Algorytmy i struktury danych - struktury danych

Marcin Żurowski

23 kwietnia 2026

Plan zajęć

Struktury danych - podział

- statyczne
- dynamiczne

Struktury danych - podział

- statyczne
- dynamiczne

Struktury statyczne

- Struktury statyczne:
  - zmienna
  - tablica
  - tablica wielowymiarowa
- Zalety:
- Wady:

Struktury statyczne

- Struktury statyczne:
  - zmienna
  - tablica
  - tablica wielowymiarowa
- Zalety:
- Wady:

Struktury statyczne

- Struktury statyczne:
  - zmienna
  - tablica
  - tablica wielowymiarowa
- Zalety:
  - tworzenie w czasie stałym $O(1)$
  - wstawianie i wyjmowanie elementu w czasie stałym $O(1)$
- Wady:

Struktury statyczne

- Struktury statyczne:
  - zmienna
  - tablica
  - tablica wielowymiarowa
- Zalety:
  - tworzenie w czasie stałym $\Theta(1)$
  - wstawianie i wyjmowanie elementu w czasie stałym $\Theta(1)$
- Wady:

Struktury statyczne

- Struktury statyczne:
  - zmienna
  - tablica
  - tablica wielowymiarowa
- Zalety:
  - tworzenie w czasie stałym $\Theta(1)$
  - wstawianie i wyjmowanie elementu w czasie stałym $\Theta(1)$
- Wady:

Struktury statyczne

- Struktury statyczne:
  - zmienna
  - tablica
  - tablica wielowymiarowa
- Zalety:
  - tworzenie w czasie stałym $\Theta(1)$
  - wstawianie i wyjmowanie elementu w czasie stałym $\Theta(1)$
- Wady:
  - nie może się powiększać ani zmniejszać
  - aby wyszukać dany element trzeba przejrzeć całą tablicę $\Theta(n)$

Struktury statyczne

- Struktury statyczne:
  - zmienna
  - tablica
  - tablica wielowymiarowa
- Zalety:
  - tworzenie w czasie stałym $\Theta(1)$
  - wstawianie i wyjmowanie elementu w czasie stałym $\Theta(1)$
- Wady:
  - nie może się powiększać ani zmniejszać
  - aby wyszukać dany element trzeba przejrzeć całą strukturę $\Theta(n)$

Struktury statyczne

- Struktury statyczne:
  - zmienna
  - tablica
  - tablica wielowymiarowa
- Zalety:
  - tworzenie w czasie stałym $\Theta(1)$
  - wstawianie i wyjmowanie elementu w czasie stałym $\Theta(1)$
- Wady:
  - nie może się powiększać ani zmniejszać
  - aby wyszukać dany element trzeba przejrzeć całą strukturę $\Theta(n)$

Struktury statyczne

- Struktury statyczne:
  - zmienna
  - tablica
  - tablica wielowymiarowa
- Zalety:
  - tworzenie w czasie stałym $\Theta(1)$
  - wstawianie i wyjmowanie elementu w czasie stałym $\Theta(1)$
- Wady:
  - nie może się powiększać ani zmniejszać
  - aby wyszukać dany element trzeba przejrzeć całą strukturę $\Theta(n)$

Struktury statyczne

- Struktury statyczne:
  - zmienna
  - tablica
  - tablica wielowymiarowa
- Zalety:
  - tworzenie w czasie stałym $\Theta(1)$
  - wstawianie i wyjmowanie elementu w czasie stałym $\Theta(1)$
- Wady:
  - nie może się powiększać ani zmniejszać
  - aby wyszukać dany element trzeba przejrzeć całą strukturę $\Theta(n)$

Struktury dynamiczne

- Struktury dynamiczne
  - stos
  - kolejka
  - lista z dowiązaniami
  - kopiec binarny
  - drzewo poszukiwań binarnych (BST)
- Zalety:
- Wady:

Struktury dynamiczne

- Struktury dynamiczne
  - stos
  - kolejka
  - lista z dowiązaniami
  - kopiec binarny
  - drzewo poszukiwań binarnych (BST)
- Zalety:
- Wady:

Struktury dynamiczne

- Struktury dynamiczne
  - stos
  - kolejka
  - lista z dowiązaniami
  - kopiec binarny
  - drzewo poszukiwań binarnych (BST)
- Zalety:
- Wady:

Struktury dynamiczne

- Struktury dynamiczne
  - stos
  - kolejka
  - lista z dowiązaniami
  - kopiec binarny
  - drzewo poszukiwań binarnych (BST)
- Zalety:
- Wady:

Struktury dynamiczne

- Struktury dynamiczne
  - stos
  - kolejka
  - lista z dowiązaniami
  - kopiec binarny
  - drzewo poszukiwań binarnych (BST)
- Zalety:
  - można się powiększać i zmniejszać
  - aby wyszukać dany element nie trzeba przejść przez wszystkich elementów
- Wady:

Struktury dynamiczne

- Struktury dynamiczne
  - stos
  - kolejka
  - lista z dowiązaniami
  - kopiec binarny
  - drzewo poszukiwań binarnych (BST)
- Zalety:
  - może się powiększać i zmniejszać
  - aby wyszukać dany element nie trzeba przeglądać wszystkich elementów
- Wady:

Struktury dynamiczne

- Struktury dynamiczne
  - stos
  - kolejka
  - lista z dowiązaniami
  - kopiec binarny
  - drzewo poszukiwań binarnych (BST)
- Zalety:
  - może się powiększać i zmniejszać
  - aby wyszukać dany element nie trzeba przeglądać wszystkich elementów
- Wady:

Struktury dynamiczne

- Struktury dynamiczne
  - stos
  - kolejka
  - lista z dowiązaniami
  - kopiec binarny
  - drzewo poszukiwań binarnych (BST)
- Zalety:
  - może się powiększać i zmniejszać
  - aby wyszukać dany element nie trzeba przeglądać wszystkich elementów
- Wady:
  - wstawianie i usuwanie elementu zajmuje więcej czasu
  - implementacja jest skomplikowana

Struktury dynamiczne

- Struktury dynamiczne
  - stos
  - kolejka
  - lista z dowiązaniami
  - kopiec binarny
  - drzewo poszukiwań binarnych (BST)
- Zalety:
  - może się powiększać i zmniejszać
  - aby wyszukać dany element nie trzeba przeglądać wszystkich elementów
- Wady:
  - wstawianie i wyjmowanie elementu zajmuje więcej czasu
  - implementacja jest skomplikowana

Struktury dynamiczne

- Struktury dynamiczne
  - stos
  - kolejka
  - lista z dowiązaniami
  - kopiec binarny
  - drzewo poszukiwań binarnych (BST)
- Zalety:
  - może się powiększać i zmniejszać
  - aby wyszukać dany element nie trzeba przeglądać wszystkich elementów
- Wady:
  - wstawianie i wyjmowanie elementu zajmuje więcej czasu
  - implementacja jest skomplikowana

Struktury dynamiczne

- Struktury dynamiczne
  - stos
  - kolejka
  - lista z dowiązaniami
  - kopiec binarny
  - drzewo poszukiwań binarnych (BST)
- Zalety:
  - może się powiększać i zmniejszać
  - aby wyszukać dany element nie trzeba przeglądać wszystkich elementów
- Wady:
  - wstawianie i wyjmowanie elementu zajmuje więcej czasu
  - implementacja jest skomplikowana

Struktury dynamiczne

- Struktury dynamiczne
  - stos
  - kolejka
  - lista z dowiązaniami
  - kopiec binarny
  - drzewo poszukiwań binarnych (BST)
- Zalety:
  - może się powiększać i zmniejszać
  - aby wyszukać dany element nie trzeba przeglądać wszystkich elementów
- Wady:
  - wstawianie i wyjmowanie elementu zajmuje więcej czasu
  - implementacja jest skomplikowana

Struktury dynamiczne

- Element zbioru dynamicznego reprezentowany jest w strukturze danych przez obiekt posiadający pewne atrybuty
- Atrybuty obiektu x można odczytywać lub modyfikować, o ile dany jest wskaźnik do tego obiektu. Uwaga: w pseudokodzie utożsamiamy wskaźnik do elementu x z samym elementem x
- Element x dynamicznej struktury danych może składać się z klucza x.key (wartości przechowywanej przez element x) oraz pól dodatkowych
- Jeśli klucze elementów należą do zbioru liniowo uporządkowanego, można zdefiniować pojęcie najmniejszego i największego elementu w zbiorze oraz poprzednika i następnika elementu x

Struktury dynamiczne

- Element zbioru dynamicznego reprezentowany jest w strukturze danych przez obiekt posiadający pewne atrybuty
- Atrybuty obiektu x można odczytywać lub modyfikować, o ile dany jest wskaźnik do tego obiektu. Uwaga: w pseudokodzie utożsamiamy wskaźnik do elementu x z samym elementem x
- Element x dynamicznej struktury danych może składać się z klucza `x.key` (wartości przechowywanej przez element x) oraz pól dodatkowych
- Jeśli klucze elementów należą do zbioru liniowo uporządkowanego, można zdefiniować pojęcie najmniejszego i największego elementu w zbiorze oraz poprzednika i następnika elementu x

Struktury dynamiczne

- Element zbioru dynamicznego reprezentowany jest w strukturze danych przez obiekt posiadający pewne atrybuty
- Atrybuty obiektu x można odczytywać lub modyfikować, o ile dany jest wskaźnik do tego obiektu. Uwaga: w pseudokodzie utożsamiamy wskaźnik do elementu x z samym elementem x
- Element x dynamicznej struktury danych może składać się z klucza `x.key` (wartości przechowywanej przez element x) oraz pól dodatkowych
- Jeśli klucze elementów należą do zbioru liniowo uporządkowanego, można zdefiniować pojęcie najmniejszego i największego elementu w zbiorze oraz poprzednika i następnika elementu x

Struktury dynamiczne

- Element zbioru dynamicznego reprezentowany jest w strukturze danych przez obiekt posiadający pewne atrybuty
- Atrybuty obiektu x można odczytywać lub modyfikować, o ile dany jest wskaźnik do tego obiektu. Uwaga: w pseudokodzie utożsamiamy wskaźnik do elementu x z samym elementem x
- Element x dynamicznej struktury danych może składać się z klucza `x.key` (wartości przechowywanej przez element x) oraz pól dodatkowych
- Jeśli klucze elementów należą do zbioru liniowo uporządkowanego, można zdefiniować pojęcie najmniejszego i największego elementu w zbiorze oraz poprzednika i następnika elementu x

Operacje na dynamicznych strukturach danych

- modyfikujące
  - `INSERT(S,x)` - dodaje do zbioru S element x
  - `DELETE(S,x)` - usuwa ze zbioru S element x
- zapytania

Czas wykonania określonej operacji zależy od rozmiaru i rodzaju struktury.

Operacje na dynamicznych strukturach danych

- modyfikujące
  - `INSERT(S,x)` - dodaje do zbioru S element x
  - `DELETE(S,x)` - usuwa ze zbioru S element x
- zapytania
  - `SEARCH(S,k)` - dla zbioru S i klucza k zwraca wskaźnik do elementu x (lub NIL, gdy taki element nie istnieje)
  - `SUCCESSOR(S,x)` - dla elementu x zbioru S zwraca wskaźnik do jego następnika
  - `MINIMUM(S)` - dla zbioru S zwraca wskaźnik do elementu o najmniejszym kluczu k (lub NIL w przypadku gdy S jest zbiorem pustym)

Czas wykonania określonej operacji zależy od rozmiaru i rodzaju struktury.

Operacje na dynamicznych strukturach danych

- modyfikujące
  - `INSERT(S,x)` - dodaje do zbioru S element x
  - `DELETE(S,x)` - usuwa ze zbioru S element x
- zapytania
  - `SEARCH(S,k)` - dla zbioru S i klucza k zwraca wskaźnik do elementu x (lub `NIL` gdy taki element nie istnieje)
  - `SUCCESSOR(S,x)` - dla elementu x zbioru S zwraca wskaźnik do jego następnika
  - `MINIMUM(S)` - dla zbioru S zwraca wskaźnik do elementu o najmniejszym kluczu k (lub `NIL` w przypadku gdy S jest zbiorem pustym)

Czas wykonania określonej operacji zależy od rozmiaru i rodzaju struktury.

Operacje na dynamicznych strukturach danych

- modyfikujące
  - `INSERT(S,x)` - dodaje do zbioru S element x
  - `DELETE(S,x)` - usuwa ze zbioru S element x
- zapytania
  - `SEARCH(S,k)` - dla zbioru S i klucza k zwraca wskaźnik do elementu x (lub `NIL` gdy taki element nie istnieje)
  - `SUCCESSOR(S,x)` - dla elementu x zbioru S zwraca wskaźnik do jego następnika
  - `MINIMUM(S)` - dla zbioru S zwraca wskaźnik do elementu o najmniejszym kluczu k (lub *NIL* w przypadku gdy *S* jest zbiorem pustym)

Czas wykonania określonej operacji zależy od rozmiaru i rodzaju struktury.

Operacje na dynamicznych strukturach danych

- modyfikujące
  - `INSERT(S,x)` - dodaje do zbioru `S` element `x`
  - `DELETE(S,x)` - usuwa ze zbioru `S` element `x`
- zapytania
  - `SEARCH(S,k)` - dla zbioru `S` i klucza `k` zwraca wskaźnik do elementu `x` (lub `NIL` gdy taki element nie istnieje)
  - `SUCCESSOR(S,x)` - dla elementu `x` zbioru `S` zwraca wskaźnik do jego następnika
  - `MINIMUM(S)` - dla zbioru `S` zwraca wskaźnik do elementu o najmniejszym kluczu `k` (lub *NIL* w przypadku gdy *S* jest zbiorem pustym)

Czas wykonania określonej operacji zależy od rozmiaru i rodzaju struktury.

Operacje na dynamicznych strukturach danych

- modyfikujące
  - `INSERT(S,x)` - dodaje do zbioru `S` element `x`
  - `DELETE(S,x)` - usuwa ze zbioru `S` element `x`
- zapytania
  - `SEARCH(S,k)` - dla zbioru `S` i klucza `k` zwraca wskaźnik do elementu `x` (lub `NIL` gdy taki element nie istnieje)
  - `SUCCESSOR(S,x)` - dla elementu `x` zbioru `S` zwraca wskaźnik do jego następnika
  - `MINIMUM(S)` - dla zbioru `S` zwraca wskaźnik do elementu o najmniejszym kluczu `k` (lub *NIL* w przypadku gdy *S* jest zbiorem pustym)

Czas wykonania określonej operacji zależy od rozmiaru i rodzaju struktury.

Operacje na dynamicznych strukturach danych

- modyfikujące
  - `INSERT(S,x)` - dodaje do zbioru `S` element `x`
  - `DELETE(S,x)` - usuwa ze zbioru `S` element `x`
- zapytania
  - `SEARCH(S,k)` - dla zbioru `S` i klucza `k` zwraca wskaźnik do elementu `x` (lub `NIL` gdy taki element nie istnieje)
  - `SUCCESSOR(S,x)` - dla elementu `x` zbioru `S` zwraca wskaźnik do jego następnika
  - `MINIMUM(S)` - dla zbioru `S` zwraca wskaźnik do elementu o najmniejszym kluczu `k` (lub *NIL* w przypadku gdy *S* jest zbiorem pustym)

Czas wykonania określonej operacji zależy od rozmiaru i rodzaju struktury.

Operacje na dynamicznych strukturach danych

- modyfikujące
  - `INSERT(S,x)` - dodaje do zbioru `S` element `x`
  - `DELETE(S,x)` - usuwa ze zbioru `S` element `x`
- zapytania
  - `SEARCH(S,k)` - dla zbioru `S` i klucza `k` zwraca wskaźnik do elementu `x` (lub `NIL` gdy taki element nie istnieje)
  - `SUCCESSOR(S,x)` - dla elementu `x` zbioru `S` zwraca wskaźnik do jego następnika
  - `MINIMUM(S)` - dla zbioru `S` zwraca wskaźnik do elementu o najmniejszym kluczu `k` (lub *NIL* w przypadku gdy *S* jest zbiorem pustym)

Czas wykonania określonej operacji zależy od rozmiaru i rodzaju struktury.

Operacje na dynamicznych strukturach danych

- modyfikujące
  - `INSERT(S,x)` - dodaje do zbioru `S` element `x`
  - `DELETE(S,x)` - usuwa ze zbioru `S` element `x`
- zapytania
  - `SEARCH(S,k)` - dla zbioru `S` i klucza `k` zwraca wskaźnik do elementu `x` (lub `NIL` gdy taki element nie istnieje)
  - `SUCCESSOR(S,x)` - dla elementu `x` zbioru `S` zwraca wskaźnik do jego następnika
  - `MINIMUM(S)` - dla zbioru `S` zwraca wskaźnik do elementu o najmniejszym kluczu `k` (lub *NIL* w przypadku gdy *S* jest zbiorem pustym)

Czas wykonania określonej operacji zależy od rozmiaru i rodzaju struktury.

Stos

Dynamiczna struktura danych, w której usuwany jest ten element, który został wstawiony jako ostatni (LIFO: last-in, first-out).

Podstawowe operacje na stosie:

- STACK-EMPTY(S) - sprawdza, czy stos S jest pusty
- PUSH(S, x) - dodaje na stos element x
- POP(S) - usuwa element z wierzchołka stosu S i zwraca go jako wynik

Stos

Dynamiczna struktura danych, w której usuwany jest ten element, który został wstawiony jako ostatni (LIFO: last-in, first-out).
Podstawowe operacje na stosie:

- STACK-EMPTY(S) - sprawdza, czy stos S jest pusty
- PUSH(S, x) - dodaje na stos element x
- POP(S) - usuwa element z wierzchołka stosu S i zwraca go jako wynik

Stos

Dynamiczna struktura danych, w której usuwany jest ten element, który został wstawiony jako ostatni (LIFO: last-in, first-out).
Podstawowe operacje na stosie:

- `STACK-EMPTY(S)` - sprawdza, czy stos S jest pusty
- `PUSH(S,x)` - dodaje na stos element x
- `POP(S)` - usuwa element z wierzchołka stosu S i zwraca go jako wynik

Stos

Dynamiczna struktura danych, w której usuwany jest ten element, który został wstawiony jako ostatni (LIFO: last-in, first-out).
Podstawowe operacje na stosie:

- `STACK-EMPTY(S)` - sprawdza, czy stos S jest pusty
- `PUSH(S,x)` - dodaje na stos element x
- `POP(S)` - usuwa element z wierzchołka stosu S i zwraca go jako wynik

Stos

Dynamiczna struktura danych, w której usuwany jest ten element, który został wstawiony jako ostatni (LIFO: last-in, first-out).
Podstawowe operacje na stosie:

- `STACK-EMPTY(S)` - sprawdza, czy stos S jest pusty
- `PUSH(S,x)` - dodaje na stos element x
- `POP(S)` - usuwa element z wierzchołka stosu S i zwraca go jako wynik

Stos

Dodatkowe operacje na stosie (nie są wymagane w podstawowej implementacji):

- `TOP(S)` - zwraca szczytowy element stosu S bez usuwania go
- `SIZE(S)` - zwraca liczbę elementów stosu S
- `CLEAR(S)` - usuwa wszystkie elementy ze stosu S
- `MULTIPOP(S,k)` - usuwa k elementów ze stosu S, lub wszystkie, jeśli było ich mniej niż k

Stos

Dodatkowe operacje na stosie (nie są wymagane w podstawowej implementacji):

- `TOP(S)` - zwraca szczytowy element stosu S bez usuwania go
- `SIZE(S)` - zwraca liczbę elementów stosu S
- `CLEAR(S)` - usuwa wszystkie elementy ze stosu S
- `MULTIPOP(S,k)` - usuwa k elementów ze stos S, lub wszystkie jeśli było ich mniej niż k

Stos

Dodatkowe operacje na stosie (nie są wymagane w podstawowej implementacji):

- `TOP(S)` - zwraca szczytowy element stosu S bez usuwania go
- `SIZE(S)` - zwraca liczbę elementów stosu S
- `CLEAR(S)` - usuwa wszystkie elementy ze stosu S
- `MULTIPOP(S,k)` - usuwa k elementów ze stos S, lub wszystkie jeśli było ich mniej niż k

Stos

Dodatkowe operacje na stosie (nie są wymagane w podstawowej implementacji):

- `TOP(S)` - zwraca szczytowy element stosu S bez usuwania go
- `SIZE(S)` - zwraca liczbę elementów stosu S
- `CLEAR(S)` - usuwa wszystkie elementy ze stosu S
- `MULTIPOP(S,k)` - usuwa k elementów ze stos S, lub wszystkie jeśli było ich mniej niż k

Stos

Dodatkowe operacje na stosie (nie są wymagane w podstawowej implementacji):

- `TOP(S)` - zwraca szczytowy element stosu S bez usuwania go
- `SIZE(S)` - zwraca liczbę elementów stosu S
- `CLEAR(S)` - usuwa wszystkie elementy ze stosu S
- `MULTIPOP(S,k)` - usuwa k elementów ze stos S, lub wszystkie jeśli było ich mniej niż k

Stos

Tablicowa implementacja stosu:

- Stos zawierający co najwyżej n elementów można zaimplementować za pomocą tabeli `S[1..n]`
- Dodatkowo dodajemy atrybut `S.top` którego wartość jest indeksem ostatniego wstawionego elementu
- Stos jest pusty gdy `S.top = 0`

Stos

Tablicowa implementacja stosu:

- Stos zawierający co najwyżej n elementów można zaimplementować za pomocą tabeli `S[1..n]`
- Dodatkowo dodajemy atrybut `S.top` którego wartość jest indeksem ostatniego wstawionego elementu
- Stos jest pusty gdy `S.top = 0`

Stos

Tablicowa implementacja stosu:

- Stos zawierający co najwyżej n elementów można zaimplementować za pomocą tabeli `S[1..n]`
- Dodatkowo dodajemy atrybut `S.top` którego wartość jest indeksem ostatniego wstawionego elementu
- Stos jest pusty gdy `S.top = 0`

Stos

Tablicowa implementacja stosu:

- Stos zawierający co najwyżej n elementów można zaimplementować za pomocą tabeli `S[1..n]`
- Dodatkowo dodajemy atrybut `S.top` którego wartość jest indeksem ostatniego wstawionego elementu
- Stos jest pusty gdy `S.top = 0`

Stos - metody

```
procedure STACK-EMPTY(S)
 if S.top = 0
  return true
 else
  return false

procedure PUSH(S,x)
 S.top = S.top + 1
 S[S.top] = x

procedure POP(S)
 S.top = S.top - 1
 return S[S.top + 1]
```

Każda z operacji wykonuje się w czasie $\Theta(1)$

Stos - metody

```
procedure STACK-EMPTY(S)
 if S.top = 0
  return true
 else
  return false

procedure PUSH(S,x)
 S.top = S.top + 1
 S[S.top] = x

procedure POP(S)
 S.top = S.top - 1
 return S[S.top + 1]
```

Każda z operacji wykonuje się w czasie $\Theta(1)$

Stos - metody

```
procedure STACK-EMPTY(S)
 if S.top = 0
  return true
 else
  return false

procedure PUSH(S,x)
 S.top = S.top + 1
 S[S.top] = x

procedure POP(S)
 S.top = S.top - 1
 return S[S.top + 1]
```

Każda z operacji wykonuje się w czasie $\Theta(1)$

Stos - metody

```
procedure STACK-EMPTY(S)
 if S.top = 0
  return true
 else
  return false

procedure PUSH(S,x)
 S.top = S.top + 1
 S[S.top] = x

procedure POP(S)
 S.top = S.top - 1
 return S[S.top + 1]
```

Każda z operacji wykonuje się w czasie $\Theta(1)$

Stos

Podczas dodawania i usuwania elementów ze stosu może wystąpić jeden z błędów:

- błąd przepełnienia - próba umieszczenia więcej niż n elementów
- błąd niedomiaru - próba usunięcia elementu z pustego stosu

Należy uzupełnić metody o mechanizm wykrywania powyższych błędów

Stos

Podczas dodawania i usuwania elementów ze stosu może wystąpić jeden z błędów:

- błąd przepełnienia - próba umieszczenia więcej niż n elementów
- błąd niedomiaru - próba usunięcia elementu z pustego stosu

Należy uzupełnić metody o mechanizm wykrywania powyższych błędów

Stos

Podczas dodawania i usuwania elementów ze stosu może wystąpić jeden z błędów:

- błąd przepełnienia - próba umieszczenia więcej niż n elementów
- błąd niedomiaru - próba usunięcia elementu z pustego stosu

Należy uzupełnić metody o mechanizm wykrywania powyższych błędów

Stos

Podczas dodawania i usuwania elementów ze stosu może wystąpić jeden z błędów:

- błąd przepełnienia - próba umieszczenia więcej niż n elementów
- błąd niedomiaru - próba usunięcia elementu z pustego stosu

Należy uzupełnić metody o mechanizm wykrywania powyższych błędów

Stos

Podczas dodawania i usuwania elementów ze stosu może wystąpić jeden z błędów:

- błąd przepełnienia - próba umieszczenia więcej niż n elementów
- błąd niedomiaru - próba usunięcia elementu z pustego stosu

Należy uzupełnić metody o mechanizm wykrywania powyższych błędów

Stos - metody

```
procedure PUSH(S,x)
 if S.top = S.length
  error "przepełnienie"
 else
  S.top = S.top + 1
  S[S.top] = x

procedure POP(S)
 if STACK-EMPTY(S)
  error "niedomiar"
 else
  S.top = S.top - 1
  return S[S.top + 1]
```

Stos - metody

```
procedure PUSH(S,x)
 if S.top = S.length
  error "przepełnienie"
 else
  S.top = S.top + 1
  S[S.top] = x

procedure POP(S)
 if STACK-EMPTY(S)
  error "niedomiar"
 else
  S.top = S.top - 1
  return S[S.top + 1]
```

Stos - metody

Implementacja dodatkowych operacji na stosie:

```
procedure TOP(S)
 if STACK-EMPTY(S)
  error "niedomiar"
 else
  return S[S.top]

procedure SIZE(S)
 return S.top
```

Stos - metody

Implementacja dodatkowych operacji na stosie:

```
procedure TOP(S)
 if STACK-EMPTY(S)
  error "niedomiar"
 else
  return S[S.top]

procedure SIZE(S)
 return S.top
```

Stos - metody

```
procedure CLEAR(S)
 S.top = 0

procedure MULTIPOP(S,k)
 if S.top < k
  S.top = 0
 else
  S.top = S.top - k
```

Stos - metody

```
procedure CLEAR(S)
 S.top = 0

procedure MULTIPOP(S,k)
 if S.top < k
  S.top = 0
 else
  S.top = S.top - k
```

Stos

Przykładowe zastosowanie stosu:

- Przechowywanie adresów zmiennych lokalnych i powrotu podczas wywoływań funkcji
- Sprawdzanie poprawności nawiasowych (kompilacja, HTML, XML)
- obsługa przycisków "wstecz" i "dalej" w przeglądarkach

Stos

Przykładowe zastosowanie stosu:

- Przechowywanie adresów zmiennych lokalnych i powrotu podczas wywoływań funkcji
- Sprawdzanie poprawności nawiasowych (kompilacja, HTML, XML)
- obsługa przycisków "wstecz" i "dalej" w przeglądarkach

Stos

Przykładowe zastosowanie stosu:

- Przechowywanie adresów zmiennych lokalnych i powrotu podczas wywoływań funkcji
- Sprawdzanie poprawności nawiasowych (kompilacja, HTML, XML)
- obsługa przycisków "wstecz" i "dalej" w przeglądarkach

Stos

Przykładowe zastosowanie stosu:

- Przechowywanie adresów zmiennych lokalnych i powrotu podczas wywoływań funkcji
- Sprawdzanie poprawności nawiasowych (kompilacja, HTML, XML)
- obsługa przycisków "wstecz" i "dalej" w przeglądarkach

Kolejka

Dynamiczna struktura danych, w której usuwany jest ten element, który został wstawiony jako pierwszy (FIFO: first-in, first-out).

Podstawowe operacje na kolejce:

- QUEUE-EMPTY(Q) - sprawdza, czy kolejka Q jest pusta
- ENQUEUE(Q, x) - dodaje do kolejki element x
- DEQUEUE(Q) - usuwa pierwszy element z kolejki Q i zwraca go jako wynik

Kolejka

Dynamiczna struktura danych, w której usuwany jest ten element, który został wstawiony jako pierwszy (FIFO: first-in, first-out).
Podstawowe operacje na kolejce:

- `QUEUE-EMPTY(Q)` - sprawdza, czy kolejka Q jest pusta
- `ENQUEUE(Q, x)` - dodaje do kolejki element x
- `DEQUEUE(Q)` - usuwa pierwszy element z kolejki Q i zwraca go jako wynik

Kolejka

Dynamiczna struktura danych, w której usuwany jest ten element, który został wstawiony jako pierwszy (FIFO: first-in, first-out).
Podstawowe operacje na kolejce:

- `QUEUE-EMPTY(Q)` - sprawdza, czy kolejka Q jest pusta
- `ENQUEUE(Q,x)` - dodaje do kolejki element x
- `DEQUEUE(Q)` - usuwa pierwszy element z kolejki Q i zwraca go jako wynik

Kolejka

Dynamiczna struktura danych, w której usuwany jest ten element, który został wstawiony jako pierwszy (FIFO: first-in, first-out).
Podstawowe operacje na kolejce:

- `QUEUE-EMPTY(Q)` - sprawdza, czy kolejka Q jest pusta
- `ENQUEUE(Q,x)` - dodaje do kolejki element x
- `DEQUEUE(Q)` - usuwa pierwszy element z kolejki Q i zwraca go jako wynik

Kolejka

Dynamiczna struktura danych, w której usuwany jest ten element, który został wstawiony jako pierwszy (FIFO: first-in, first-out).
Podstawowe operacje na kolejce:

- `QUEUE-EMPTY(Q)` - sprawdza, czy kolejka Q jest pusta
- `ENQUEUE(Q,x)` - dodaje do kolejki element x
- `DEQUEUE(Q)` - usuwa pierwszy element z kolejki Q i zwraca go jako wynik

Kolejka

Dodatkowe operacje na kolejce (nie są wymagane w podstawowej implementacji):

- `FRONT(Q)` - zwraca pierwszy element kolejki Q bez usuwania go
- `SIZE(Q)` - zwraca liczbę elementów kolejki Q
- `CLEAR(Q)` - usuwa wszystkie elementy z kolejki Q

Kolejka

Dodatkowe operacje na kolejce (nie są wymagane w podstawowej implementacji):

- `FRONT(Q)` - zwraca pierwszy element kolejki Q bez usuwania go
- `SIZE(Q)` - zwraca liczbę elementów kolejki Q
- `CLEAR(Q)` - usuwa wszystkie elementy z kolejki Q

Kolejka

Dodatkowe operacje na kolejce (nie są wymagane w podstawowej implementacji):

- `FRONT(Q)` - zwraca pierwszy element kolejki Q bez usuwania go
- `SIZE(Q)` - zwraca liczbę elementów kolejki Q
- `CLEAR(Q)` - usuwa wszystkie elementy z kolejki Q

Kolejka

Dodatkowe operacje na kolejce (nie są wymagane w podstawowej implementacji):

- `FRONT(Q)` - zwraca pierwszy element kolejki Q bez usuwania go
- `SIZE(Q)` - zwraca liczbę elementów kolejki Q
- `CLEAR(Q)` - usuwa wszystkie elementy z kolejki Q

Kolejka

Tablicowa implementacja kolejki

- Kolejkę zawierającą co najwyżej $n - 1$ elementów można zaimplementować za pomocą tabeli `Q[1..n]`
- Dodatkowo dodajemy atrybuty:
  - Kolejka składa się z elementów $Q[Q.head], \ldots, Q[Q.tail - 1]$, przy czym tablica Q jest jest cykliczna, tzn. następnikiem elementu o indeksie n jest pozycja o indeksie 1
  - Kolejka jest pusta, gdy $Q.head = Q.tail$
  - Kolejka jest pełna, gdy $Q.head = Q.tail + 1$

Kolejka

Tablicowa implementacja kolejki

- Kolejkę zawierającą co najwyżej $n - 1$ elementów można zaimplementować za pomocą tabeli `Q[1..n]`
- Dodatkowo dodajemy atrybuty:
  - $Q.head$ jest indeksem początku kolejki
  - $Q.tail$ jest indeksem pierwszej wolnej pozycji
  - $Q.length$ jest liczbą elementów w kolejce
- Kolejka składa się z elementów $Q[Q.head], \ldots, Q[Q.tail - 1]$, przy czym tablica Q jest jest cykliczna, tzn. następnikiem elementu o indeksie n jest pozycja o indeksie 1
- Kolejka jest pusta, gdy $Q.head = Q.tail$
- Kolejka jest pełna, gdy $Q.head = Q.tail + 1$

Kolejka

Tablicowa implementacja kolejki

- Kolejkę zawierającą co najwyżej $n - 1$ elementów można zaimplementować za pomocą tabeli `Q[1..n]`
- Dodatkowo dodajemy atrybuty:
  - $Q.head$ jest indeksem początku kolejki
  - $Q.tail$ jest indeksem pierwszej wolnej pozycji
  - $Q.length$ jest liczbą elementów w kolejce
- Kolejka składa się z elementów $Q[Q.head], \ldots, Q[Q.tail - 1]$, przy czym tablica Q jest jest cykliczna, tzn. następnikiem elementu o indeksie n jest pozycja o indeksie 1
- Kolejka jest pusta, gdy $Q.head = Q.tail$
- Kolejka jest pełna, gdy $Q.head = Q.tail + 1$

Kolejka

Tablicowa implementacja kolejki

- Kolejkę zawierającą co najwyżej $n - 1$ elementów można zaimplementować za pomocą tabeli `Q[1..n]`
- Dodatkowo dodajemy atrybuty:
  - $Q.head$ jest indeksem początku kolejki
  - $Q.tail$ jest indeksem pierwszej wolnej pozycji
  - $Q.length$ jest liczbą elementów w kolejce
- Kolejka składa się z elementów $Q[Q.head], \ldots, Q[Q.tail - 1]$, przy czym tablica Q jest jest cykliczna, tzn. następnikiem elementu o indeksie n jest pozycja o indeksie 1
- Kolejka jest pusta, gdy $Q.head = Q.tail$
- Kolejka jest pełna, gdy $Q.head = Q.tail + 1$

Kolejka

Tablicowa implementacja kolejki

- Kolejkę zawierającą co najwyżej $n - 1$ elementów można zaimplementować za pomocą tabeli `Q[1..n]`
- Dodatkowo dodajemy atrybuty:
  - $Q.head$ jest indeksem początku kolejki
  - $Q.tail$ jest indeksem pierwszej wolnej pozycji
  - $Q.length$ jest liczbą elementów w kolejce
- Kolejka składa się z elementów $Q[Q.head], \ldots, Q[Q.tail - 1]$, przy czym tablica Q jest jest cykliczna, tzn. następnikiem elementu o indeksie n jest pozycja o indeksie 1
- Kolejka jest pusta, gdy $Q.head = Q.tail$
- Kolejka jest pełna, gdy $Q.head = Q.tail + 1$

Kolejka

Tablicowa implementacja kolejki

- Kolejkę zawierającą co najwyżej $n - 1$ elementów można zaimplementować za pomocą tabeli `Q[1..n]`
- Dodatkowo dodajemy atrybuty:
  - $Q.head$ jest indeksem początku kolejki
  - $Q.tail$ jest indeksem pierwszej wolnej pozycji
  - $Q.length$ jest liczbą elementów w kolejce
- Kolejka składa się z elementów $Q[Q.head], \ldots, Q[Q.tail - 1]$, przy czym tablica Q jest jest cykliczna, tzn. następnikiem elementu o indeksie n jest pozycja o indeksie 1
- Kolejka jest pusta, gdy $Q.head = Q.tail$
- Kolejka jest pełna, gdy $Q.head = Q.tail + 1$

Kolejka

Tablicowa implementacja kolejki

- Kolejkę zawierającą co najwyżej $n - 1$ elementów można zaimplementować za pomocą tabeli `Q[1..n]`
- Dodatkowo dodajemy atrybuty:
  - $Q.head$ jest indeksem początku kolejki
  - $Q.tail$ jest indeksem pierwszej wolnej pozycji
  - $Q.length$ jest liczbą elementów w kolejce
- Kolejka składa się z elementów $Q[Q.head], \ldots, Q[Q.tail - 1]$, przy czym tablica Q jest jest cykliczna, tzn. następnikiem elementu o indeksie n jest pozycja o indeksie 1
- Kolejka jest pusta, gdy $Q.head = Q.tail$
- Kolejka jest pełna, gdy $Q.head = Q.tail + 1$

Kolejka

Tablicowa implementacja kolejki

- Kolejkę zawierającą co najwyżej $n - 1$ elementów można zaimplementować za pomocą tabeli `Q[1..n]`
- Dodatkowo dodajemy atrybuty:
  - $Q.head$ jest indeksem początku kolejki
  - $Q.tail$ jest indeksem pierwszej wolnej pozycji
  - $Q.length$ jest liczbą elementów w kolejce
- Kolejka składa się z elementów $Q[Q.head], \ldots, Q[Q.tail - 1]$, przy czym tablica Q jest jest cykliczna, tzn. następnikiem elementu o indeksie n jest pozycja o indeksie 1
- Kolejka jest pusta, gdy $Q.head = Q.tail$
- Kolejka jest pełna, gdy $Q.head = Q.tail + 1$

Kolejka

Tablicowa implementacja kolejki

- Kolejkę zawierającą co najwyżej $n - 1$ elementów można zaimplementować za pomocą tabeli `Q[1..n]`
- Dodatkowo dodajemy atrybuty:
  - $Q.head$ jest indeksem początku kolejki
  - $Q.tail$ jest indeksem pierwszej wolnej pozycji
  - $Q.length$ jest liczbą elementów w kolejce
- Kolejka składa się z elementów $Q[Q.head], \ldots, Q[Q.tail - 1]$, przy czym tablica Q jest jest cykliczna, tzn. następnikiem elementu o indeksie n jest pozycja o indeksie 1
- Kolejka jest pusta, gdy $Q.head = Q.tail$
- Kolejka jest pełna, gdy $Q.head = Q.tail + 1$

Kolejka - metody

```
procedure QUEUE-EMPTY(Q)
 if Q.head = Q.tail
  return true
 else
  return false

procedure ENQUEUE(Q,x)
 Q[Q.tail] = x
 if Q.tail = Q.length
  Q.tail = 1
 else
  Q.tail = Q.tail + 1
```

Kolejka - metody

```
procedure QUEUE-EMPTY(Q)
 if Q.head = Q.tail
  return true
 else
  return false

procedure ENQUEUE(Q,x)
 Q[Q.tail] = x
 if Q.tail = Q.length
  Q.tail = 1
 else
  Q.tail = Q.tail + 1
```

Kolejka - metody

```
procedure DEQUEUE(Q,x)
 x = Q[Q.head]
 if Q.head = Q.length
  Q.head = 1
 else
  Q.head = Q.head + 1
 return x
```

Każda z operacji wykonuje się w czasie $\Theta(1)$

Implementacje należy uzupełnić o błąd przepełnienia i niedomiaru

Kolejka - metody

```
procedure DEQUEUE(Q,x)
 x = Q[Q.head]
 if Q.head = Q.length
  Q.head = 1
 else
  Q.head = Q.head + 1
 return x
```

Każda z operacji wykonuje się w czasie $\Theta(1)$
Implementacje należy uzupełnić o błąd przepełnienia i niedomiaru

Stos - metody

```
procedure ENQUEUE(Q,x)
 if Q.head=Q.tail+1 or (Q.head=1 and Q.tail=Q.length)
  error "przepełnienie"
 else
  Q[Q.tail] = x
  if Q.tail = Q.length
   Q.tail = 1
  else
   Q.tail = Q.tail + 1
```

Stos - metody

```
procedure DEQUEUE(Q,x)
 if QUEUE-EMPTY(Q)
  error "niedomiar"
 else
  x = Q[Q.head]
  if Q.head = Q.length
   Q.head = 1
  else
   Q.head = Q.head + 1
```

Kolejka - metody

Implementacja dodatkowych operacji na kolejce:

```
procedure FRONT(Q)
 if QUEUE-EMPTY(Q)
  error "niedomiar"
 else
  return Q[Q.head]

procedure SIZE(Q)
 if Q.tail ≥ Q.head
  return Q.tail - Q.head
 else
  return Q.tail - Q.head + Q.length

procedure CLEAR(Q)
 Q.tail = Q.head
```

Kolejka - metody

Implementacja dodatkowych operacji na kolejce:

```
procedure FRONT(Q)
 if QUEUE-EMPTY(Q)
  error "niedomiar"
 else
  return Q[Q.head]

procedure SIZE(Q)
 if Q.tail ≥ Q.head
  return Q.tail - Q.head
 else
  return Q.tail - Q.head + Q.length

procedure CLEAR(Q)
 Q.tail = Q.head
```

Kolejka - metody

Implementacja dodatkowych operacji na kolejce:

```
procedure FRONT(Q)
 if QUEUE-EMPTY(Q)
  error "niedomiar"
 else
  return Q[Q.head]

procedure SIZE(Q)
 if Q.tail ≥ Q.head
  return Q.tail - Q.head
 else
  return Q.tail - Q.head + Q.length

procedure CLEAR(Q)
 Q.tail = Q.head
```

Kolejka - metody

Implementacja dodatkowych operacji na kolejce:

```
procedure FRONT(Q)
 if QUEUE-EMPTY(Q)
  error "niedomiar"
 else
  return Q[Q.head]

procedure SIZE(Q)
 if Q.tail ≥ Q.head
  return Q.tail - Q.head
 else
  return Q.tail - Q.head + Q.length

procedure CLEAR(Q)
 Q.tail = Q.head
```

Kolejka

Przykładowe zastosowanie kolejki:

- przydzielanie procesom dostępu do twardego dysku
- obsługa zdarzeń w programie
- przetwarzanie komunikatów, w tym przesyłanie pakietów danych w sieci
- buforowanie

Kolejka

Przykładowe zastosowanie kolejki:

- przydzielanie procesom dostępu do twardego dysku
- obsługa zdarzeń w programie
- przetwarzanie komunikatów, w tym przesyłanie pakietów danych w sieci
- buforowanie

Kolejka

Przykładowe zastosowanie kolejki:

- przydzielanie procesom dostępu do twardego dysku
- obsługa zdarzeń w programie
- przetwarzanie komunikatów, w tym przesyłanie pakietów danych w sieci
- buforowanie

Kolejka

Przykładowe zastosowanie kolejki:

- przydzielanie procesom dostępu do twardego dysku
- obsługa zdarzeń w programie
- przetwarzanie komunikatów, w tym przesyłanie pakietów danych w sieci
- buforowanie

Kolejka

Przykładowe zastosowanie kolejki:

- przydzielanie procesom dostępu do twardego dysku
- obsługa zdarzeń w programie
- przetwarzanie komunikatów, w tym przesyłanie pakietów danych w sieci
- buforowanie