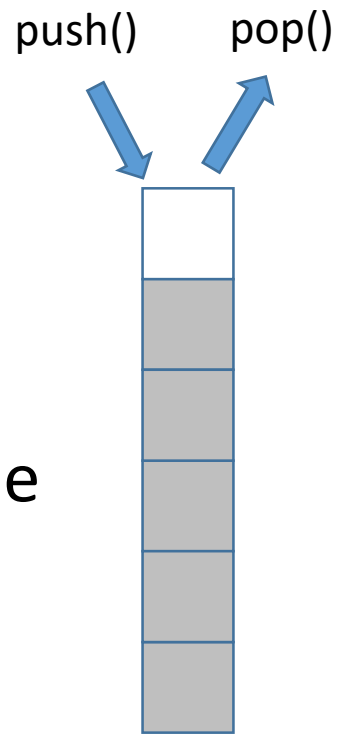
Stos



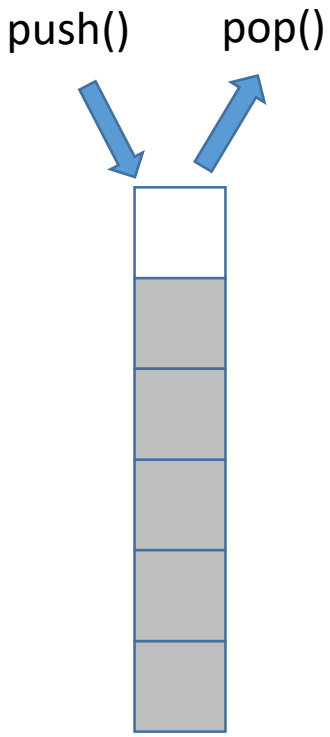
Stos (ang. *Stack*)

Struktura danych przechowująca obiekty, funkcjonująca na zasadzie **LIFO** (**L**ast **I**n **F**irst **O**ut)

Analogia do „tradycyjnego” stosu np. dokumentów na biurku, listów elektronicznych wyświetlonych w programie pocztowym, itp.:

- Nowy element kładziony jest na wierzch istniejącego stosu.
- Do obsługi/analizy brany jest element ze szczytu stosu (czyli najkrócej przebywający na stosie), tym samym opuszczając stos.

Stos



Interfejs klasy **Stos**:

Opis	Metoda
Konstruktor:	`Stos()`, inaczej: `Stack()`
Położenie elementu na stos (na górę):	`poloz()`, inaczej: `push()`
Pobranie elementu ze stosu (z góry):	`zdejmij()`, inaczej: `pop()`
Pobranie rozmiaru stosu:	`rozmiar()`, inaczej: `size()`
Sprawdzenie, czy stos jest pusty:	`czyPusty()`, inaczej: `isEmpty()`

Stos

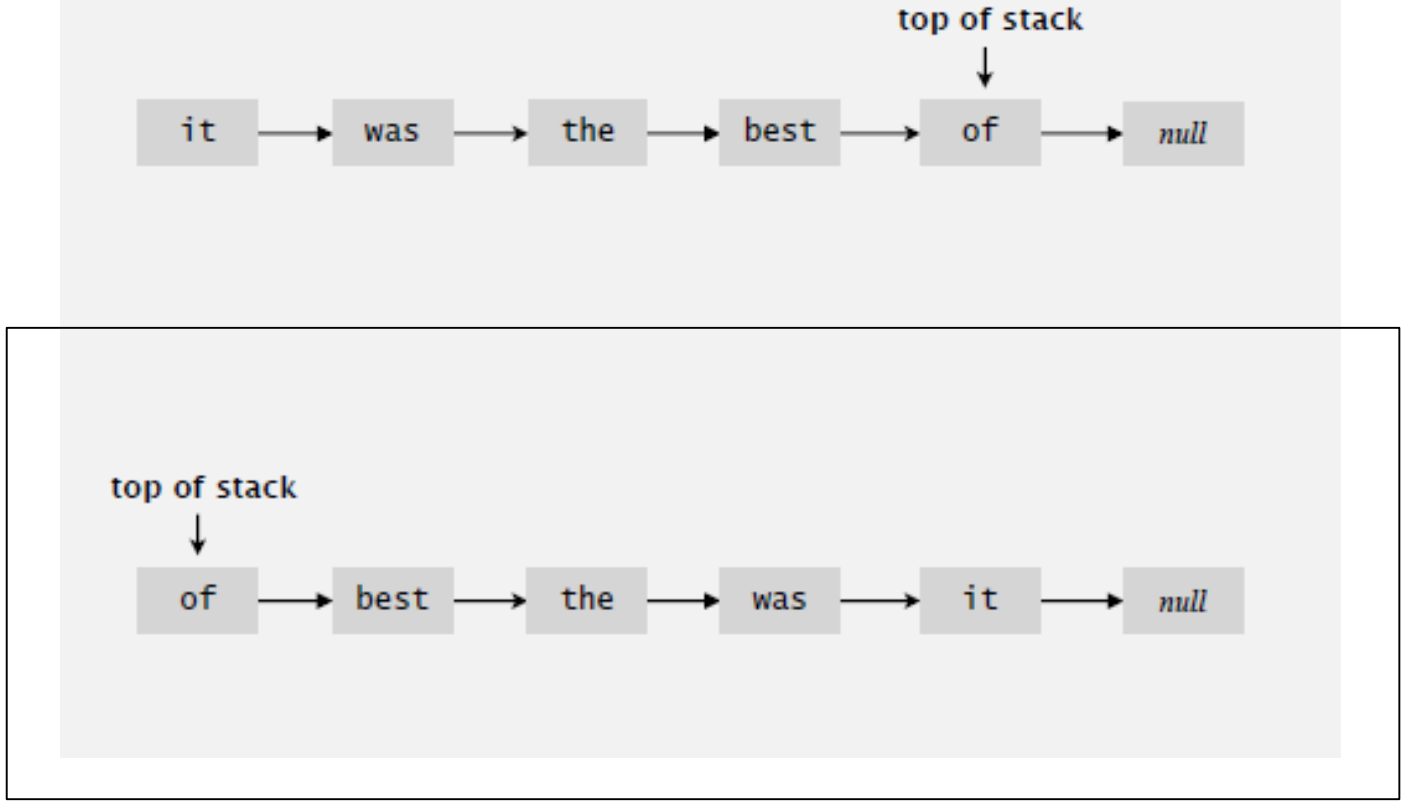
Implementacje:

- Za pomocą listy z dowiązaniami
- Za pomocą tablicy

Stos

Implementacja za pomocą listy

?



[SW-2018]

Stos

Implementacja za pomocą listy:

- Każdy element umieszczany na stosie ma postać:

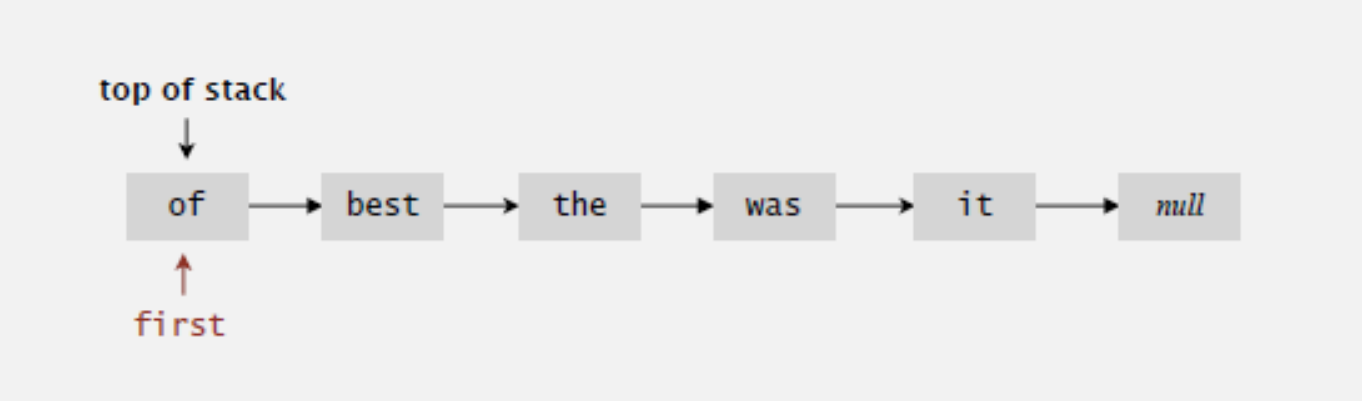
```
private class Node {
    Item item;
    Node next;
}
```

- Stos posiada:
  - głowę – **head** lub **first** (wskaźnik do pierwszego elementu)

Stos

Implementacja za pomocą listy:

- `push()` – nowy element wstawiamy na początek listy (element dotychczas wskazywany przez **first** stanie się drugi, a **first** będzie wskazywało na element wstawiony)
- `pop()` – zwracamy element z początku (wskazywanego dotychczas przez **first**, po wykonaniu operacji wskaźnik **first** będzie wskazywał na obecny drugi element)



[SW-2018]

Stos

Implementacja za pomocą listy:

```
public class Stos <Item> {
   private Node first;
   private int N;

   private class Node {
      Item item;
      Node next;
   }
   …
}
```

Stos

Implementacja za pomocą listy:

- Interfejs:
  - `Stos()`
  - `void push(Item item)`
  - `Item pop()`
  - `int getSize()`
  - `boolean isEmpty()`

-> laboratorium

Stos

Implementacja za pomocą listy:

- Interfejs:
  - `Stos()`
  - `void push(Item item)`
  - `Item pop()`
  - `int getSize()`
  - `boolean isEmpty()`

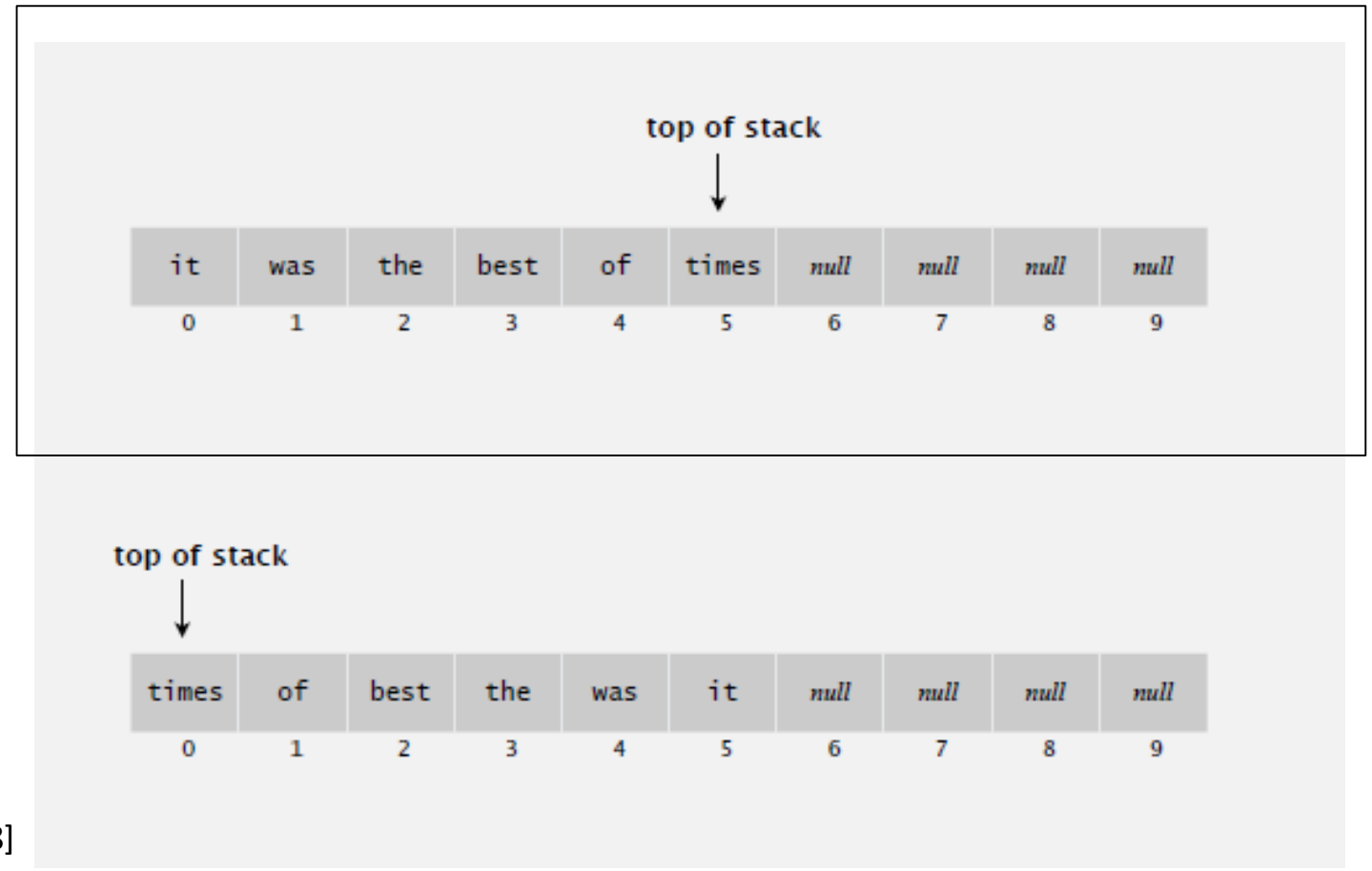
```
public class Stos
{
   private Node first = null;

   private class Node {
      // jak poprzednio
   }

   public boolean isEmpty() {
      return first == null;
   }

   public void push(String item) {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop() {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

Stos

Implementacja za pomocą tablicy

?

- push()
- pop()

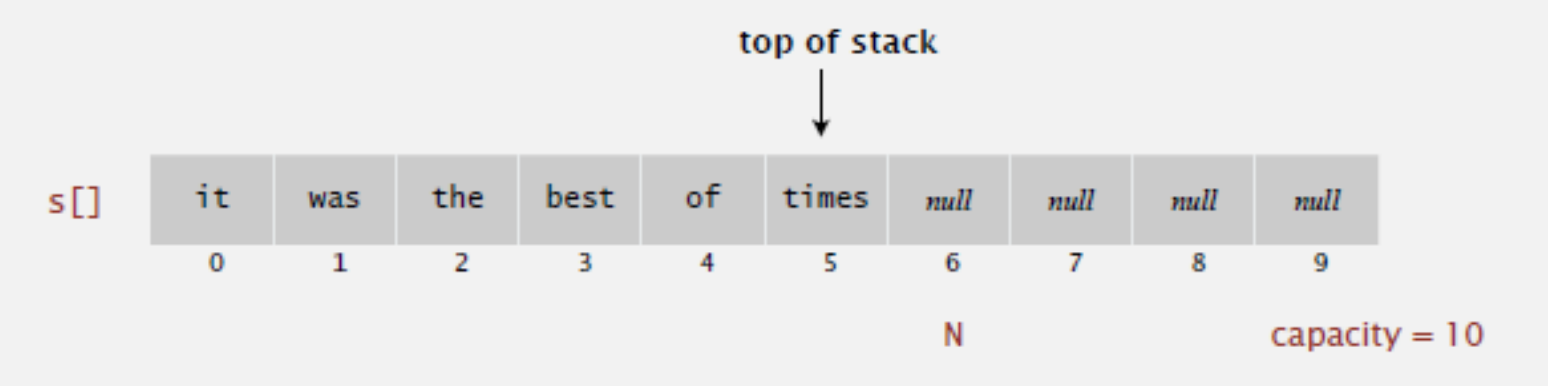


[SW-2018]

Stos

Implementacja za pomocą tablicy

- Wykorzystujemy tablicę S[] do przechowywana *N* elementów.
- push(): dodajemy nowy element na pozycję S[*N*].
- pop(): zdejmujemy/usuwamy element z S[*N*-1].



[SW-2018]

Stos

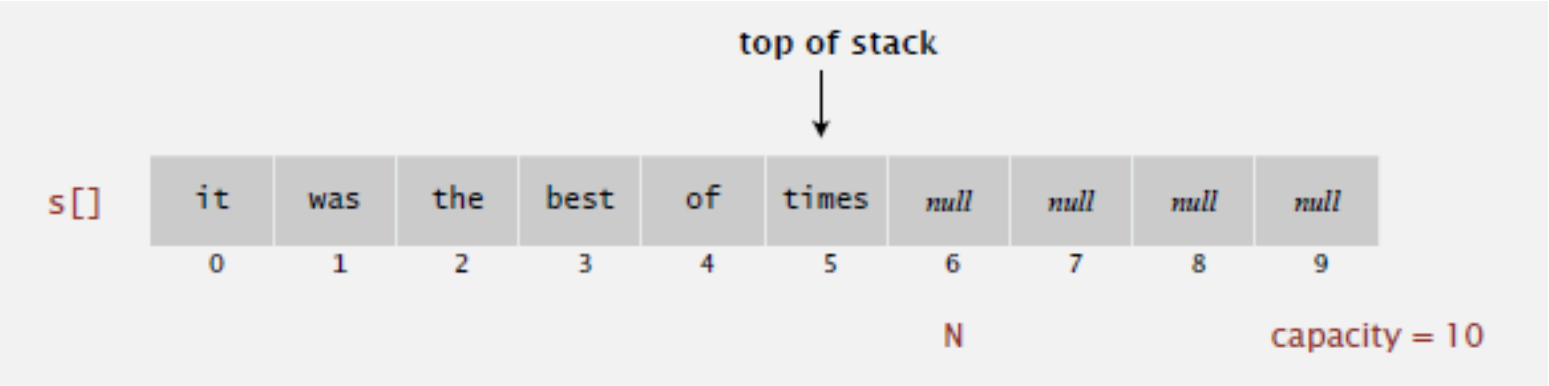
Implementacja za pomocą tablicy

Konieczność znajomości z góry rozmiaru stosu,

Problemy:

- *overflow* – przy dodawaniu kolejnego elementu na stos
- *underflow* – przy pobieraniu elementów ze stosu

- Zmiana rozmiaru tablicy



[SW-2018]

Kolejka/Stos

Implementacja za pomocą tablicy

Zmiana rozmiaru tablicy

Stos:

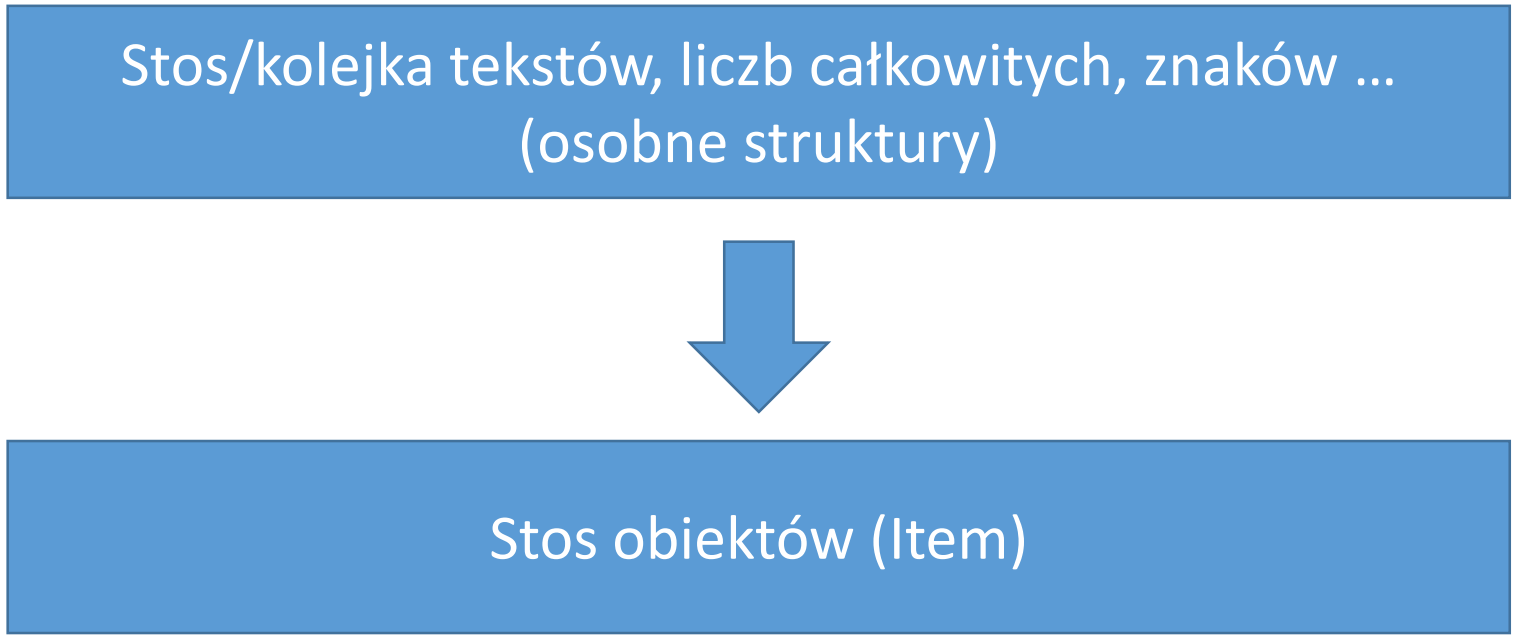
- `push()`: jeśli tablica pełna -> zwiększ rozmiar dwukrotnie.
- `pop()`: jeśli tablica wypełniona w ¼ -> zmniejsz rozmiar tablicy o połowę.

Kolejka

- `enqueue(), dequeue()`:?

Kolejka/Stos

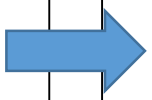
Parametryzowanie kolejki/stosu:



Kolejka/Stos

Parametryzowanie kolejki/stosu:

```
public class Stos
{
   private Node first = null;

   private class Node {
      String item;
      Node next;
   }

   public boolean isEmpty() {
      return first == null;
   }

   public void push(String item) {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop() {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```



```
public class Stos<Item>
{
   private Node first = null;

   private class Node {
      Item item;
      Node next;
   }

   public boolean isEmpty() {
      return first == null;
   }

   public void push(Item item) {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public Item pop() {
      Item item = first.item;
      first = first.next;
      return item;
   }
}
```

Stos

Zastosowania stosu:

- Interpretacja wyrażeń w kompilatorach.
- Operacja Undo w edytorach tekstu
- Przycisk Back w przeglądarkach
- Implementacja rekurencji

Stos

Przykład. Interpretacja wyrażeń w kompilatorach.

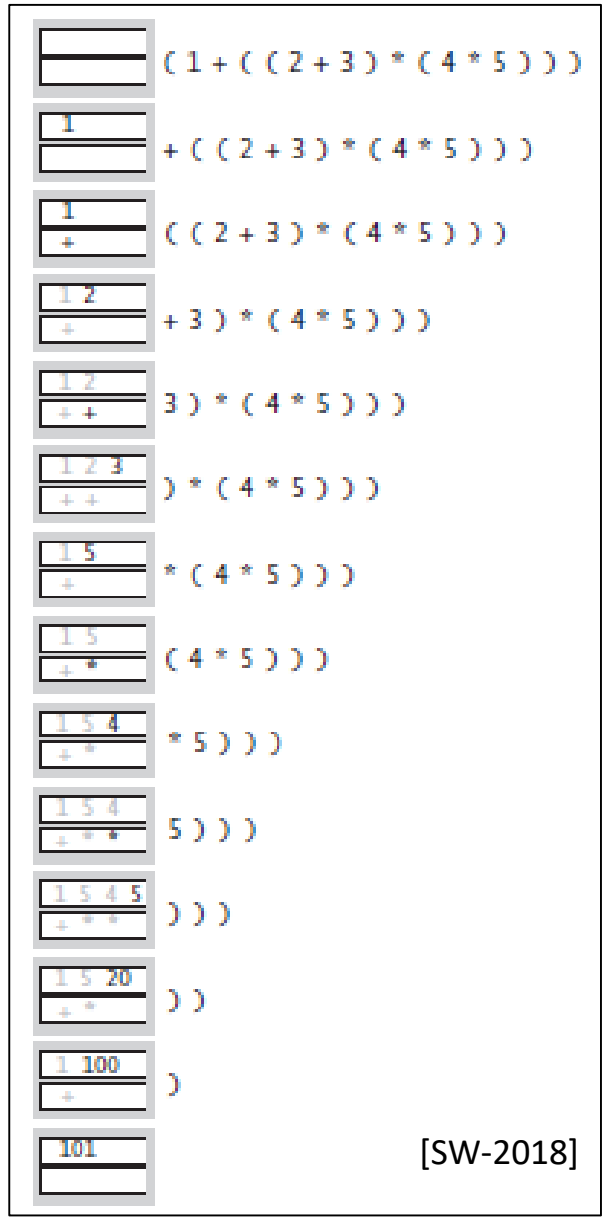
Algorytm E.W. Dijkstry (dwa stosy: osobno dla wartości oraz operatorów)

Wejście: wyrażenie, np. (1+((2+3)*(4*5))), notacja infix

Analizujemy od lewej po jednym elemencie (liczba, ogranicznik, itp.). Gdy napotkamy:

- *Wartość:* dodaj ją na stos wartości
- *Operator:* dodaj go na stos operatorów
- *Lewy nawias:* ignoruj go
- *Prawy nawias:* zdejmij operator i dwie wartości z odpowiednich stosów; połóż rezultat zastosowania operatora do tych wartości na stos wartości

Jeśli badane wyrażenie nie zostało wyczerpane – powtarzamy, a jeśli tak - wartość znajdująca się na stosie wartości jest wynikiem końcowym.



[SW-2018]

Stos

Przykład. Interpretacja wyrażeń w kompilatorach.

Algorytm Jana Łukasiewicza (jeden stos wspólny dla wartości oraz operatorów)

Wejście: wyrażenie, np. (1((23+)(45*)*)+), notacja postfix

Obserwacja: Nawiasy są zbędne -> 123+*45**+ (odwrotna notacja polska ONP – *ang. Polish reverse notation*)

Analizujemy od lewej po jednym elemencie (liczba, ogranicznik, itp.). Gdy napotkamy:

- *Wartość:* dodaj ją na stos wartości
- *Operator:* zdejmij ze stosu odpowiednią dla danego operatora liczbę argumentów, wykonaj na nich obliczenia, a uzyskany wynik połóż na stos.

Jeśli badane wyrażenie nie zostało wyczerpane – powtarzamy, a jeśli tak - wartość znajdująca się na stosie jest wynikiem końcowym.

A jak przekształcić wyrażenie do postaci ONP?