Temat: Liniowe uporządkowane struktury danych: stos, kolejka. Specyfikacja, przykładowe implementacje i zastosowania. Struktura słownika.

1. Pojęcie struktury danych

Nieformalnie

Struktura danych (ang. *data structure*) to sposób uporządkowania informacji w komputerze. Na strukturach danych operują algorytmy.

Przykładowe struktury danych to:

- rekord lub struktura,
- tablica,
- stos,
- kolejka,
- lista,
- drzewo,
- graf

Formalnie

Struktura danych to system relacyjny $\langle U, \{r_i\}_{i \in I} \rangle$, gdzie U to uniwersum systemu, a $\{r_i\}_{i \in I}$ to zbiór relacji (operacji na strukturze danych). Uniwersum systemu to zbiór typów parametrów wszystkich operacji zbioru $\{r_i\}_{i \in I}$.

Formalnie definiujemy strukturę danych poprzez:

- określenie sygnatury systemu $\langle U, \{r_i\}_{i \in I} \rangle$,
- określenie sygnatur każdej z relacji $\{r_i\}_{i \in I}$
- specyfikację każdej z relacji $\{r_i\}_{i \in I}$

2. Definicja struktury stosu

Nieformalnie

Stos jest strukturą liniową uporządkowanych danych, z których jedynie ostatni element, zwany *wierzchołkiem*, jest w danym momencie dostępny. W wierzchołku odbywa się dołączanie nowych elementów, również jedynie wierzchołek można usunąć.

Formalnie

Stos to system relacyjny o następującej sygnaturze:

$$\langle E \cup S, push, pop, top, empty \rangle$$

gdzie:
$E \cup S$ to uniwersum struktury stosu
E – niepusty zbiór elementów (możliwych danych, które chcemy zapamiętać w stosie)
S – niepusty zbiór zwany zbiorem stosów

Sygnatury operacji struktury stosu (parametry i wyniki operacji)

$push : S \times E \to S$
$pop : S \to S$
$top : S \to E$
$empty : S \to \{0, 1\}$ (0 – fałsz, 1 - prawda)

Nieformalna specyfikacja operacji stosu
Zakładamy, że : $e \in E, s \in S$

1. Operacja *empty*(*s*) zwraca wartość 1, gdy stos jest pusty, a 0 w przypadku przeciwnym.

2. Operacja *pop*(*s*) usuwa element, który znajduje się w wierzchołku stosu („najpóźniej" wstawiony do stosu). Jeżeli przed wywołaniem operacji *pop*(*s*) stos *s* był pusty, to operacja *pop*(s) nie zmienia stosu *s*.

3. Operacja *push*(*s*, *e*) wstawia element *e* w wierzchołku stosu.

4. Operacja *top*(*s*) zwraca element znajdujący się aktualnie w wierzchołku stosu *s*. Jeżeli przed wywołaniem *top*(*s*) stos *s* był pusty, to wartość operacji *top*(*s*) jest nieokreślona.

3. Przykładowa implementacja struktury stosu

Implementacja tablicowa stosu liczb całkowitych dodatnich

Miejscem zapamiętania liczb w stosie jest jednowymiarowa tablica *S*. W indeksie 0 tablicy *S* znajduje się indeks wierzchołka stosu. Jeżeli stos jest pusty, to *S*[0]=0

Implementacja będzie poprawna względem specyfikacji stosu pod warunkiem, że w stosie nie będzie się znajdowało więcej liczb niż *n* (*n*+1 – wielkość tablicy *S*)

```
top(S)
{
  if (S[0]!=0)  return S[S[0]];
}

empty(S)
{
    if  (S[0]==0)  return 1;
    else  return 0;
}

S push(S, e)
{
   S[0]=S[0]+1;
   S[S[0]]=e;
   return S;
}
```

```
S pop(S)
{
  if (S[0]>0) S[0]=S[0]-1;
  return S;
}
```

Przykład 1

a) stos pusty: *empty*(*S*)=1 , *top*(*S*) – wartość nieokreślona, *pop*(*S*) nie zmienia stosu

S:

b) *push*(*S,* 4)

S:

c) *push*(*S*, 2)

S:

top(*S*)=2, *empty*(*S*)=0

d) *pop*(*S*)

S:

top(*S*)=4, *empty*(*S*)=0

4. Przykłady zastosowania struktury stosu

Przykład 2

Wyświetlenie ciągu podanych przez użytkownika znaków w rewersie. Ciąg jest zakończony znakiem końca linii.

```
iter_reverse()
{

 char ch;
 S=∅;  //Inicjalizacja stosu
 do
   {
     "pobierz ch";
      push(S, ch);
    }while (ch ! =10);
  while (!empty(S))
   {
     ch=top(S);
     "drukuj ch"
     pop(s);
   }
```

Przykład 3

Dodawanie długich liczb całkowitych dodatnich. Zakładamy, że cyfry obydwu dodawanych liczb zostały wstawione do stosu. W wierzchołku stosu znajduje się najmniej znacząca cyfra.

```
S dodaj(X, Y)
{
   S=∅; n=0;
   while (!empty(X) && !empty(Y))
         {
            x=top(X); out(X);
            y=top(Y); out(Y);
            push((x+y+n)%10, S);
           n = (x+y+n)/10;
         }
```

```
 while (!empty(X))
        {
            x=top(X); out(X);
            push((x+n)%10, S);
            n= (x+n)/10;
         }
  while (!empty(Y))
        {
            y=top(Y); out(Y);
            push((y+n)%10, S);
            n= (y+n)/10;
        }
   if  (n>0) push(n, S);
   return S;
}
```

5. Definicja struktury kolejki

Nieformalnie

Kolejka jest strukturą liniową uporządkowanych danych, z których dostępny jest element pierwszy z wstawionych i ostatni z wstawionych. Miejsce przechowywania pierwszego elementu nazywamy *początkiem* kolejki, a miejsce elementu ostatniego jej *końcem*. Dołączanie nowych elementów do kolejki odbywa się na końcu kolejki, a usuwanie elementów kolejki w jej początku.

Formalnie

Kolejka to system relacyjny

$$\langle E \cup Q, in,\ out,\ first,\ empty \rangle$$

E –niepusty zbiór elementów (możliwych danych)
Q –niepusty zbiór zwany zbiorem kolejek

Sygnatura operacji (parametry i wyniki operacji) kolejki

$$in : Q \times E \to Q$$
$$out : Q \to Q$$
$$first : Q \to E$$
$$empty : Q \to \{0, 1\}$$

Nieformalna specyfikacja operacji kolejki

Zakładamy, że : $e \in E, q \in Q$

1. Operacja *empty*(*q*) zwraca wartość 1, gdy kolejka jest pusta, a 0 w przypadku przeciwnym.

2. Operacja *out*(*s*) usuwa element, który znajduje się na początku kolejki („najwcześniej" wstawiony do kolejki). Jeżeli przed wywołaniem operacji *out*(*q*) kolejka *q* była pusta, to operacja *out*(*q*) nie zmienia kolejki *q*.

3. Operacja *in*(*q, e*) wstawia element *e* na koniec kolejki *q*.

4. Operacja *first*(*q*) zwraca element znajdujący się aktualnie na początku kolejki *q*. Jeżeli przed wywołaniem *first*(*q*) kolejka *q* była pusta, to wartość operacji *first*(*q*) jest nieokreślona.

5. Przykładowa implementacja kolejki

Implementacja tablicowa kolejki liczb całkowitych dodatnich

Miejscem zapamiętania liczb kolejki jest jednowymiarowa tablica *Q*. W indeksie 0 tablicy *Q* znajduje się indeks początku kolejki, a w indeksie *n*+1 indeks końca kolejki. Jeżeli kolejka jest pustya, to *Q*[0]=*Q*[*n*+1]=0.

Implementacja będzie poprawna względem specyfikacji kolejki pod warunkiem, że w kolejce nie będzie się znajdowało więcej liczb niż *n* (*n*+2 – wielkość tablicy *Q*)

```
first(Q)
{
  if (Q[0]!=0)  return Q[Q[0]];
}

empty(Q)
{
    if  (Q[0]==0)  return 1;
    else  return 0;
}

Q in(Q, e)
{
   if (Q[n+1]<n)
         {
           Q[n+1]++;
           Q[Q[n+1]]=e;
         }
   else
         {
           Q[1]=e;
           Q[n+1]=1;
         }
     return Q;
 }

Q out(Q)
{
   if (Q[0]!=0)
            if (Q[0]!=n) Q[0]++;
            else  Q[0]=1;
  return Q;
}
```

Przykład 4

a) Kolejka jest pusta: *empty*(*Q*)=1 , *first*(*Q*) – wartość nieokreślona, *out*(*Q*) nie zmienia kolejki

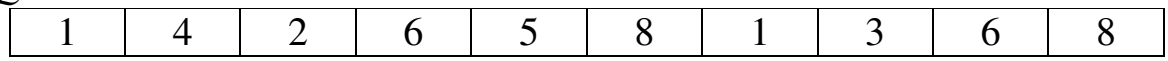
Q:

b) *in*(*Q*, 4)

Q:

c) *in*(*Q*, 2), *in*(*Q*, 6), *in*(*Q*, 5), *in*(*Q*, 8), *in*(*Q*, 1), *in*(*Q*, 3), *in*(*Q*, 6),

Q:



first(*Q*)=4, *empty*(*Q*)=0

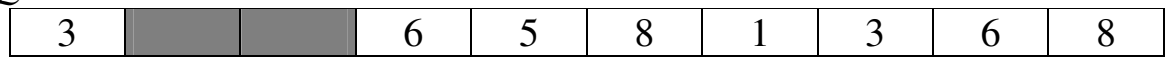
d) *out*(*Q*)

Q:

first(*Q*)=2, *empty*(*Q*)=0

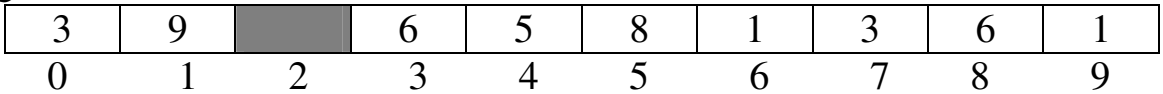
e) *out*(*Q*)

Q:



f) *in*(*Q*, 9)

Q:



6. Przykład zastosowania struktury kolejki

Algorytm generowania wszystkich liczb pierwszych nie większych niż *n*.

Idea sita Eratostenesa

1. Tworzymy kolejkę *P* wszystkich liczb większych równych 2 i mniejszych bądź równych od *n*.

2. Ustalamy liczbę $s = first(P)$.

3. Usuwamy z kolejki *P* pierwszy element i wstawiamy go do kolejki wynikowej *W*.

4. Usuwamy z kolejki *P* pozostałe elementy i wstawiamy je do kolejki pomocniczej *Q*, pod warunkiem, że nie dzielą się przez *s*.

5. Zamieniamy kolejki *P* i *Q*.

6. Kroki 2, 3, 4, 5 powtarzamy, aż do momentu, gdy kolejka *P* będzie pusta.

7. Wynik, czyli liczby pierwsze mniejsze bądź równe *n* znajdują się w kolejce *W*.

Przykład 5
n=15
1.
P: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
Q:
W:
2.
P: 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
Q:
W: 2

3.
P:
Q: 3, 5, 7, 9, 11, 13, 15
W: 2
4.
P: 3, 5, 7, 9, 11, 13, 15
Q:
W: 2
5. itd.
P:
Q: 5, 7, 11, 13
W: 2, 3

P: 5, 7, 11, 13
Q:
W: 2, 3
P:
Q: 7, 11, 13
W: 2, 3, 5
.....
Ostatecznie:
P:
Q:
W: 2, 3, 5, 7, 11, 13

sito_Eratostenesa

```
{
  P=∅; Q=∅; W=∅;
  for (i=2; i<=n; i++)  in(P, i);
  while (!empty(P))
    {
       s=first(P);
       in(W, s);
       while (!empty(P))
             {
```

```
            d=first(P);
            if (d% s!=0) in(Q, d);
        }
    exchange(P, Q);  //zamiana kolejek P oraz Q
}
```

Odmianą struktury kolejki jest kolejka priorytetowa. Struktura ta zostanie omówiona na ćwiczeniach**.**

8. Struktura słownika

Nieformalnie

Słownik jest strukturą do reprezentacji zbioru danych i realizacji na tym zbiorze operacji wstawiania, wyszukiwania i usuwania elementów zbioru. Miejscem przechowywania pierwszego elementu słownika jest jego *początek*. Operacja wstawiania zachodzi o ile wstawiany element nie należy już do słownika.

Formalnie

Słownik to system relacyjny

$$\langle E \cup D, insert,\ search, delete \rangle$$

E –niepusty zbiór elementów (możliwych danych)
D –niepusty zbiór zwany zbiorem słowników

Sygnatura operacji (parametry i wyniki operacji) słownika

$$insert : D \times E \to D$$
$$search : D \times E \to \{0, 1\}$$
$$delete : D \times E \to D$$

Nieformalna specyfikacja operacji słownika

Zakładamy, że : $e \in E, d \in D$

1. Operacja *search*(*d, e*) zwraca wartość 0 gdy element *e* nie należy do słownika, a 1 w przeciwnym przypadku.

2. Operacja *insert*(*d*, *e*) wstawia element *e* do słownika *d*, o ile operacja *search*(*d*, *e*) zwraca wartość 0.

3. Operacja *delete*(*d*, *e*) usuwa element *e* ze słownika *d*, o ile operacja *search*(*d*, *e*) zwraca wartość 1.

Słownik może być zaimplementowany na wiele różnych sposobów, np:

- w tablicy,
- na liście,
- na drzewie

Niech *n* oznacza rozmiar słownika.

Koszt implementacji tablicowej:

a) tablica uporządkowana

$T^{search}_{max}(n) = \Theta(\log n)$, $T^{insert}_{max}(n) = \Theta(n)$, $T^{delete}_{max}(n) = \Theta(n)$

b) tablica nieuporządkowana

$T^{search}_{max}(n) = \Theta(n)$, $T^{insert}_{max}(n) = \Theta(n)$, $T^{delete}_{max}(n) = \Theta(n)$

Koszt implementacji słownika na liście:

$T^{search}_{max}(n) = \Theta(n)$, $T^{insert}_{max}(n) = \Theta(n)$, $T^{delete}_{max}(n) = \Theta(n)$

Na kolejnym wykładzie poznamy implementację słownika w strukturze drzew binarnych: BST i AVL.