Algorytmy i struktury danych

Wykład 3

- **Podstawy struktur danych**
- **Liniowe struktury danych (listy)**
- **Rodzaje struktur liniowych**
- **Implementacja list**
- **Podstawowe operacje na listach jednokierunkowych**

Definicja struktury danych

Definicja

Strukturą danych nazywamy trójkę uporządkowaną **S = (D, R, {e})**, gdzie:

D – oznacza zbiór danych elementarnych

R={r_{we}, N} – jest zbiorem dwóch relacji określonych na strukturze danych:

$r_{we} \subset \{e\} \times D$ – relacja wejścia do struktury danych S,

$N \subset D \times D$ – relacja następstwa (porządkująca) strukturę S,

e – jest elementem wejściowym do struktury danych S (nie jest to element danych struktury danych S).

Zbiór danych elementarnych *D*

- Zbiór danych elementarnych jest skończony i dla wygody operowania oznaczeniami jego elementów poszczególne dane są indeksowane z wykorzystaniem zbioru liczb naturalnych.
- Przykład:

$$D = \{d_1, d_2, d_3, d_4, d_5\}$$

- Liczność powyższego zbioru danych elementarnych wynosi 5, czyli $\overline{D} = 5$

Dane elementarne - zmienne programowe

- *Dane elementarna* d_i są pojęciem abstrakcyjnym, rozumianym jako tzw. „nazwana wartość". Jest ona określana jako uporządkowana dwójka elementów:

$$d_i = (n_i, w_i).$$

gdzie:

  - n_i – nazwa danej,
  - w_i – aktualna wartość danej z określonego zbioru wartości.

- *Zmienna programowa* jest pojęciem związanym z realizacją danej w konkretnym środowisku programistycznym. Posiada zdefiniowaną nazwę, wraz z określeniem zbioru wartości, które może przyjmować

Relacja r_{we} – wskazanie korzenia struktury S

- Relacja $r_{we} \subset \{e\} \times D$ jest opisywana poprzez zbiór (jedno- lub wieloelementowy) par uporządkowanych elementów, z których pierwszym jest zawsze element wejściowy ***e***, natomiast drugim elementem jest jedna z danych elementarnych ze zbioru ***D***.
- Element d_i „uczestniczący" w relacji r_{WE} nazywamy ***korzeniem struktury danych S***.
- Struktura musi mieć zdefiniowany co najmniej jeden korzeń.
- Przykład:

  Jeśli struktura S posiada dwa korzenie: d_2 oraz d_5, to

$$r_{we} = \{(e, d_2), (e, d_5)\}$$

Relacja *N* – ustalenie porządku struktury S

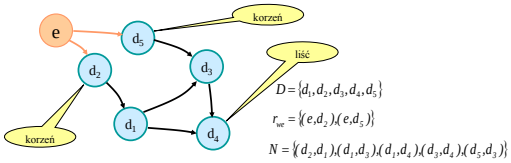
- Relacja następstwa ***N*** opisuje wzajemne uporządkowanie elementów w strukturze danych ***S***. Relację ***N*** definiujemy często w postaci zbioru par uporządkowanych elementów.
- Przykładowy porządek pięcioelementowej struktury danych ***S***:

$$N = \{(d_2, d_1), (d_1, d_3), (d_1, d_4), (d_3, d_4), (d_5, d_3)\}$$

Model grafowy struktury danych

- Aby łatwiej zobrazować strukturę danych i w ten sposób lepiej ją zrozumieć, można zbudować dla niej *model grafowy*. W modelu tym:
  - węzły (wierzchołki) oznaczają poszczególne dane elementarne,
  - łuki (strzałki) służą do odwzorowania następstw poszczególnych danych elementarnych w strukturze ***S***.

Przykład: Model grafowy dla opisywanej struktury ***S***:



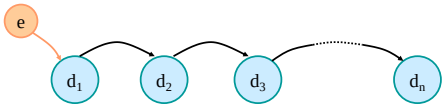
Liniowe struktury danych

Wyróżniamy cztery rodzaje list:

- jednokierunkowe listy niecykliczne,
- dwukierunkowe listy niecykliczne,
- jednokierunkowe listy cykliczne (pierścienie jednokierunkowe),
- dwukierunkowe listy cykliczne (pierścienie dwukierunkowe).

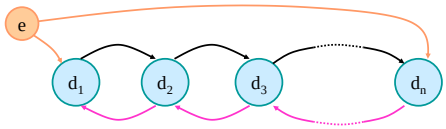
Jednokierunkowe listy niecykliczne

Model grafowy listy jednokierunkowej:



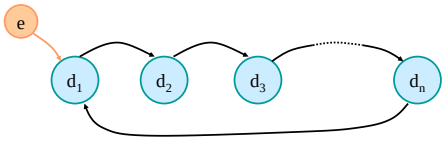
Dwukierunkowe listy niecykliczne

Model grafowy listy dwukierunkowej:



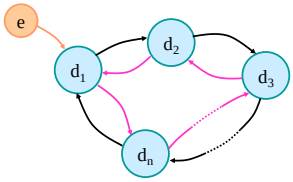
Pierścień jednokierunkowy

Model grafowy pierścienia jednokierunkowego:



Pierścienie dwukierunkowe

Model grafowy pierścienia dwukierunkowego:

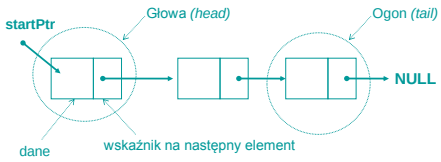


Podstawowe operacje na listach

1. Wyszukiwanie elementu na liście
2. Dołączanie elementu do listy
3. Usuwanie elementu z listy
4. Przestawianie elementów na liście
5. Porządkowanie (sortowanie) listy

Dynamiczne realizacje struktur listowych

- Element listy zawiera:
  - Dane elementarne,
  - Odwzorowanie relacji następstwa – informacja o dowiązaniu do innych elementów;



Dynamiczne realizacje struktur listowych

- Deklaracja elementu listy:

```
struct Node {
        char dane;
        struct Node *next;
};
```

dane elementarne

dowiązanie do kolejnego elementu

```
typedef struct Node *NodePtr;
```
// definicja typu wskaźnikowego do struktury *Node*

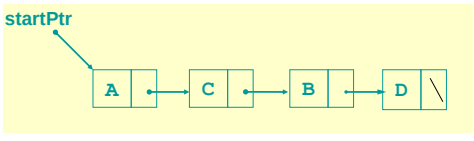
Wyszukiwanie elementu na liście

Algorytm wyszukania elementu na liście jednokierunkowej:

- Cel:
  - Wyszukanie elementu na liście;
- Dane wejściowe:
  - Wskaźnik na pierwszy element listy;
  - Kryterium poszukiwania, np. wartość szukanej danej elementarnej;
- Uwagi:
  - W skrajnym przypadku należy przejrzeć wszystkie elementy (złożoność: $\Theta(n)$);

Algorytm wyszukiwania elementu na liście jednokierunkowej

1. Rozpocznij wyszukiwanie od pierwszego elementu listy;
2. Dopóki nie osiągnięto końca listy oraz szukana wartość jest różna od wartości aktualnego elementu, przejdź do następnego elementu listy;
3. Zwróć wskaźnik na znaleziony (aktualny) element lub wartość NULL (gdy szukanej wartości nie ma na liście)



Algorytm wyszukiwania elementu na liście jednokierunkowej

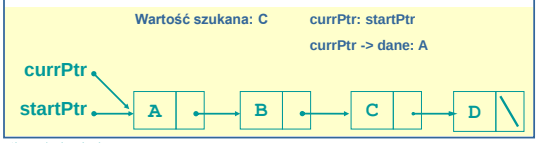
```
struct Node {
        char dane;
        struct Node *next;
      };

Node *Find (Node *startPtr, char szukwart) {
    NodePtr currPtr = startPtr;     // adres na pierwszy element listy
    while ((currPtr != NULL) && (currPtr -> dane != szukwart))
      currPtr = currPtr -> next;
    return currPtr;
}
```

1. Co zwraca funkcja *Find*?
2. Co zwraca funkcja *Find*, gdy szukanego elementu nie ma na liście?

Dynamiczne realizacje struktur listowych

Przykład:

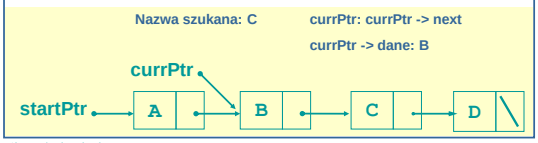
Node *element = Find (startPtr, 'C');



Dynamiczne realizacje struktur listowych

Przykład (cd):

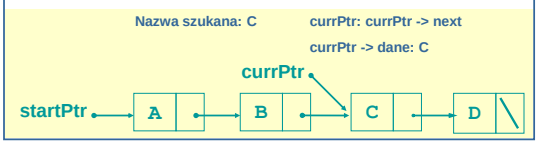
Node *element = Find (startPtr, 'C');



Dynamiczne realizacje struktur listowych

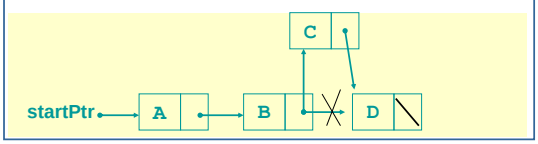
Przykład (cd.)

Node *element = Find (startPtr, 'C');



Algorytm dołączania elementu do uporządkowanej listy jednokierunkowej

- Cel:
  - Dodanie nowego elementu (z wartością C) do listy uporządkowanej (we właściwe miejsce);
- Dane wejściowe:
  - Wskaźnik na pierwszy element listy;
  - Dane do wstawienia na listę;

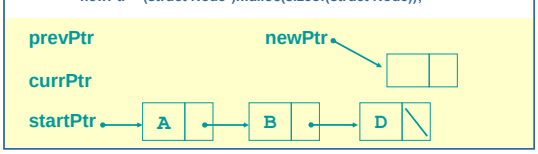


Algorytm dołączania elementu do uporządkowanej listy jednokierunkowej

1. **Utwórz element i ustal dane elementarne**
2. **Znajdź miejsce wstawienia elementu na listę**
   - Zainicjuj *currPtr* na początek listy a *prevPtr* na NULL;
   - Dopóki *currPtr* jest różny od NULL i wartość wstawiana jest większa od *currPtr->dane*:
     - Ustaw *prevPtr* na *currPtr*;
     - Przesuń *currPtr* na następny element listy;
3. **Wstaw element na listę:**
   - wstaw element jako pierwszy na liście:
     - Ustaw pole *next* elementu wstawianego na pierwszy element dotychczasowej listy;
     - Ustaw wskaźnik do listy na element wstawiony;
   - lub wstaw element w wyznaczone miejsce na liście:
     - Ustaw pole *next* elementu *prevPtr* na element wstawiany;
     - Ustaw pole *next* elementu wstawianego na element *currPtr*;

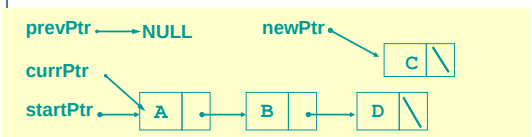
1. Utwórz element i ustal dane elementarne

```
struct Node {
        char dane;
        struct Node *next; };
typedef struct Node *NodePtr;
int insert (NodePtr *startPtr, char wstwart) {
    struct Node *newPtr, *currPtr, *prevPtr;

    // Utwórz element Node
    newPtr = (struct Node*)malloc(sizeof(struct Node));
```



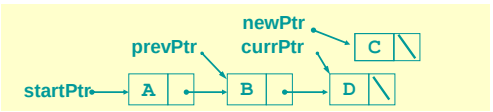
1. Utwórz element i ustal dane elementarne (cd.)

```
if (newPtr == NULL)  // weryfikacja przydzielonej pamięci
   return 1;
else
{  // Ustal dane elementarne w Node
   newPtr -> dane = wstwart;
   newPtr -> next = NULL;
   // Zainicjowanie wskaźników pomocniczych
   currPtr = startPtr;   /* ustaw wskaźnik na głowę listy */
   prevPtr = NULL;
```



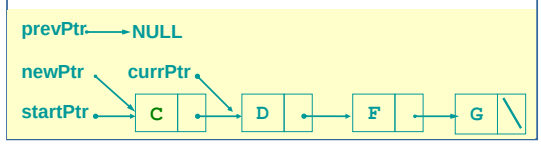
2. Znajdź miejsce wstawienia elementu na listę

```
// Znajdź miejsce wstawienia
while ((currPtr != NULL) && (wstwart  > currPtr->dane))
{
        prevPtr = currPtr;
        currPtr = currPtr -> next;
}
```



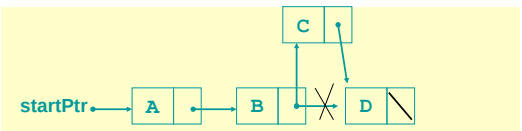
3. Wstaw element na listę

```
if (prevPtr == NULL)
// Wstaw element jako pierwszy na liście
{
  newPtr -> next = *startPtr;
  *startPtr = newPtr;
}
```



3. Wstaw element na listę (cd.)

```
// Wstaw element w miejsce między prevPtr a currPtr
else
{
  newPtr->next = currPtr;
  prevPtr->next = newPtr;
}
}
 return 0;
}
```



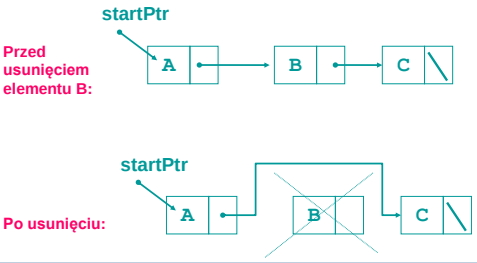
Algorytm dołączania elementu do listy jednokierunkowej – pełny kod

```
int insert (NodePtr *startPtr, char wstwart) {
   struct Node *newPtr, *currPtr, *prevPtr;

   /* Utwórz element Node */
   newPtr = (struct Node *)malloc(sizeof(Node));
   if (newPtr == NULL)  /* weryfikacja przydzielonej pamięci*/
      return 1;
   else
   {  /* Ustal dane elementarne w Node */
      newPtr -> dane = wstwart;
      newPtr -> next = NULL;
      /* Zainicjowanie wskaźników pomocniczych */
      currPtr = startPtr;  /* ustaw wskaźnik na głowę listy */
      prevPtr = NULL;
      while ((currPtr != NULL) && (wstwart > currPtr->dane)) {
                prevPtr = currPtr;
                currPtr = currPtr -> next;
      }
      if (prevPtr == NULL) { /* Wstaw element jako pierwszy w liście */
                        newPtr -> next = *startPtr;
                        *startPtr = newPtr;
                }

      else  { /* Wstaw element w miejsce między prevPtr a currPtr */
                        newPtr->next = currPtr;
                        prevPtr->next = newPtr;
                }
   }
   return 0;
}
```


Algorytm usuwania elementu z uporządkowanej listy jednokierunkowej

- Cel:
  - Usunięcie elementu z listy;
- Dane wejściowe:
  - Wskaźnik na pierwszy element listy *startPtr*;
  - Opis usuwanego elementu, np. wartość danej elementarnej;

Idea algorytmu usuwania elementu z uporządkowanej listy jednokierunkowej



Algorytm usuwania elementu z uporządkowanej listy jednokierunkowej

1. Znajdź element do usunięcia na liście;
2. Jeżeli nie znaleziono elementu, zwróć kod błędu;
3. Usuń znaleziony element z listy.

Algorytm usuwania elementu z uporządkowanej listy jednokierunkowej

- Jeżeli dane są zgodne z danymi pierwszego elementu listy usuń pierwszy element listy;

```
int delete (Node **startPtr, char uswart) {
   Node *prevPtr, *currPtr, *tempPtr;
   if (*startPtr == NULL) return 1;              // Lista pusta
   else {
      if (uswart == (*startPtr)->dane)    // Usuń pierwszy element listy
          {
            tempPtr = *startPtr;
            *startPtr = (*startPtr)->next;
            free (tempPtr);
          }
```


Algorytm usuwania elementu z uporządkowanej listy jednokierunkowej

- Znajdź element do usunięcia na liście:

```
   …
   else  // znajdź na liście element do usunięcia
      {
        prevPtr = *startPtr;                  // początek listy
        currPtr = (*startPtr)->next;       // drugi element
        while (currPtr != NULL && currPtr -> dane != uswart)
        {
           prevPtr = currPtr;
           currPtr = currPtr -> next;
        }
```


Algorytm usuwania elementu z uporządkowanej listy jednokierunkowej

- Znajdź element do usunięcia na liście (cd.):
  - Jeżeli nie znaleziono elementu, zwróć kod błędu
  - Usuń znaleziony element;

```
   …
   if (currPtr == NULL)
      return 1;     // element nie został znaleziony
   else   // Usuń znaleziony element
   {
      tempPtr = currPtr;
      prevPtr->next = currPtr->next;
      free (tempPtr);
      } } }
  return 0;
  }
```


Algorytm usuwania elementu z uporządkowanej listy jednokierunkowej

```
int delete (NodePtr *startPtr, char uswart)
    {  Node *prevPtr, *currPtr, *tempPtr;

       if (*startPtr == NULL)   /* Lista pusta */
         return 1;
       else  {
         if (uswart == (*startPtr)->dane)  { /* Usuń pierwszy element listy */
                    tempPtr = *startPtr;
                    *startPtr = (*startPtr)->next;
                    free (tempPtr);
         }
       else {/* znajdź w liście element do usunięcia */
                    prevPtr = *startPtr;                  /* początek listy */
                    currPtr = (*startPtr)->next;          /* drugi element*/

                    while (currPtr != NULL && currPtr -> dane != uswart)   {
                              prevPtr = currPtr;
                              currPtr = currPtr -> next;
                    }
                    if (currPtr == NULL)
                              return 1;     /* element nie został znaleziony */
                    else  {  /* Usuń znaleziony element */
                              tempPtr = currPtr;
                              prevPtr->next = currPtr->next;
                              free (tempPtr);
                    }
         }
       }
       return 0;
    }
```

Dołączanie elementu do uporządkowanej listy dwukierunkowej

Uwagi:

- każdy element posiada dodatkowe pole (wskaźnik) *prev*, który dla pierwszego elementu listy jest równe *NULL*;
- lista może być przeglądana w obydwu kierunkach;
- często pamięta się dowiązania do pierwszego i ostatniego elementu;
- należy zawsze uaktualniać dowiązania w obydwu kierunkach;

Dołączanie elementu do uporządkowanej listy jednokierunkowej cyklicznej

Uwagi:

- brak dowiązań wskazujących na NULL;
- w liście jednoelementowej dowiązania wskazują na ten sam element;
- aby uniknąć pętli nieskończonej podczas przeglądania listy, można zastosować wartownika – dowiązanie do pierwszego elementu (umownego);

Zastosowania struktur listowych

- Kolejki LIFO
- Kolejki FIFO

Przykłady listowych struktur danych

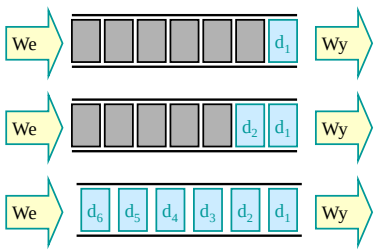
- Definicja listy stanowi podstawę dla zdefiniowania struktury liniowej. Wszystkie przypadki struktur liniowych można zdefiniować bazując na ich odpowiednich reprezentacjach listowych
- Przykłady listowych struktur danych:
  - kolejki,
  - napisy,
  - tablice rzadkie (implementowane dynamicznie),
  - tablice rozproszone (z haszowaniem).

Kolejki

- Kolejka (*queue*) jest strukturą danych wykorzystywaną najczęściej jako bufor przechowujący dane określonych typów.
- Organizacje kolejek:
  - **FIFO** (First In, First Out) - pierwszy element wchodzący staje się pierwszym wychodzącym
  - **Round-Robin** - cykliczna kolejka z dyscypliną czasu obsługi elementów przechowywanych w kolejce (np. w systemach operacyjnych lub sieciach komputerowych)
  - **LIFO** (Last In, First Out) - ostatni wchodzący staje się pierwszym wychodzącym (stos)
  - **kolejki priorytetowe** - dodatkowo w standardowym mechanizmie kolejki uwzględnia się wartości priorytetów przechowywanych danych

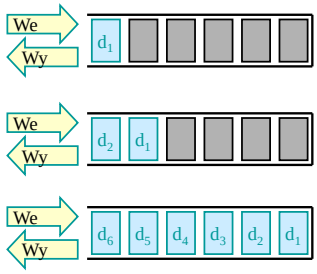
Kolejki FIFO

- Dyscyplina First In First Out:



Stos (kolejka LIFO)

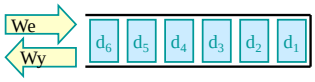
- Dyscyplina Last In First Out:



Realizacje listowych struktur danych

- *Realizacje sekwencyjne* - wtedy, gdy z góry znamy maksymalny rozmiar przetwarzanej struktury liniowej i z góry chcemy zarezerwować dla niej określony zasób (pamięć operacyjna, pamięć zewnętrzna); w czasie wytwarzania programów komputerowych bazujemy wtedy na *zmiennych statycznych*,
- *Realizacje dynamiczne (łącznikowe)* - wtedy, gdy rozmiar struktury nie jest z góry znany i w czasie jej przetwarzania może istnieć konieczność dodawania do niej nowych elementów lub ich usuwania; bazujemy wtedy na *zmiennych dynamicznych (wskaźnikowych)*,
- *Realizacje dynamiczno-sekwencyjne* - połączenie obu powyższych metod - wtedy gdy konieczny jest odpowiedni balans pomiędzy zmiennymi statycznymi i dynamicznymi

Implementacja stosu (kolejki LIFO)



- W praktyce wykorzystuje się wiele różnych implementacji stosu, np.
  - realizacja tablicowa,
  - realizacja dynamiczna (wskaźnikowa), np. z użyciem list
- Typowe operacje na stosie:
  - clear() – wyczyszczenie stosu
  - isEmpty() – sprawdzenie, czy stos jest pusty
  - isFull() – sprawdzenie, czy stos jest pełny
  - push(wart) – włożenie na stos wartości *wart*
  - pop() – zdjęcie ze stosu ostatniego elementu
  - top() – odczytanie (bez zdejmowania) ostatniego elementu

Tablicowa realizacja stosu (kolejki LIFO)

```
#define rozmiar 100
static TypElmStosu  stos[rozmiar];
static int ses =–1;    //szczytowy element stosu


int isFull(void) {   // sprawdzenie, czy stos jest pełny
return ses == rozmiar – 1;
}
int isEmpty(void) {  // sprawdzenie, czy stos jest pusty
return ses == – 1;
}
void clear(void) {   // wyczyszczenie stosu
ses = – 1;
}
```

Zwykłe wywołanie funkcji

Wywołanie funkcji ze zmienną *static*

Tablicowa realizacja stosu (kolejki LIFO)

```
void push(TypElmStosu elm) {   // włożenie na stos elementu elm
   if(!isFull()){
      ses += 1;
      stos[ses] = elm;  }
  else printf("\nNie mozna dodac elementu. Stos jest pelny!");
}
TypElmStosu pop(void) {  // zdjęcie ze stosu ostatniego elementu
  if(!isEmpty())
    ses -= 1;
  else printf("\nNie mozna zdjac elementu. Stos jest pusty!");
}
TypElmStosu top(void) {   // odczytanie ostatniego elementu
  if(!isEmpty())
    return stos[ses];
  else printf("\nNie mozna odczytac wartosci elementu szczytowego. Stos jest
    pusty!");
}
```


Listowa realizacja stosu (kolejki LIFO)

- Stos wygodnie jest implementować za pomocą jednokierunkowej listy niecyklicznej
- Korzeń listy wskazuje na element szczytowy
- Włożenie nowego elementu na stos realizowane jest poprzez dodanie go na początek listy
- Zdjęcie elementu ze stosu polega na usunięciu pierwszego elementu listy
- Stos nie ma wówczas ograniczenia na liczbę elementów (jedynym ograniczeniem jest pojemność pamięci operacyjnej)

Listowa realizacja stosu (kolejki LIFO)

```
struct ElmStosu {
    TypElmStosu wartosc;
    struct ElmStosu  *next;};
static ElmStosu *stos;  // wskaźnik na szczytowy element stosu

int isEmptyl(void) {  // sprawdzenie, czy stos jest pusty
return stos==NULL;
}
void clear(void) {   // wyczyszczenie (usunięcie) stosu
while (!isEmpty())
    pop();
}
```

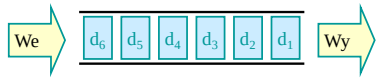

Listowa realizacja stosu (kolejki LIFO)

```
void push(TypElmStosu  wart) {   // włożenie na stos wartości wart
    ElmStosu *new
    new=malloc(sizeof(ElmStosu));
     if(new != NULL){
                new->wartosc=wart;
                new->next=stos;
                stos=new;
    }
  else printf("\nNie mozna dodac elementu. Brak pamieci!");
}
void pop(void) {  // zdjęcie ze stosu elementu szczytowego
 ElmStosu *first;
 if(!isEmpty()) {
    first = stos;
    stos=first->next;
    free(first);
  else printf("\nNie mozna zdjac elementu. Stos jest pusty!");
}
```


Listowa realizacja stosu (kolejki LIFO)

```
TypElmStosu top(void) {   // odczytanie (bez zdejmowania) ostatniego elementu
  if(!isEmpty())
    return stos->wartosc;
  else printf("\nNie mozna odczytac wartosci elementu szczytowego.
  Stos jest pusty!");
}
```

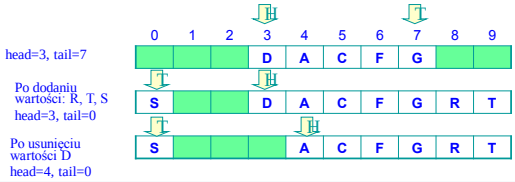

Implementacja kolejki FIFO



- W praktyce wykorzystuje się wiele różnych implementacji kolejki FIFO, np.
  - realizacja tablicowa
  - realizacja dynamiczna (wskaźnikowa), np. z użyciem list lub drzewa binarnego (kopca)
- Typowe operacje na kolejce:
  - clear() – wyczyszczenie (usunięcie) kolejki
  - isEmpty() – sprawdzenie, czy kolejka jest pusta
  - isFull() – sprawdzenie, czy kolejka jest pełna
  - enqueue(wart) – wstawienie do kolejki wartości *wart*
  - dequeue() – usunięcie z kolejki pierwszego elementu
  - first() – odczytanie (bez usuwania) pierwszego elementu

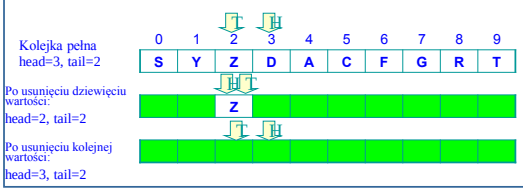
Tablicowa implementacja kolejki FIFO

- Kolejka FIFO jest trudniejsza w implementacji niż kolejka LIFO
- W implementacji tablicowej często stosuje się tzw. tablicę cykliczną
- Wymagane są dwa wskaźniki:
  - *head* – wskazuje na początek kolejki
  - *tail* – wskazuje na koniec kolejki



Tablicowa implementacja kolejki FIFO

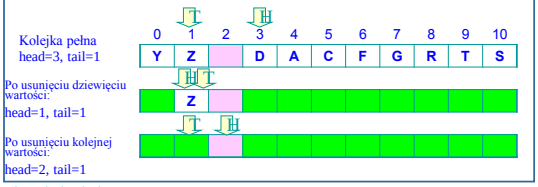
- Wstawienie elementu do kolejki zwiększa *tail* o 1
- Usunięcie elementu z kolejki zwiększa *head* o 1
- W tablicy cyklicznej może być problem z określeniem, czy kolejka jest pusta czy pełna



Tablicowa implementacja kolejki FIFO

Możliwe rozwiązania:

1) wprowadzenie nowej zmiennej, która przechowuje liczbę elementów w kolejce (licznik elementów)
2) użycie tablicy o jedną pozycję dłuższej niż liczba elementów kolejki



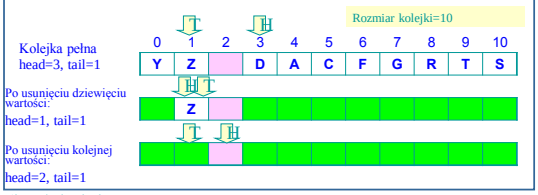
Tablicowa implementacja kolejki FIFO

- Kolejka jest pusta, gdy spełniony jest warunek:

  (tail+1)%RozmiarKolejki==head

- Kolejka jest pełna, gdy gdy spełniony jest warunek:

  (tail+2)%RozmiarKolejki==head



Tablicowa realizacja kolejki FIFO

```
#define RozmiarKolejki  100     //maksymalna liczba elementów w kolejce
#define RozmiarTablicy  (RozmiarKolejki+1)
static TypElmKolejki  kolejka[RozmiarTablicy];
static int head=1;
static int tail=0;

int isFull(void) {   // sprawdzenie, czy kolejka jest pełna
    return (tail+2)% RozmiarTablicy ==head;
}
int isEmpty(void) {  // sprawdzenie, czy kolejka jest pusta
return (head+1)% RozmiarTablicy ==head;
}
void clear(void) {   // wyczyszczenie (usunięcie) kolejki
    head=1;
    tail=0;
```


Tablicowa realizacja kolejki FIFO

```
void enqueue(TypElmKolejki wart) {
if(!isFull()){
    tail=(tail+1)% RozmiarTablicy;
    kolejka[tail]=wart;  }
  else printf("\nNie mozna dodac wartosci. Kolejka jest pelna!");
}
void dequeue(void) {
  if(!isEmpty())
    head=(head+1)% RozmiarTablicy;
  else printf("\nNie mozna pobrac wartosci. Kolejka jest pusta!");
}
TypElmKolejki first(void) {
  if(!isEmpty())
    return kolejka[head];
  else printf("\nNie mozna pobrac wartosci. Kolejka jest pusta!");
}
```


Kolejka priorytetowa

- Implementacja kolejek priorytetowych jest trudniejsza niż „zwykłych" kolejek (o kolejności pobierania elementów z kolejki decyduje ich priorytet);
- Zasad ustalania priorytetów może być wiele.

Kolejka priorytetowa

Przykładowe możliwości implementacji kolejki priorytetowej:

- za pomocą jednej listy nieuporządkowanej (złożoność wstawiania elementu do kolejki wynosi O(1), pobieranie ma złożoność O(n))
- za pomocą jednej listy uporządkowanej wg priorytetów (złożoność wstawiania elementu do kolejki wynosi O(n), pobieranie ma złożoność O(1))
- za pomocą dwóch list: krótkiej, uporządkowanej wg priorytetów i listy nieuporządkowanej pozostałych elementów; liczba elementów na liście krótkiej zależy od tzw. priorytetu progowego i może wynosić np. $\sqrt{n}$; wówczas złożoność wstawiania elementu do kolejki wynosi $O(\sqrt{n})$, pobieranie ma złożoność O(1))
- za pomocą kopca; złożoność operacji wstawiania/pobierania elementu do/z kolejki wynosi O(lg n)

Kopiec jako kolejka priorytetowa

- Kopiec może być podstawą bardzo efektywnej implementacji kolejki priorytetowej;
- Aby wstawić element do kolejki dodaje się go na koniec jako ostatni liść (należy wówczas najczęściej odtworzyć własność kopca);
- Pobieranie elementu z kolejki polega na pobraniu wartości z korzenia; na jego miejsce przesuwany jest ostatni liść (najczęściej trzeba potem odtworzyć własność kopca);
- Implementacja kolejki priorytetowej za pomocą kopca będzie przedmiotem jednego z kolejnych wykładów.