wartości z tablicy: $O(n)$
  - sprawdzenie czy element należy do tablicy: $O(n)$
  - znalezienie minimum: $O(n)$
  - znalezienie maksimum: $O(n)$
  - wstawienie elementu przed/za wskazanym elementem: ? (ćwiczenie)

Tablica dynamiczna

- co jeżeli nie wiemy ile maksymalnie będziemy potrzebować miejsca?
- możemy przydzielić początkowo m komórek pamięci i gdy miejsce się skończy, alokować nową tablicę, o jeden większą, przepisać zawartość starej i zwolnić starą tablicę
- takie rozwiązanie powoduje, że złożoność wstawienia elementu spada do $O(n)$ (musimy przepisać wszystkie elementy z jednej tablicy do drugiej)

Tablica dynamiczna

- możemy ten wynik poprawić
- zmieniając rozmiar tablicy nie tworzymy o jeden większej lecz dwa razy większą
- będziemy marnować więcej pamięci
- pesymistyczna złożoność wstawienia elementu pozostanie nadal $O(n)$
- jednak koszt zamortyzowany wstawienia spadnie do $O(1)$

Tablica dynamiczna

- rozważmy sekwencję n wstawień i niech tablica ma początkowo rozmiar 1
- pierwsze wstawienie wypełni tablicę
- drugie wstawienie: zmiana rozmiaru na 2 (1 przepisanie)
- trzecie wstawienie: zmiana rozmiaru na 4 (2 przepisania)
- czwarte wstawienie: wstawienie w wolne miejsce
- piąte wstawienie: zmiana rozmiaru na 8 (4 przepisania)
- ...

Tablica dynamiczna

- w najgorszym przypadku (tuż przed wstawieniem ostatniego elementu zmieniamy rozmiar) wykonamy $n - 1 + (n-1)/2 + (n-1)/4 + \ldots + 2 + 1 \leq 2n$ przepisań
- średnio na wstawienie wykonamy $2n/n = 2$ przepisania
- koszt zamortyzowany wstawienia wynosi zatem $O(1)$

Tablica zwykła vs dynamiczna

	zwykła	dynamiczna
złożoność pamięciowa	$O(K+1)$	$O(n)$
wstawienie (pesym.)	$O(1)$	$O(n)$
wstawienie (zamort.)	$O(1)$	$O(1)$
usunięcie	$O(1)$	$O(1)$ lub $O(n)$
wyszukanie	$O(n)$	$O(n)$
modyfikacja	$O(1)$	$O(1)$
trwałość wskaźników	nie	nie
trwałość wsk. (bez usuwania)	tak	nie

Wyszukiwanie elementu

- dany jest ciąg elementów a_i, $i = 1, \ldots, n$
- należy sprawdzić, czy element w znajduje się w tym ciągu (i na jakiej pozycji)
- gdy nie wiemy nic o danych w naszej strukturze nie ma wyjścia — trzeba przejrzeć wszystkie elementy i porównać je z elementem w
- złożoność: $O(n)$ — w najgorszym przypadku (w nie ma w ciągu lub jest na ostatnim miejscu) potrzebujemy n kroków

Wyszukiwanie w tablicy

1: **for** $i = 1, \ldots, n$ **do**
2: **if** $w = a_i$ **then**
3: **return** i
4: **end if**
5: **end for**
6: **return** "nie znaleziono"

Dlaczego algorytm działa?

- dlaczego ten algorytm jest poprawny?
- dowód poprawności algorytmu
- koncepcja niezmiennika

Wyszukiwanie liniowe

- wejście: $a_1, \ldots, a_n, w, n$
- niezmiennik: w kroku $i = 1, \ldots, n$ element w nie znajduje się pośród elementów $a_1, \ldots, a_{i-1}$
- dowód niezmiennika (indukcja):
  - $i = 1$ — oczywiste
  - $i + 1$ — niezmiennik zachodzi dla i, jeżeli $a_i = w$ algorytm się kończy, w przeciwnym wypadku $a_i \neq w$ więc niezmiennik zachodzi dla $i + 1$
- dodatkowo dowiedliśmy, że algorytm zwraca pierwszą, od lewej, pozycję w

Wyszukiwanie liniowe

- implementacja:

```
int Znajdz(int a[], int n, int w) {
      int i;
      for(i = 0; i < n; ++i) {
            if(a[i] == w) return i;
            };
      return -1; // -1 to ‘‘nie znaleziono’’
      };
```

Wyszukiwanie liniowe

- implementacja rekurencyjna:

```
int ZnajdzRek(int a[], int n, int w) {
      return ZnajdzR(a, n, w, 0);
      };
int ZnajdzR(int a[], int n, int w, int p) {
      if(n == 0) return -1;
      if(a[p] == w) return p;
      return ZnajdzR(a, n-1, w, p+1);
      };
```

Szukanie w uporządkowanych danych

- uporządkowanie danych pozwala przyspieszyć wyszukiwanie
- przed elementem a_i znajdują się elementy nie większe, za nim — nie mniejsze
- jeżeli poszukiwany element jest mniejszy od tego, z którym go porównujemy to jeżeli jest w tablicy, na pewno będzie gdzieś przed nim

Szukanie w uporządkowanych danych

- zatem:
- sprawdzamy element w połowie przedziału $(\frac{n}{2})$
- jeżeli jest on większy od wyszukiwanego, powtarzamy procedurę w górnej połowie, jeżeli mniejszy — w dolnej
- ...aż znajdziemy nasz element lub zawęzimy przedział do 1 elementu — wtedy w nie ma w tablicy

Wyszukiwanie binarne

1: $l = 1,\ p = n$
2: **while** $l \leq p$ **do**
3: $\quad s = podloga(\frac{l+p}{2})$
4: $\quad$ **if** $w = a_s$ **then**
5: $\quad\quad$ **return** s
6: $\quad$ **else if** $w < a_s$ **then**
7: $\quad\quad p = s - 1$
8: $\quad$ **else**
9: $\quad\quad l = s + 1$
10: $\quad$ **end if**
11: **end while**
12: **return** “nie znaleziono”

Wyszukiwanie binarne

- dlaczego jest poprawne?
- niezmiennik: wyszukiwany element może znajdować się tylko na pozycjach $l, \ldots, p$ w tablicy

Wyszukiwanie binarne

- złożoność: $O(\log n)$ (dlaczego?)
- dlaczego wybieramy środkowy element?
- jaka będzie złożoność jeżeli wybierzemy element $l + (p - l) * 0.9$?
- jaka będzie złożoność jeżeli wybierzemy element $p - 4$ lub $l + 3$?

Wyszukiwanie binarne

- podobny algorytm można wykorzystać do szukania miejsca zerowego funkcji (przy pewnych dodatkowych założeniach)
- nie nadaje się do struktur bez łatwego dostępu do wybranego elementu

Wyszukiwanie binarne

- co zrobić gdy nie znamy n?
- zaczynamy od $l = 1$ i $p = 1$
- dopóki $w > a_p$ niech $p = 2p$
- wyszukaj binarnie w dla przedziału l do p
- ciągle $O(\log n)$

Wyszukiwanie interpolacyjne

- jeżeli dane rosną w przybliżeniu liniowo, możemy pokusić się o próbę szybszego "trafienia" w wyszukiwany element
- np. szukając słowa na literę "w" w słowniku nie otwieramy słownika w połowie lecz pod koniec
- zamiast wybierać środek przedziału, wybieramy element $l + \frac{w - a_l}{a_p - a_l}(p - l)$
- jeżeli nie trafimy, ograniczamy przedział tak jak w wyszukiwaniu binarnym i strzelamy dalej

Wyszukiwanie interpolacyjne

- przy "dobrych" danych zakończymy pracę szybciej niż wyszukiwanie binarne ($O(\log \log n)$)
- przy "złych" danych będziemy potrzebowali do $O(n)$ kroków, np. $\{a_i\} = 1, 100, 101, 102, 103, 104$, szukamy 100

Wyszukiwanie

- C (`stdlib.h`)
  - `lfind`
  - `bsearch`
- C++ (STL, `algorithm`)
  - `find`
  - `binary_search`
  - `lower_bound`
  - `upper_bound`

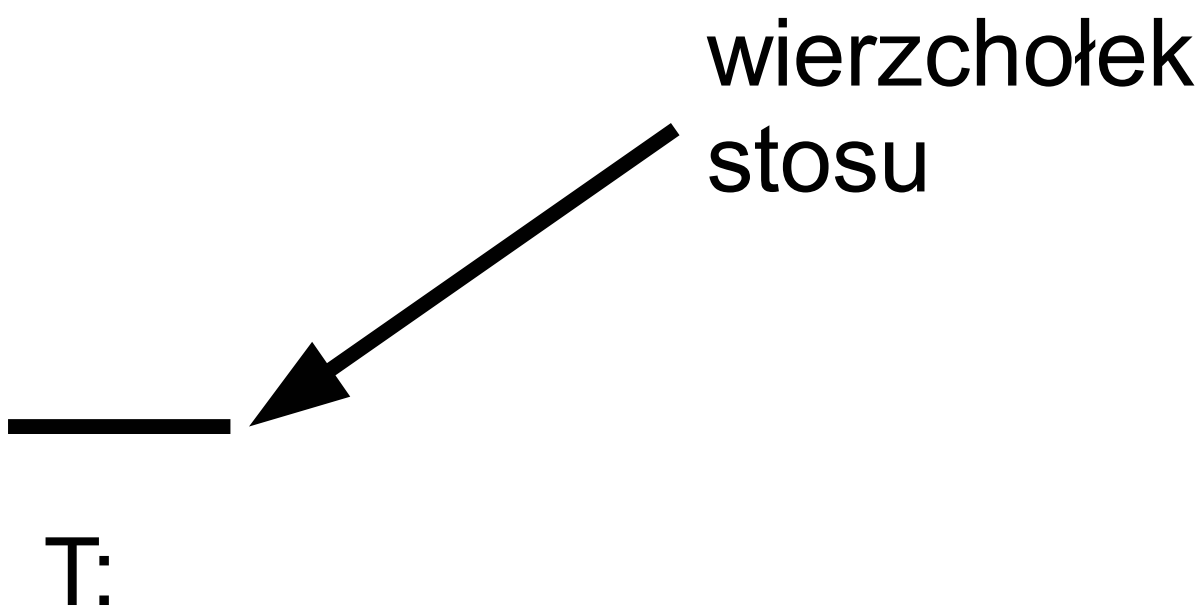
Stos

- stos jest strukturą danych umożliwiającą następujące operacje:
  - `push` — włożenie elementu na wierzchołek stosu (w czasie $O(1)$)
  - `pop` — zdjęcie elementu z wierzchołka stosu (w czasie $O(1)$)
  - `top` — wartość elementu na szczycie stosu (w czasie $O(1)$)
  - `empty` — sprawdzenie, czy stos jest pusty (w czasie $O(1)$)
- LIFO — Last In First Out — ostatni włożony element będzie pierwszym wyjętym
- włożenie n elementów na stos a następnie ich zdjęcie odwraca kolejność

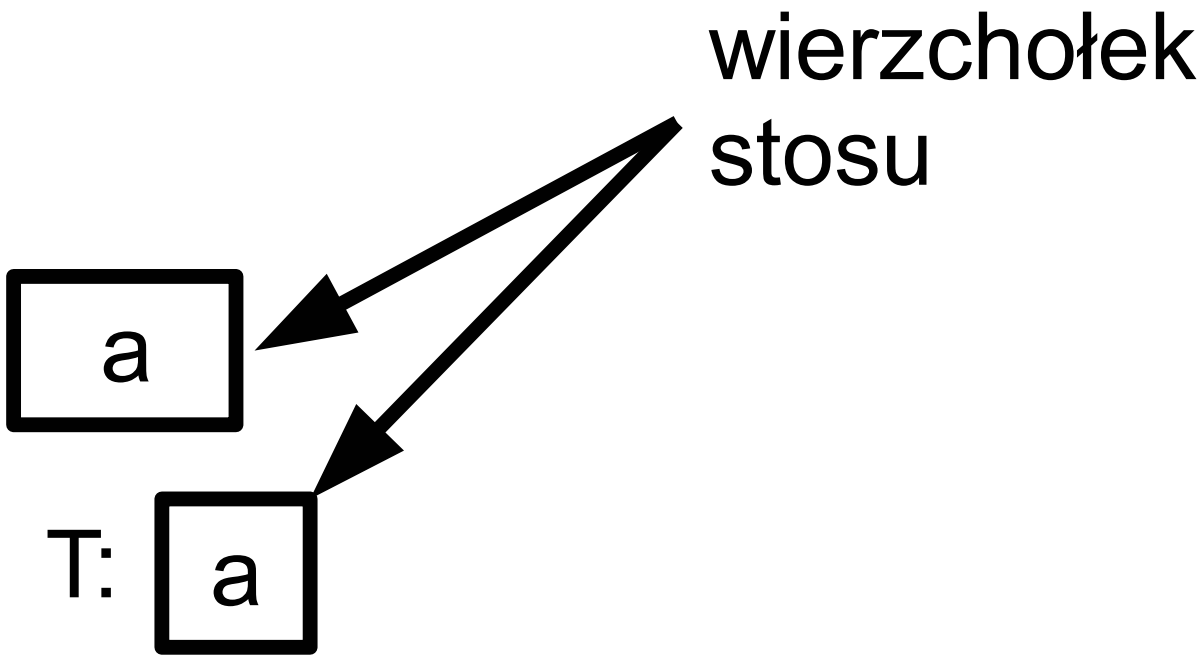
Implementacja stosu w tablicy

- składniki: tablica T (zwykła lub dynamiczna)
- `push(`E`)` — włożenie na stos elementu E:
  - dodaj E na koniec T
- `pop()` — zdjęcie ze stosu elementu:
  - zapisz w E element z końca tablicy T
  - usuń ustatni element z T
  - zwróć E
- `top()` — wartość elementu na szczycie stosu:
  - zwróć element z końca tablicy T
- `empty()` — sprawdzenie czy stos jest pusty:
  - sprawdź, czy tablica T zawiera 0 elementów

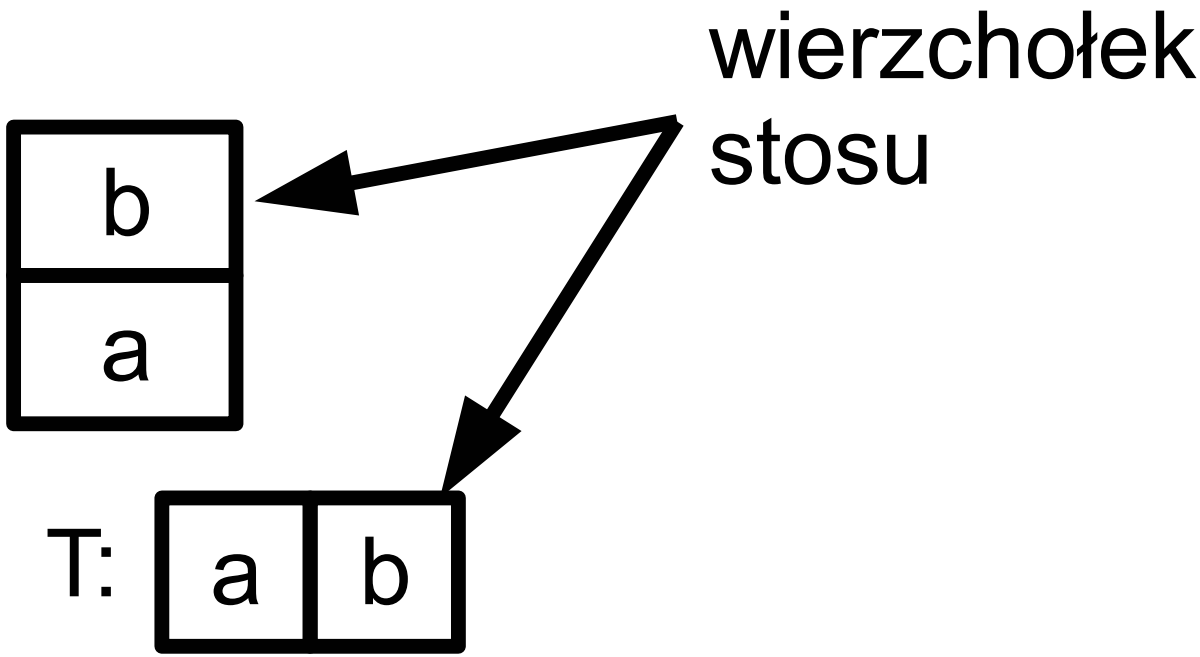
stos pusty



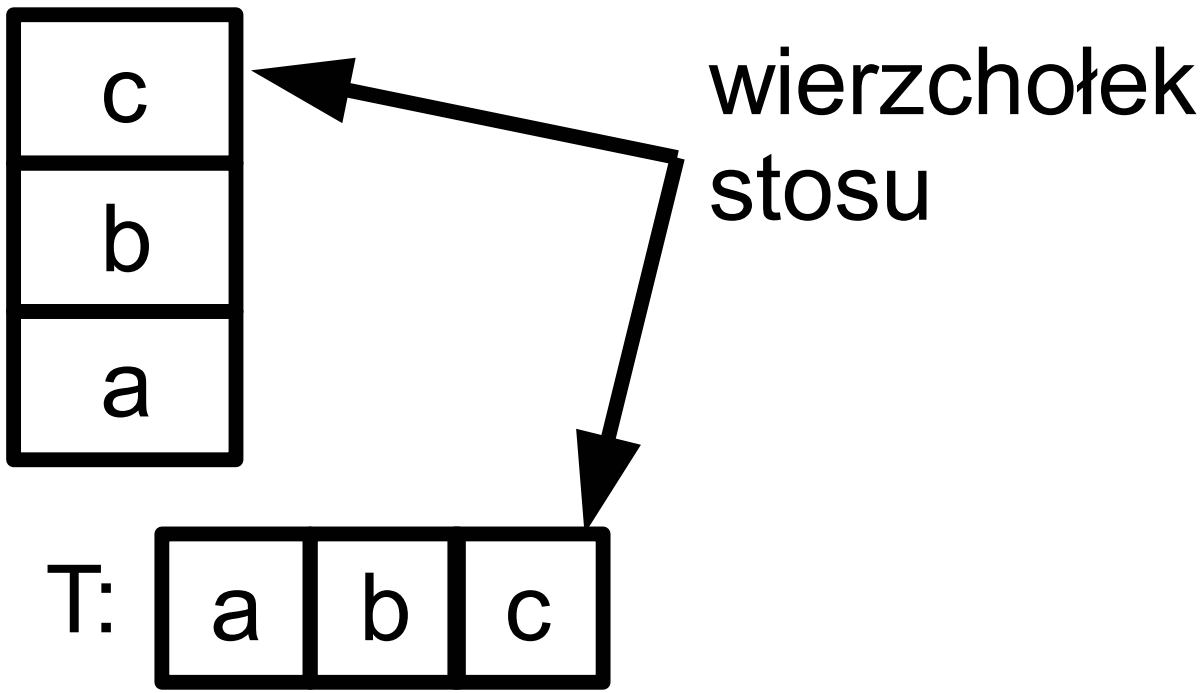
włóż a na stos



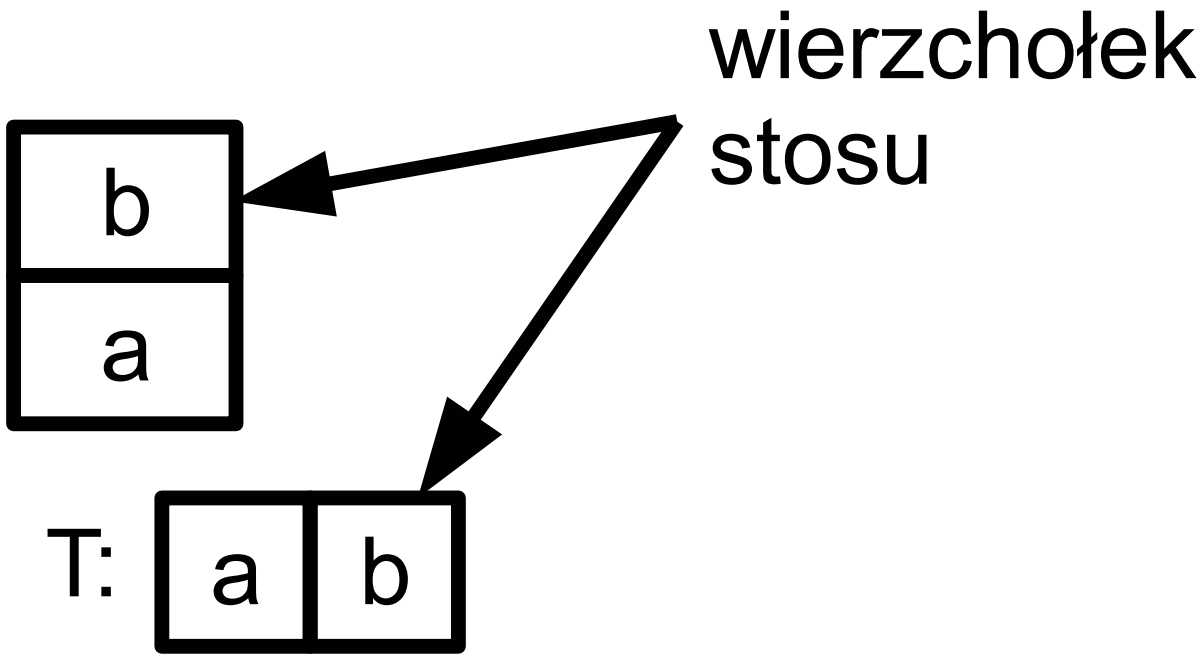
włóż b na stos



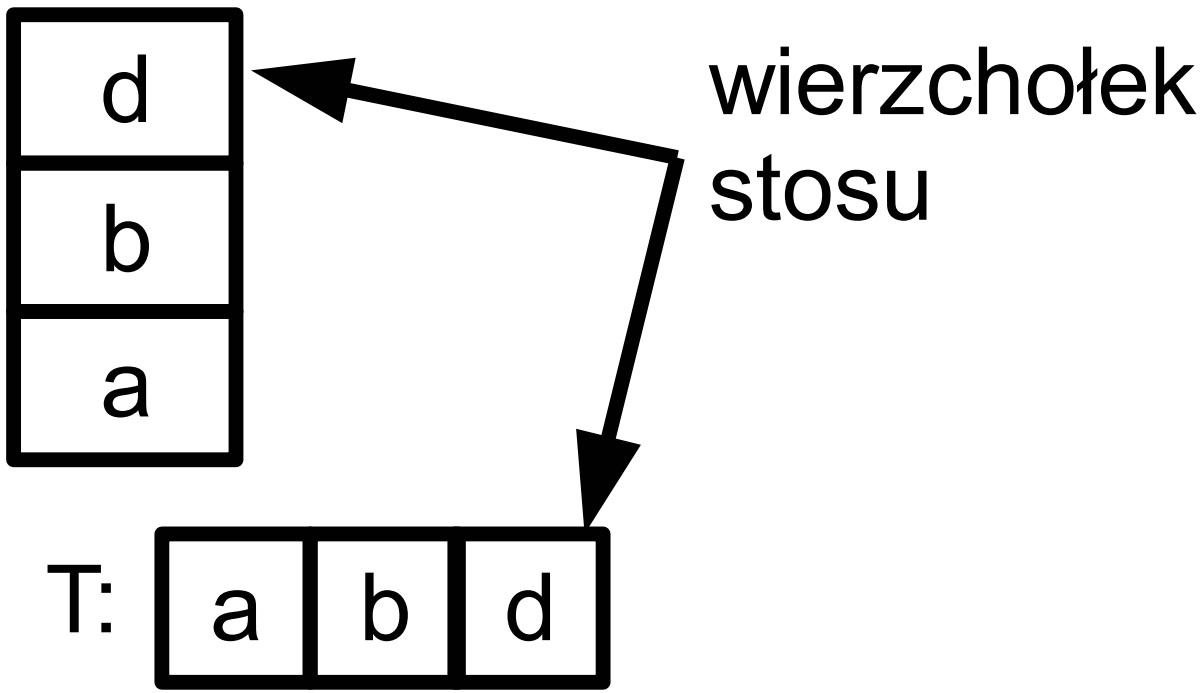
włóż c na stos



zdejmij ze stosu $\longrightarrow$ c



włóż d na stos



Kolejka

- kolejka jest strukturą danych umożliwiającą następujące operacje:
  - `enqueue` — włożenie elementu na koniec kolejki (w czasie $O(1)$)
  - `dequeue` — zdjęcie elementu z początku kolejki (w czasie $O(1)$)
  - `front` — wartość elementu na początku kolejki (w czasie $O(1)$)
  - `empty` — sprawdzenie, czy kolejka jest pusta (w czasie $O(1)$)
- FIFO — first in first out — pierwszy włożony element będzie pierwszym wyjętym

Implementacja kolejki w tablicy

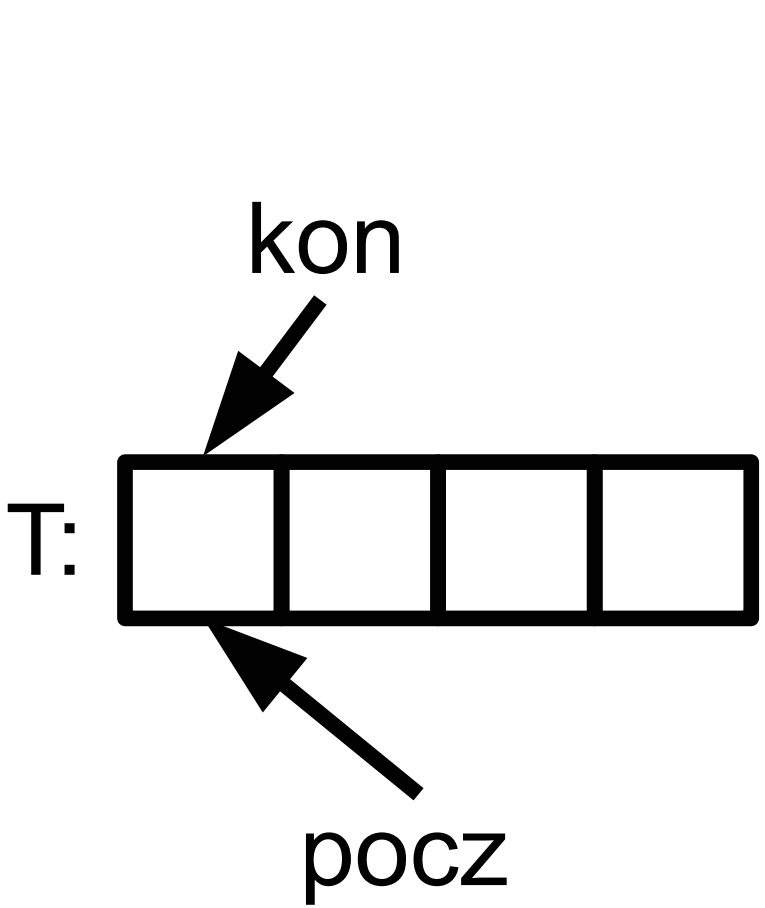
- składniki: tablica T o rozmiarze równym pojemności kolejki + 1 (n), zmienna *pocz* (numer komórki w której kolejka się zaczyna), zmienna *kon* (numer pierwszej wolnej komórki za kolejką), początkowo $pocz = kon = 0$
- `enqueue(`E`)` — włożenie na koniec kolejki elementu E:
  - $T[kon] = E$
  - $kon = (kon + 1)\%n$
- `dequeue()` — pobranie elementu z początku kolejki:
  - $E = T[pocz]$
  - $pocz = (pocz + 1)\%n$
  - zwróć E

Implementacja kolejki w tablicy

- `front()` — wartość elementu na początku kolejki:
  - zwróć $T[pocz]$
- `empty()` — sprawdzenie czy kolejka jest pusta:
  - $kon == pocz$
- `full()` — sprawdzenie czy kolejka jest pełna:
  - $pocz == (kon + 1)\%n$

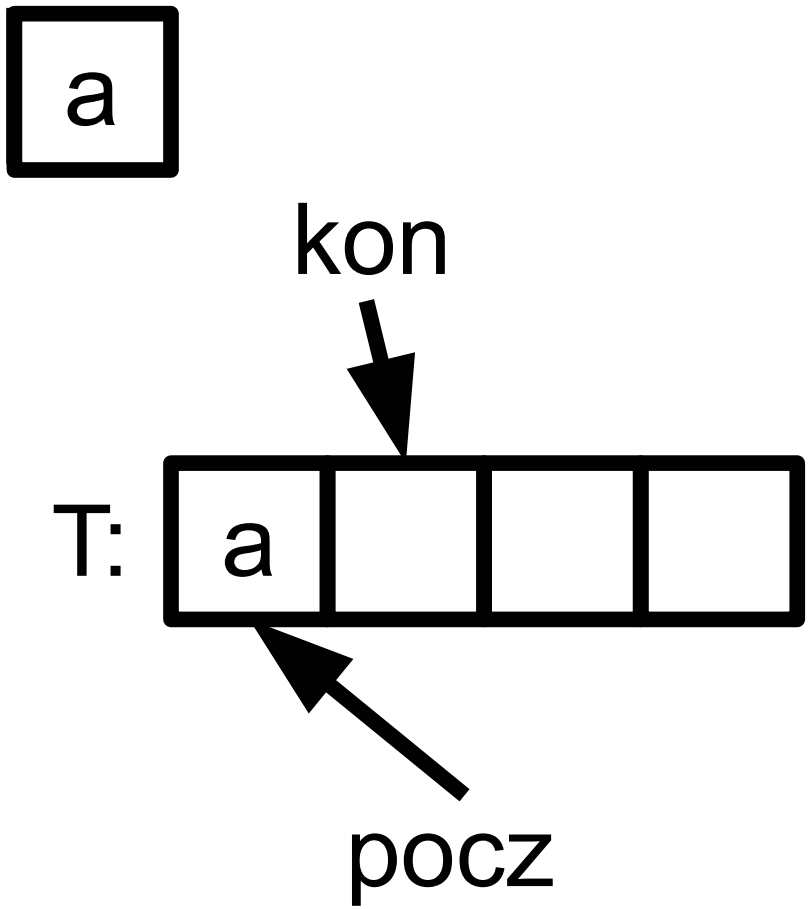
Kolejka

kolejka pusta



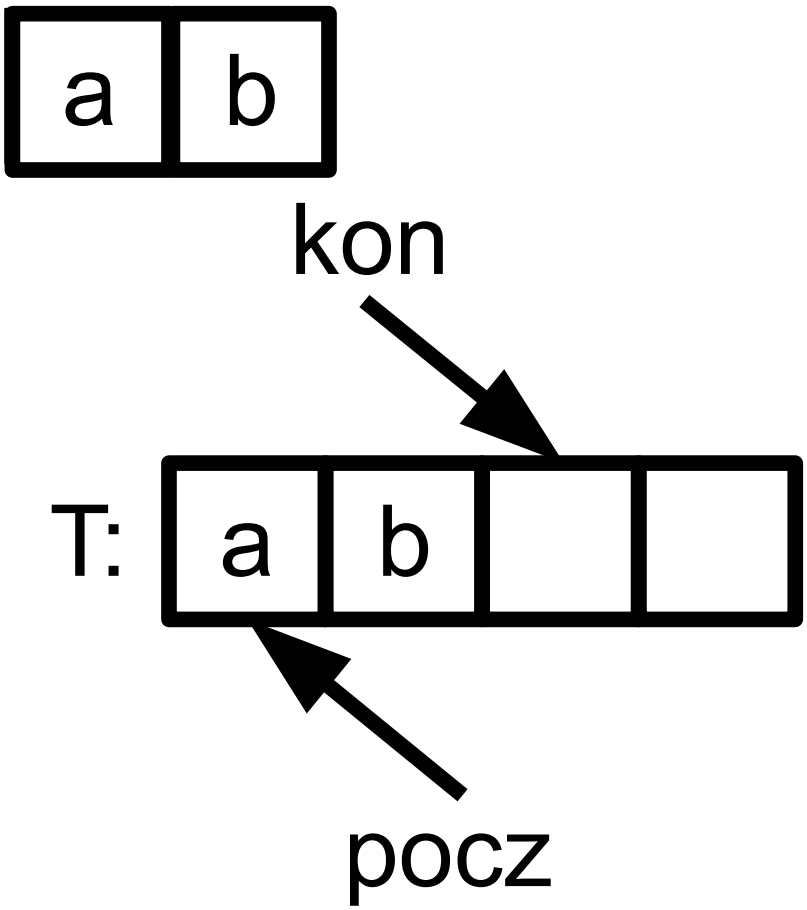
Kolejka

dodaj a do kolejki

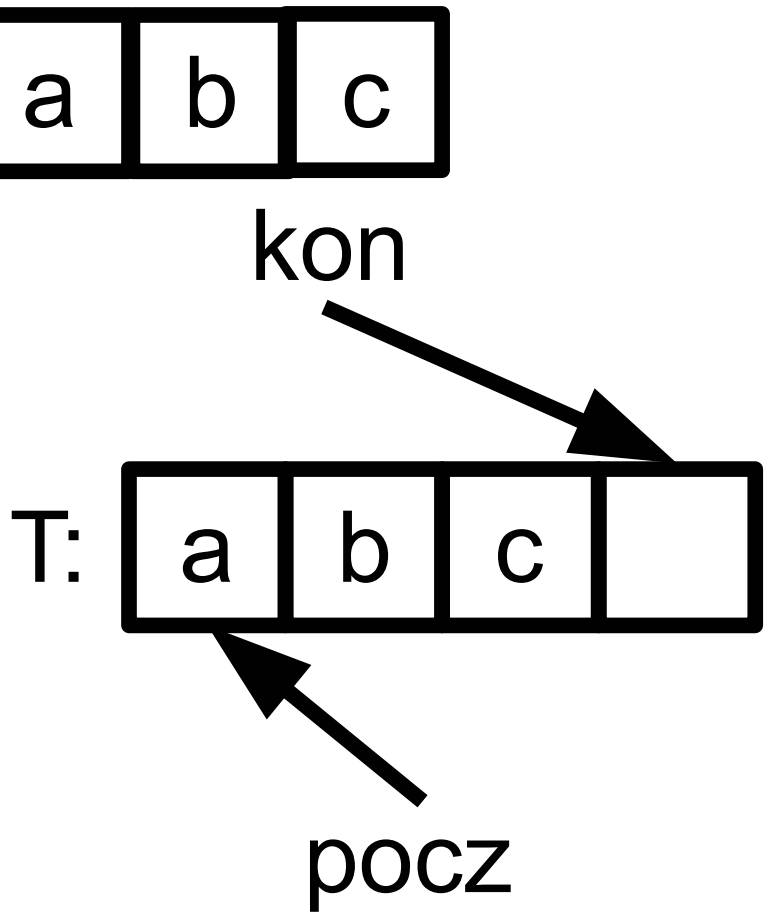


Kolejka

dodaj b do kolejki

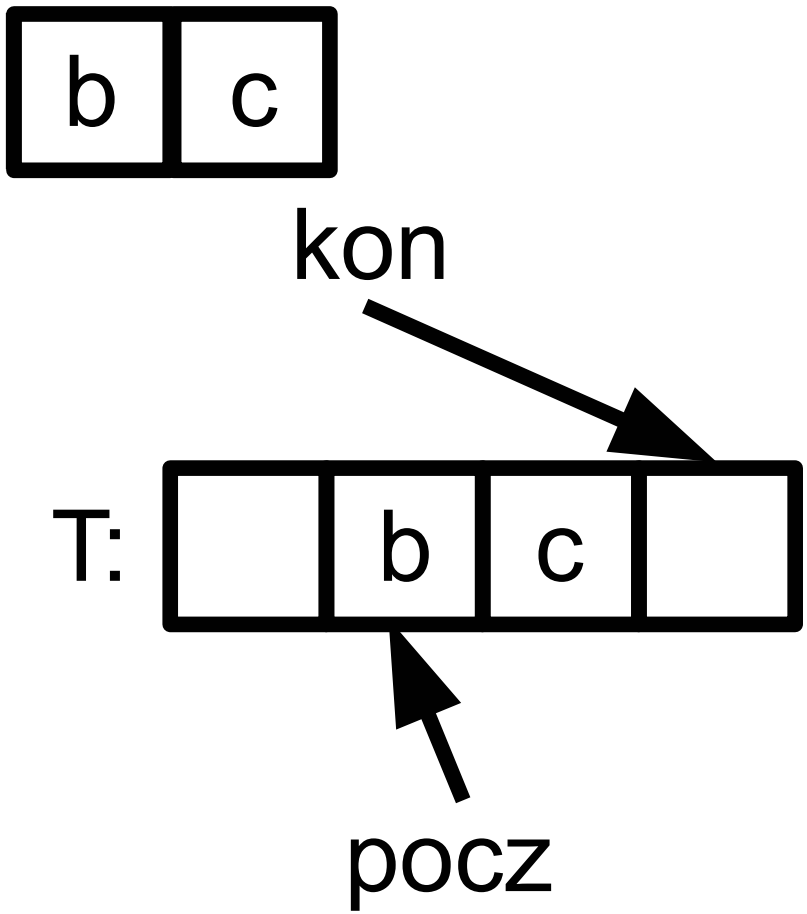


dodaj c do kolejki



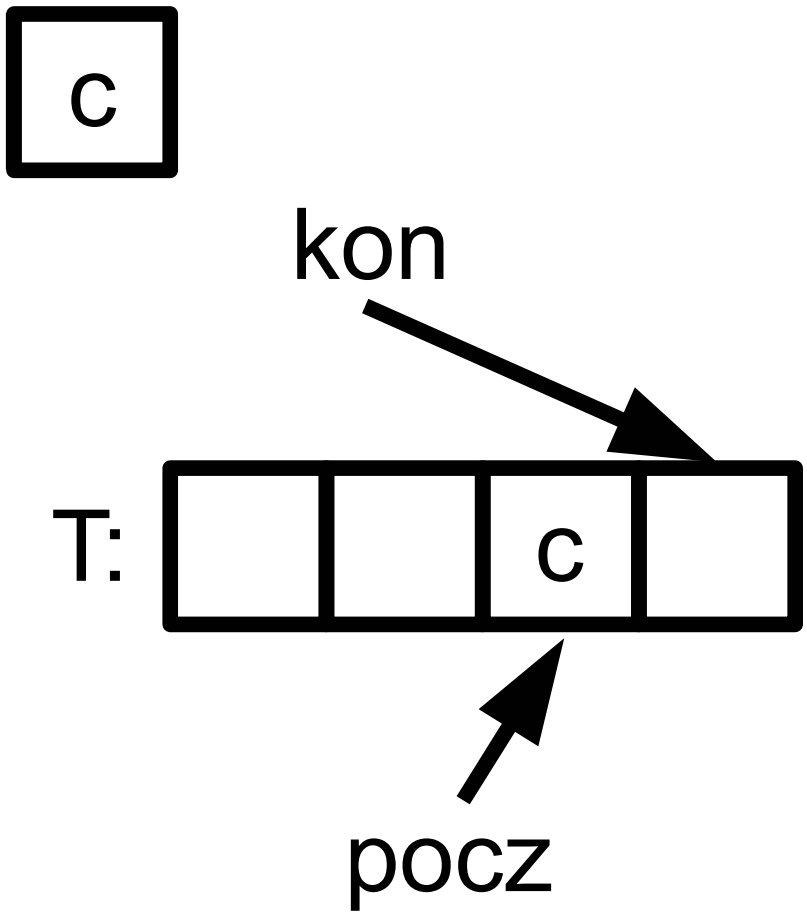
Kolejka

usuń z kolejki $\longrightarrow$ a

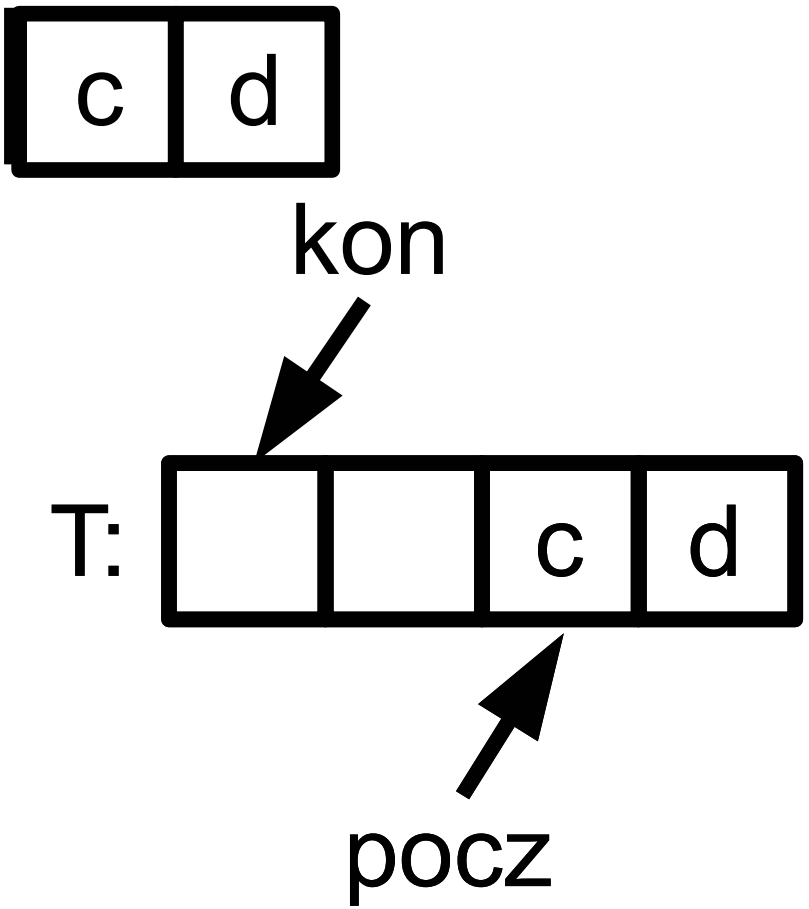


Kolejka

usuń z kolejki $\longrightarrow$ b

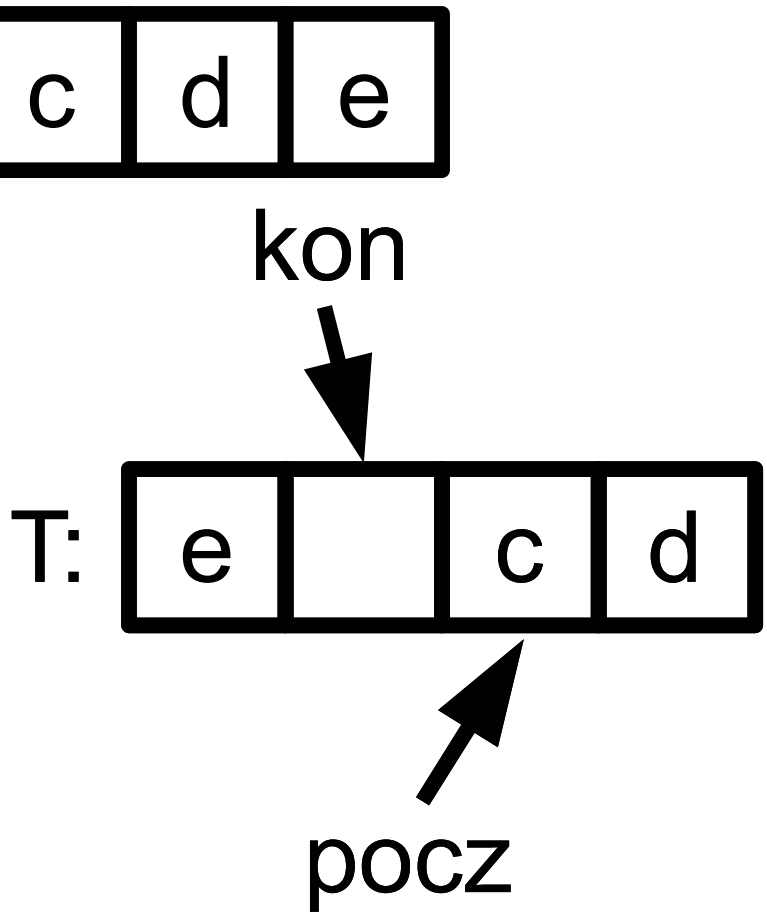


dodaj d do kolejki



Kolejka

dodaj e do kolejki



Odwrotna Notacja Polska

- postfiksowa notacja wyrażeń arytmetycznych (w odróżnieniu od "tradycyjnej" — infiksowej)
- nie wymaga użycia nawiasów ani przypisania operatorom priorytetów
- zamiast notacji:
  pierwszy-argument operator drugi-argument
  posługujemy się zapisem
  pierwszy-argument drugi-argument operator
- np. wyrażenie
  $(a + b) * (x - 7)$
  będzie miało postać
  $a\ b + x\ 7 - *$

Odwrotna Notacja Polska

- obliczenie wartości takiego wyrażenia jest łatwiejsze niż obliczanie wartości wyrażenia infiksowego (między innymi z powodu braku nawiasów)
- przetwarzamy wyrażenie od lewej do prawej: jeżeli kolejny składnik wyrażenia jest liczbą lub zmienną, włóż jej wartość na stos, jeżeli jest operatorem, zdejmij ze stosu argumenty, oblicz operator i włóż wynik na stos
- konwersja wyrażenia z notacji infiksowej na postfiksową także wymaga zastosowania stosu

Konwersja

1: stos $= \phi$
2: **for** każdy symbol S w wyrażeniu infiksowym **do**
3: **if** S jest liczbą lub zmienną **then**
4: przepisz ją na wyjście
5: **else if** $S == '('$ **then**
6: włóż nawias otwierający na stos
7: **else if** $S == ')'$ **then**
8: zdejmuj elementy ze stosu przepisując je na wyjście, aż do napotkania nawiasu otwierającego
9: zdejmij nawias otwierający ze stosu (nie przepisuj go na wyjście)
10: **else**
11: **while** element na szczycie stosu nie jest nawiasem i ma wyższy priorytet niż S **do**
12: przepisz operator ze szczytu stosu na wyjście i zdejmij go ze stosu
13: **end while**
14: włóż symbol na stos
15: **end if**
16: **end for**
17: zdejmij wszystkie elementy ze stosu przepisując je na wyjście

Konwersja — przykład

wejście	wyjście	stos
b*c-(d+e)*4		
*c-(d+e)*4	b	
c-(d+e)*4	b	*
-(d+e)*4	b c	*
(d+e)*4	b c *	-
d+e)*4	b c *	- (
+e)*4	b c * d	- (
e)*4	b c * d	- (+
)*4	b c * d e	- (+
*4	b c * d e +	-
4	b c * d e +	- *
	b c * d e + 4	- *
	b c * d e + 4 * -	