

# Algorytmy i SD

## Struktury danych - Stos

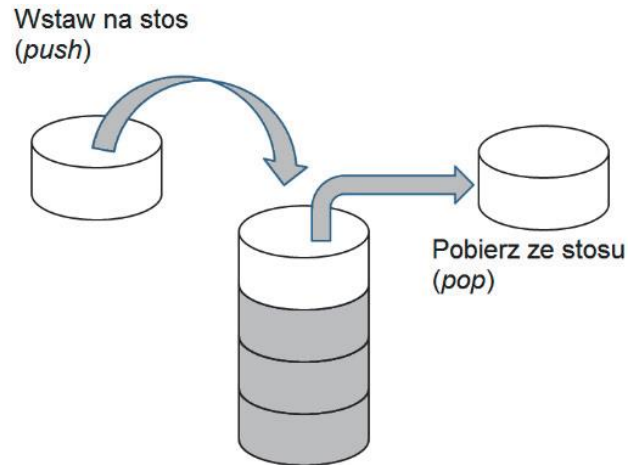


Piotr Ciskowski, Łukasz Jeleń  
Wrocław, 2023

## ADT **stos**:

przechowuje dowolne obiekty

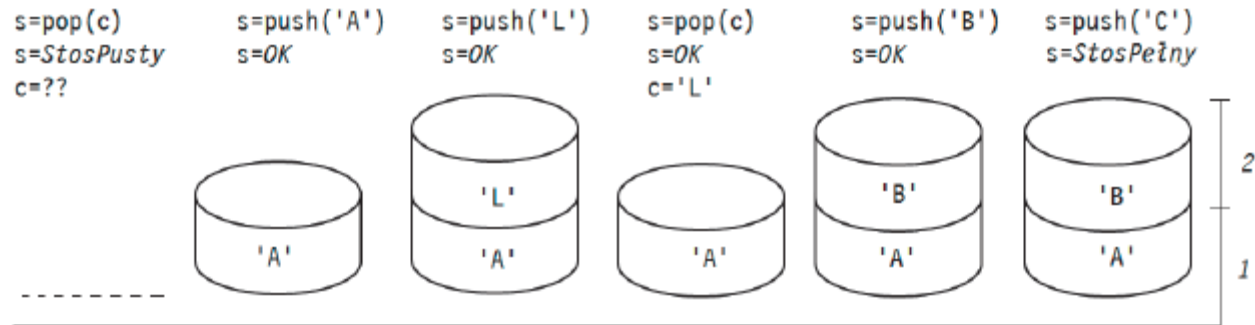
- dodawanie i usuwanie – **LIFO**: *Last In First Out*
- operacje podstawowe:
  - push(element) – dodanie elementu na wierz/wierzchołek/górę
  - element pop() – usunięcie i zwrócenie elementu z wierzchołka
- operacje dodatkowe:
  - element top() – zwraca ostatni element umieszczony na stosie bez jego usuwania
  - integer size() – podaje liczbę przechowywanych elementów
  - boolean isEmpty() – mówi, czy na stosie są przechowywane jakieś elementy



- Jedynym bezpośrednio dostępnym elementem stosu jest jego wierzchołek.
- Po wykonaniu operacji `push(x)` element `x` sam staje się nowym wierzchołkiem stosu, przykrywając poprzedni wierzchołek (jeśli oczywiście coś na stosie już było).
- Próba wstawienia czegoś na pełny stos powinna zakończyć się błędem.
- Próba pobrania elementu z pustego stosu powinna zakończyć się błędem.

## ADT **stos**:

- wyjątki - błędy:
  - pop() i top()
    - nie mogą być wykonane, jeśli stos jest pusty
    - próba wywołania pop() lub top() dla pustego stosu wyrzuci wyjątek *EmptyStackException*



## ADT **stos**:

- zastosowania:
  - bezpośrednio
    - historia odwiedzanych stron w przeglądarce
    - sekwencja operacji „Cofnij”
    - łańcuch wywołania metod w wirtualnej maszynie Javy
  - pośrednio
    - struktura pomocnicza dla algorytmów
    - składowa innych struktur danych

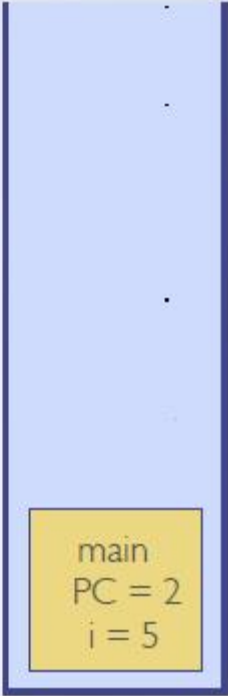
## STOS WYWOŁAŃ W C++

- C++ kontroluje łańcuch wywołań funkcji za pomocą stosu
- Kiedy jakaś funkcja jest wywoływana, system dodaje do stosu ramkę zawierającą:
  - Zmienne lokalne i wartość zwracaną z funkcji
  - Licznik programu kontrolujący wywoływane
- Jeśli funkcja kończy swoje działanie, to ramka jest pobierana ze stosu (pop()), a kontrola jest przekazywana do metody na górze stosu.
- Pozwala na stosowanie rekurencji

```
main() {  
    int i = 5;  
    foo(i);  
}
```

```
foo(int j) {  
    int k;  
    k = j + 1;  
    bar(k);  
}
```

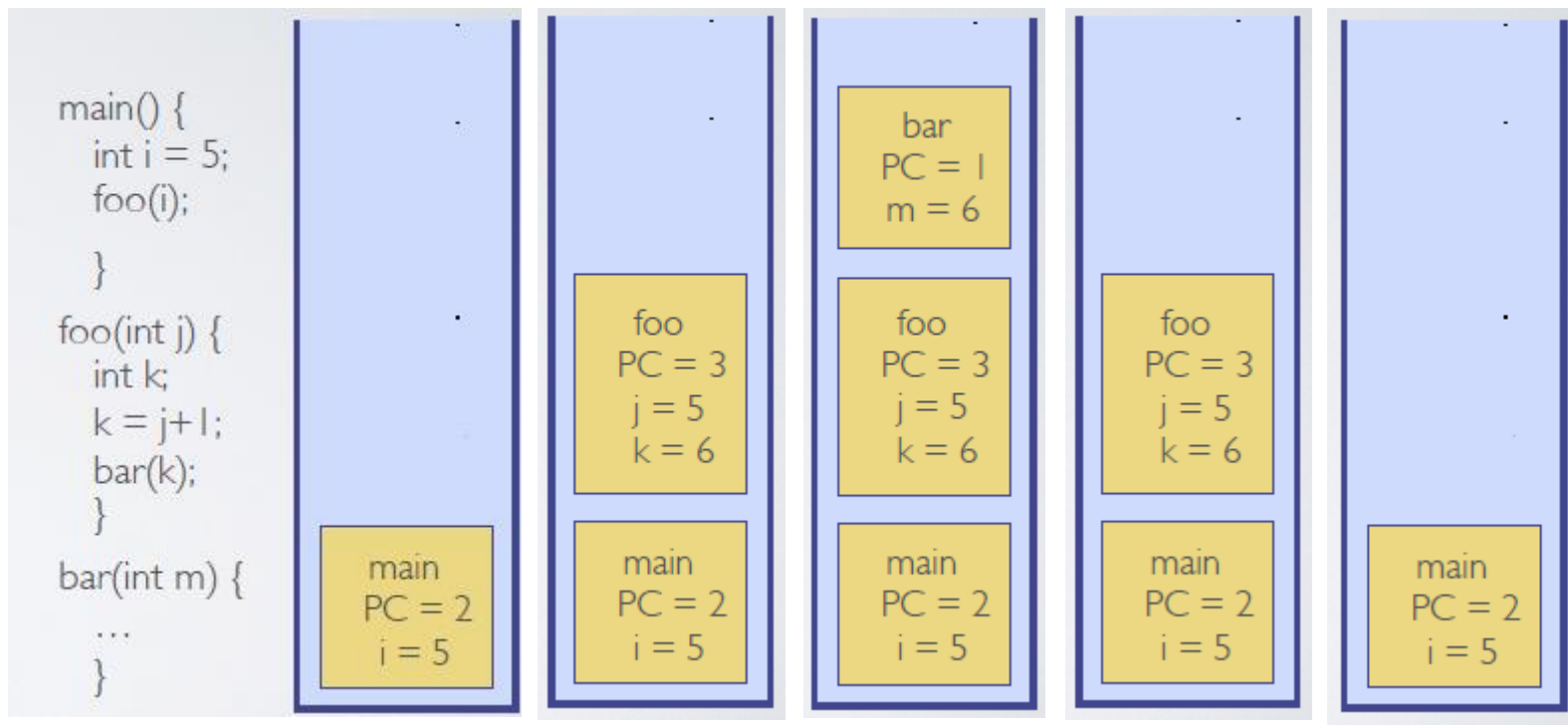
```
bar(int m) {  
    ...  
}
```



main  
PC = 2  
i = 5

© 2004 Goodrich, Tamassia

## STOS WYWOŁAŃ W C++





## stos bazujący na tablicy:

- prosta implementacja
- dodajemy elementy od prawej do lewej
- dodatkowa zmienna – kontroluje indeks elementu na wierzchu stosu



```
Algorytm size()
return t + 1

Algorytm pop()
if isEmpty() then
    throw EmptyStackException
else
    t ← t - 1
    return S[t + 1]
```

## stos bazujący na tablicy:

- tablica może się przepełnić
- wtedy operacja push() powinna wyrzucić wyjątek FullStackExeption



```
Algorytm push(e)
  if t = S.length - 1 then
    throw FullStackExeption
  else
    t ← t + 1
    S[t] ← e
```

### **stos** bazujący na tablicy:

- wydajność – dla  $n$  elementów na stosie:
  - wykorzystane miejsce:  $O(n)$
  - każda operacja działa w czasie:  $O(1)$
- ograniczenia:
  - maksymalny rozmiar stosu musi zostać zdefiniowany a priori i nie może zostać zmieniony
  - próba dodania nowego elementu do pełnego stosu powoduje wyjątki – zależne od implementacji

## **stos** bazujący na powiększanej tablicy:

- podczas wykonywania operacji push() zamiast od razu wyrzucać wyjątek można spróbować poszukać miejsca w pamięci na większą tablicę
- trochę pracochłonne
  - trzeba znaleźć miejsce, skopiować stare elementy, skasować starą tablicę, dopisać nowy element
- jak duża ta nowa?
  - strategia inkrementalna
    - zawsze dodawaj tyle samo elementów
  - strategia podwajania
    - zawsze podwajaj rozmiar

```
Algorytm push(e)
  if t = S.length - 1 then
    A ← nowa tablica o rozmiarze...
    for i ← 0 to t do
      A[i] ← S[i]
    S ← A
  t ← t + 1
  S[t] ← e
```

## porównanie strategii – inkrementacja vs. podwajanie

- zaczynamy od pustego stosu – tablica o rozmiarze 1
- inkrementacja – dodajemy  $c$  elementów
- podwajanie – wiadomo
- całkowity czas  $T(n)$  – całkowity czas operacji  $\text{push}()$   
niezbędny do wykonania serii  $n$  operacji  $\text{push}()$
- średni czas  $T(n)/n$  – średni czas operacji  $\text{push}()$   
niezbędny do wykonania serii operacji

### STRATEGIA INKREMENTALNA

- Tablica zostanie zastąpiona  $k = n/c$  razy
- Całkowity czas  $T(n)$  wykonania  $n$  operacji  $\text{push}$  jest proporcjonalny do:

$$n + c + 2c + 3c + 4c + \dots + kc =$$

$$n + c(1 + 2 + 3 + \dots + k) =$$

$$n + ck(k+1)/2$$

- Ponieważ  $c$  jest stałą,  $T(n)$  jest w  $O(n + k^2)$ , tj.  $O(n^2)$
- Średni czas operacji  $\text{push}$  jest w  $O(n)$

© 2004 Goodrich, Tamassia

### STRATEGIA PODWAJANIA

- Tablica zostanie zastąpiona  $k = \log_2 n$  razy
- Całkowity czas  $T(n)$  wykonania  $n$  operacji  $\text{push}$  jest proporcjonalny do:

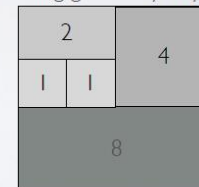
$$n + 1 + 2 + 4 + 8 + \dots + 2^k =$$

$$n + 2^{k+1} - 1 = 2n - 1$$

- $T(n)$  jest w  $O(n)$
- Średni czas operacji  $\text{push}$  jest w  $O(1)$

© 2004 Goodrich, Tamassia

ciąg geometryczny



## interfejs stosu w C++

- interfejs – tylko definicja operacji  
– bez szczegółów

```
template <typename Object>
class Stack{
public:
    int size();
    bool isEmpty();
    Object& top()
        throw (EmptyStackException);
    void push(Object o);
    Object pop()
        throw (EmptyStackException);
}
```

## przykład stosu opartego na tablicy – w C++

```
template <typename Object>
class TabStack{
private:
    int pojemnosc; //pojemność stosu
    Object *S //tablica stosu
    int top //góra stosu
public:
    TabStack(int p){
        pojemnosc = p;
        S = new Object[pojemnosc];
        top = -1;
    }
```

```
    bool isEmpty()
        {return (top<0);}
    Object pop()
        throw (EmptyStackException){
        if (isEmpty())
            throw EmptyStackException
            ("Dostęp do pustego stosu");
        return S[top--];
    }
    // ... pozostałe funkcje ominięte
```

## przykład stosu opartego na liście – w Pythonie

```
class StosOgraniczony:
    def __init__(self, pRozmiar):
        self._stos = list()          # Właściwa kolekcja danych
        self._MaxElt=pRozmiar        # Maksymalny rozmiar stosu

    def zeruj(self):    # Zerowanie stosu
        self._stos.clear()

    def wypisz(self, s):
        print(s)
        if self._stos!=None:
            print(" Zawartość stosu: [", end=" ")
            for x in self._stos:      # Wywołajmy iterator klasy list()
                print(x, end=" ")
            print("]")

    def push(self, obj):
        print("Odkładam: ", str(obj)) # Konwersja na postać tekstową
        if len(self._stos) < self._MaxElt:
            self._stos.append(obj)    # Dokładamy kolejny element na koniec
        else:
            print("** POJEMNOŚĆ PRZEKROCZONA **")

    def pop(self):
        if len(self._stos_)>0:
            tmp=self._stos.pop()      # Pobiera ostatni element
            return tmp                # Usuwamy ze stosu, ale nie tracimy dostępu do elementu usuwanego
```



## przykład stosu opartego na liście – w Pythonie

```
from MojeTypy import StosOgraniczony as s
x=s.StosOgraniczony(2)
x.wypisz("Zawartość stosu")
x.push(2)
x.push('A')
x.push("małe co nieco")
x.wypisz("Zawartość stosu")
```

Oto wyniki naszego programu:

```
Zawartość stosu
Zawartość stosu: [ ]
Odkładam: 2
Odkładam: A
Odkładam:  małe co nieco
* POJEMNOŚĆ PRZEKROCZONA *
Zawartość stosu
Zawartość stosu: [2 A]
```