

# Podstawy Programowania

Wykład nr 10: Typy danych użytkownika. Dynamiczne struktury danych.

dr hab. inż. Dariusz Dereniowski

Katedra Algorytmów i Modelowania Systemów  
Wydział ETI, Politechnika Gdańska

## Typ wyliczeniowy enum

- Przypuśćmy, że chcemy w programie zadeklarować zmienne, które będą przechowywać wartości odpowiadające kierunkom.
- Możliwe podejście jest następujące. Deklarujemy stałe, które będą reprezentować poszczególne kierunki:

```
#define LEWO 0  
#define PRAWO 1  
#define GORA 2  
#define DOL 3
```

- Następnie, używamy zmiennych typu `int` do przechowywania wartości.
- Rozwiązaniem bardziej polecanym jest użycie typu `enum`

## Typ wyliczeniowy enum

Typy wyliczeniowy jest „okrojona” formą zmiennych całkowitych (typu int), w których zamiast wartości liczbowych używamy własnych etykiet.

```
enum kierunek_t { /* kierunek_t jest nazwą typu */  
    LEWO,        /* pierwszy element ma wartość 0 */  
    PRAWO,       /* drugi element ma wartość 1 */  
    GORA,        /* itd. */  
    DOL  
} a, b; /* a i b to zmienne typu enum kierunek_t */
```

- Przyrostek “\_t” jest jedynie konwencją: widać wyraźnie, że identyfikator ten jest nazwą typu, co odróżnia go od identyfikatorów stałych, zmiennych, czy funkcji.
- Późniejsza deklaracja zmiennej: `enum kierunek_t a = LEWO;`
- Przypisania: `a = DOL;`
- **Uwaga:** język C++ ma ścisłą kontrolę typu; w szczególności nie można do zmiennej typu `kierunek_t` przypisywać wartości typu `int` (na przykład `a=0` spowoduje błąd kompilacji). Takiej kontroli nie ma w C.

## Typ enum – uwagi

- Stałym typu wyliczeniowego można przypisać własne wartości. We wcześniejszym przykładzie, w deklaracji typu `kierunek_t` możemy zastąpić `GORA` przez `GORA=0`. Oczywiście możemy użyć dowolnej innej liczby typu `int` zamiast 0.
- Wartości liczbowe są przypisywane domyślnie do stałych, którym wartości nie przypisujemy: wartość identyfikatora jest o jeden większa od wartości identyfikatora poprzedniego.
- W wyniku inicjalizacji `GORA=0`, `DOL` przyjmie wartość 1. **Uwaga:** w tej sytuacji `PRAWO == DOL` będzie prawdą!
- Na stałych i zmiennych wyliczeniowych (dotyczy C++) nie można wykonywać operacji arytmetycznych. Na przykład wyrażenia `DOL+1`, `a++` lub `a+3` są niepoprawne, gdzie `a` jest zmienną typu `kierunek_t`.

## Typ agregacyjny (struct)

Struktury pozwalają na zapisywanie rekordów, to jest zapamiętanie **wszystkich** spośród wymienionych w definicji typu pól.

```
struct student_t { /* student_t jest nazwą typu */  
    char imie[15];  
    char nazwisko[25];  
    unsigned int nr_albumu;  
    char rok_studiow;  
} stud1, stud2; /* stud1 i stud2 to zmienne typu struct student_t */
```

- Identyfikator `student_t` po słowie kluczowym `struct` jest opcjonalny. W przypadku jego braku mamy do czynienia z typem anonimowym.
- Identyfikatory `stud1` oraz `stud2` są również opcjonalne: w przypadku ich braku mamy do czynienia jedynie z deklaracją typu.
- Późniejsza deklaracja zmiennych: `struct student_t ktos;`

## Rozróżnialność typów struktur

- Każda deklaracja struktury jest definicją nowego typu, nawet jeśli deklaracja ta jest identyczna do deklarowanej wcześniej struktury. Na przykład:

```
struct a_t { int x; };  
struct b_t { int x; };
```

powoduje utworzenie dwóch różnych typów o nazwach `a_t` i `b_t`.

- W przypadku deklaracji:

```
struct a_t a1, a2;  
struct b_t b;
```

zmienne `a1` oraz `a2` są tego samego typu, natomiast `a1` oraz `b` są innych typów.

- Powyższe oznacza w szczególności, że przypisanie `b = a1`; spowoduje błąd kompilacji.

## Odwołania do składowych struktury

- Metoda podstawowa:

```
nazwa_zmiennej_strukturalnej.nazwa_składowej
```

- Metody pośrednie. Jeśli dysponujemy adresem (wskaźnikiem) zmiennej strukturalnej, to do poszczególnych składowych możemy się odwołać używając jednoargumentowego operatora \*:

```
(*wskaznik).nazwa_skladnika
```

lub używając operatora ->:

```
wskaznik->nazwa_skladnika
```

## Zagnieżdżanie struktur

- Polem składowym struktury może być inna struktura.
- W takich przypadkach struktury zagnieżdżone są zwykle anonimowe.
- Przykład:

```
struct obiekt_t {  
    char nazwa[15];  
    struct {  
        int x;  
        int y;  
    } wspolzedne;  
} o1, o2;
```

Wówczas możliwe odwołania do składowych:

`o1.wspolzedne.x`

`o1.wspolzedne = o2.wspolzedne;`

W drugim przypadku występuje zgodność typów i operator `=` przypisuje wartości składowych struktury `o2.wspolzedne` do odpowiednich składowych struktury `o1.wspolzedne`.



## Inicjalizacja zmiennych strukturalnych

Zmienną strukturalną możemy inicjalizować na różne sposoby:

- Podając w nawiasach klamrowych wartości poszczególnych składowych. Kolejność podania wartości musi być zgodna z kolejnością deklaracji poszczególnych składowych.
- Podając składowe w sposób jawny. Kolejność inicjalizacji jest dowolna. W tej metodzie oraz w metodzie powyższej, składowe którym nie przypisano wartości są inicjalizowane wartością 0. **Uwaga:** metoda dostępna tylko w C.
- Dokonując przypisania wartości innej zmiennej strukturalnej (tego samego typu).

```
struct punkt_t { int x,y; };

int main() {
    struct punkt_t p1 = {2,3}; // metoda 1: podanie składowych
    struct punkt_t p2 = {.y=4,.x=6}; // metoda 2: jawne odwołanie do składowych
    struct punkt_t p3 = p1; // metoda 3: kopiowanie zawartości innej struktury

    struct punkt_t p4 = {2}; // p4.x=2, p4.y=0
    struct punkt_t p5 = {.y=3}; // p5.x=0, p5.y=3

    return 0;
}
```

## Tablice struktur

- Elementami tablicy mogą być struktury. Przykładowa deklaracja takiej tablicy jest następująca:

```
struct typ_t t[N];
```

- Wówczas `t[i]` jest typu `struct typ_t`, gdzie `i` jest indeksem z zakresu od 0 do `N-1`.
- Aby odwołać się do składowej struktury numer `i` w tablicy `t`, piszemy:

```
t[i].składowa
```

- Powyższe jest równoważne `(t[i]).składowa` ze względu na priorytety operatorów.

**Uwaga:** przekazywanie struktur do funkcji odbywa się, tak jak w przypadku każdej innej zmiennej, przez "wartość", tzn. kopia struktury jest przekazywana jako argument. Jeśli polem struktury jest tablica, to następuje kopiowanie tablicy.

# Unie

- Unie deklaruje się analogicznie, jak struktury, przy czym używa się słowa kluczowego `union` zamiast `struct`.
- W przeciwieństwie do struktury, unia nie przechowuje w pamięci wszystkich swoich pól, ale tylko to pole, które zostało zapisane jako **ostatnie**.
- Odwołania do innych pól unii (tzn. wszystkich z wyjątkiem ostatnio zapisanego) mogą prowadzić do nieokreślonego działania.
- Rozmiar zajmowany w pamięci przez unię odpowiada rozmiarowi jej największego pola.

## Unie – przykład

```
#include <iostream>
using namespace std;

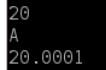
enum typy_t { FLOAT, CHAR };
struct zmienny_typ_t { // W zmiennej typu zmienny_typ_t będziemy zawsze
    enum typy_t typ; // przechowywać jedną wartość, wartość typu float lub char.
    union { // O tym, jakiego typu wartość jest przechowywana w danym
        float liczba; // momencie będzie nas informować zmienna wyliczeniowa typ.
        char znak; // Np, jeśli chcemy przechować zmienną typu float, to przypisujemy
    } wartosc; // ją do pola wartosc.liczba i ustawiamy pole typ na FLOAT.
};

struct zmienny_typ_t przypisz_float( float d ) {
    struct zmienny_typ_t p;
    p.typ = FLOAT; p.wartosc.liczba = d;
    return p;
}

struct zmienny_typ_t przypisz_znak( char c ) {
    struct zmienny_typ_t p;
    p.typ = CHAR; p.wartosc.znak = c;
    return p;
}

void wypisz_wartosc( struct zmienny_typ_t a ) { // Wiemy, że tylko jedno z pól a.wartosc.
    liczba
    switch ( a.typ ) { // lub a.wartosc.znak jest poprawne. Które? O tym mówi pole a.typ.
        case FLOAT: cout << a.wartosc.liczba << endl; break;
        case CHAR: cout << a.wartosc.znak << endl; break;
    }
}

int main() {
    struct zmienny_typ_t a;
    a = przypisz_float( 20 ); wypisz_wartosc( a ); // Wypiszemy 20.
    a = przypisz_znak( 'A' ); wypisz_wartosc( a ); // Wypiszemy znak A.
    cout << a.wartosc.liczba << endl; // to pole nie jest zainicjalizowane!
    return 0;
}
```



20  
A  
20.0001

## Definicje własnych typów: typedef

Słowo kluczowe **typedef** pozwala na definicje własnych typów. Składnia jest następująca:

**typedef typ identyfikator;**

Przykłady:

```
#include<iostream>
using namespace std;

typedef unsigned char bit_t;

typedef struct {
    int x, y;
} punkt_t;

typedef struct {
    char c;
    punkt_t wsp;
} znak_t;

int main() {
    bit_t b = (bit_t)0;
    znak_t t[10];

    cout << sizeof( char ) << endl;
    cout << sizeof( int ) << endl;
    cout << sizeof( punkt_t ) << endl;
    cout << sizeof( znak_t ) << endl;
    return 0;
}
```

1  
4  
8  
12

- Rozmiar struktury jest nie mniejszy niż suma rozmiarów jej składowych.
- Rozmiar struktury może przekraczać sumę rozmiarów jej składowych.
- Powyższy efekt jest związany z tzw. efektem wyrównywania położenia poszczególnych składowych do granicy słowa maszynowego (zwykle 2,4 lub 8 bajtów).
- Powyższy efekt jest podyktowany kwestiami związanymi z wydajnością (szybszy dostęp do pól składowych) przez procesor.

## Unie – zmodyfikowany przykład

```
#include <iostream>
using namespace std;

typedef enum { FLOAT, CHAR } typy_t;
typedef struct {          // W zmiennej typu zmienny_typ_t będziemy zawsze
    typy_t typ;           // przechowywać jedną wartość, wartość typu float lub char.
    union {               // O tym, jakiego typu wartość jest przechowywana w danym
        float liczba;     // momencie będzie nas informować zmienna wyliczeniowa typ.
        char znak;        // Np, jeśli chcemy przechować zmienną typu float, to przypisujemy
    } wartosc;           // ją do pola wartosc.liczba i ustawiamy pole typ na FLOAT.
} zmienny_typ_t;

zmienny_typ_t przypisz_float( float d ) {
    zmienny_typ_t p;
    p.typ = FLOAT; p.wartosc.liczba = d;
    return p;
}

zmienny_typ_t przypisz_znak( char c ) {
    zmienny_typ_t p;
    p.typ = CHAR; p.wartosc.znak = c;
    return p;
}

void wypisz_wartosc( zmienny_typ_t a ) { // Wiemy, że tylko jedno z pól a.wartosc.liczba
    switch ( a.typ ) { // lub a.wartosc.znak jest poprawne. Które? O tym mówi pole a.typ.
        case FLOAT: cout << a.wartosc.liczba << endl; break;
        case CHAR: cout << a.wartosc.znak << endl; break;
    }
}

int main() {
    zmienny_typ_t a;
    a = przypisz_float( 20 ); wypisz_wartosc( a ); // Wypiszemy 20.
    a = przypisz_znak( 'A' ); wypisz_wartosc( a ); // Wypiszemy znak A.
    cout << a.wartosc.liczba << endl; // to pole nie jest zainicjalizowane!
    return 0;
}
```

# Dynamiczne struktury

Przykłady dynamicznych struktur danych:

- tablica o dynamicznie dostosowywanym rozmiarze
- wektor
- lista jednokierunkowa
- lista dwukierunkowa

Przykłady zastosowań dynamicznych struktur danych:

- stos
- kolejka

## Tablica o dynamicznie dostosowywanym rozmiarze

- Tablica w języku C charakteryzuje się rozmiarem zadany przy inicjalizacji i nie może być w łatwy sposób rozszerzona.
- W przypadku, gdy chcemy zwiększyć rozmiar bloku pamięci złożonego z  $a$  komórek, poczynając od adresu wskazywanego przez wskaźnik `ptr`, należy dokonać realokacji:
  - Alokujemy pewną liczbę  $A$  komórek pamięci ( $A > a$ ) pod pewnym adresem `newPtr`. Po tym kroku mamy zaalokowane obydwa bloki pamięci.
  - Przepisujemy zawartość  $a$  kolejnych komórek pamięci spod adresu `ptr` pod adres `newPtr`.
  - Zwalniamy blok pamięci pod adresem `ptr`.
  - Podmieniamy wskaźnik `ptr`, by wskazywał na nowy adres bloku (instrukcja: `ptr = newPtr`).
- Realokacja jest operacją kosztowną czasowo (przepisanie danych), dlatego też należy dokonywać tego typu operacji możliwie rzadko.
- Typowym podejściem pozwalającym na ograniczenie liczby realokacji jest np. podwajanie rozmiaru nowo alokowanego bloku względem rozmiaru bloku poprzedniego.



## Wektor

- **Wektorem** nazywamy dynamiczną strukturę danych, złożoną z pary liczb całkowitych  $a, n$  oraz bloku  $a$  komórek pamięci zaalokowanych pod adresem wskazywanym przez wskaźnik `ptr`.
- Wektor zachowuje się tak, jak tablica  $n$ -elementowa: w zakresie indeksów  $i \in \{0, \dots, n-1\}$ , wartość `ptr[i]` można odczytywać bądź zapisywać, jak dla tablicy.
- Operacja `push_back(x)` powoduje wstawienie wartości  $x$  na koniec wektora, tzn. kolejno:
  - zwiększenie wartości  $n$  o 1,
  - realokację (zwiększenie) bloku pamięci `ptr` w przypadku, gdy  $n > a$ ,
  - dodanie na koniec bloku elementu o wartości  $x$  (`ptr[n-1] = x`).
- Operacja `pop_back()` zwraca zawartość ostatniej zapełnionej komórki pamięci wektora oraz powoduje jej usunięcie, tzn. kolejno:
  - zmniejszenie wartości  $n$  o 1,
  - realokację (zmniejszenie) bloku pamięci w przypadku, gdy  $n << a$  (zazwyczaj, gdy:  $n \simeq a/4$ ).
- Bardzo dobrą wydajność czasową uzyskujemy, gdy przy realokacjach powiększających podwajamy rozmiar bloku `ptr` ( $a = 2*a$ ), a przy zmniejszających – zmniejszać go dwukrotnie ( $a = a/2$ ).

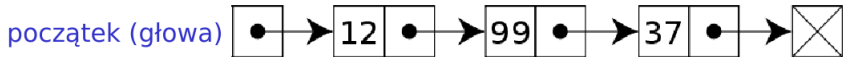
Implementacja wektora oraz funkcji do jego obsługi w języku C: program `w10p05.c` – do pobrania ze strony przedmiotu.

## Lista jednokierunkowa i dwukierunkowa

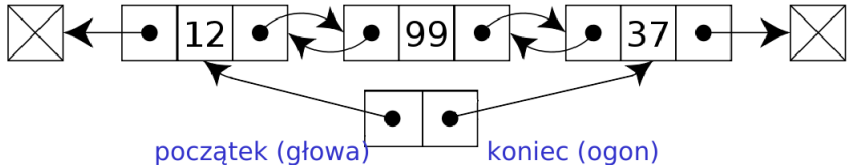
**Listą** nazywamy strukturę danych, w której elementy są uporządkowane liniowo, a każdy element listy obejmuje:

- Wartość w nim przechowywaną.
- Wskaźnik (wskazanie) na kolejny element listy.
- Wskaźnik (wskazanie) na poprzedni element listy – tylko w przypadku list dwukierunkowych.

Ilustracja listy jednokierunkowej:



Ilustracja listy dwukierunkowej:



## Uwagi o listach

- Zmienna „opisująca” listę w programie to tak naprawdę wskaźnik na początkowy element listy. Opcjonalnie, można dodatkowo przechowywać wskaźnik na końcowy element listy i/lub liczbę elementów listy.
- W czasie stałym można zrealizować operację wstawienia bądź usunięcia elementu na początku listy (jak również na końcu listy, o ile dysponujemy wskaźnikiem do niego).
- Lista nie oferuje dostępu swobodnego, tzn. w celu wypisania wartości jej  $j$ -tego elementu należy iterować (przemieszczać się)  $j$  razy po kolejnych elementach listy, rozpoczynając od jej początku.
  - W przypadku list dwukierunkowych możliwe jest również iterowanie odpowiednią liczbę razy od końca listy.
  - Dodanie nowego elementu listy bezpośrednio po elemencie bieżącym (tzn. takim, do którego dysponujemy wskaźnikiem) można zrealizować w czasie stałym.
  - Usunięcie bieżącego elementu listy dwukierunkowej można zrealizować w czasie stałym. W liście jednokierunkowej jest to również możliwe, o ile dysponujemy wskaźnikiem do elementu poprzedzającego.
- Użycie listy najczęściej wiąże się z alokacją każdego elementu listy w odrębnym, oddzielnie alokowanym bloku pamięci.

## Przykład implementacji listy jednokierunkowej – definicja i inicjalizacja

```
typedef struct list {
    int value;
    struct list *next;
} list_t;

// Inicjalizuje listę o głowie "head".
void init( list_t *head ) {
    head->next = NULL;
}
```

- Element typu `list_t` zawiera “pole danych”, które nazywamy `value` oraz wskaźnik `next` na tego typu strukturę.
- Jeśli dysponujemy wskaźnikiem `ptr` na strukturę typu `list_t`, która jest elementem listy, to mamy dwie możliwości:
  - Jeśli `ptr == NULL`, to `ptr` wskazuje na ostatni element listy.
  - Jeśli `ptr != NULL`, to `ptr` nie jest ostatnim elementem listy. Wówczas, adres kolejnego elementu listy to `ptr->next`.

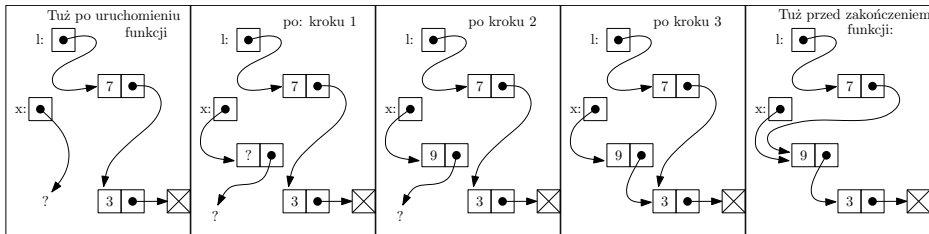
Tą własność musimy zagwarantować odpowiednią implementacją funkcji, które wstawiają i usuwają elementy listy.

- W prezentowanej implementacji, pierwszy element listy ma nieokreśloną wartość pola `value`. Implementacja jest wykonana w ten sposób, aby wstawianie i usuwanie elementów nigdy nie dokonało zmiany pierwszego jej elementu. Takie podejście nie jest konieczne, lecz ułatwia przekazywanie parametrów do funkcji podanych na kolejnych slajdach.

## Przykład implementacji listy jednokierunkowej – wstawianie elementu

```
void add_after( list_t *l, int value ) {  
    list_t *x = (list_t *) malloc( sizeof(list_t) ); // Krok 1.  
  
    x->value = value;      // Krok 2.  
    x->next = l->next;    // Krok 3.  
    l->next = x;           // Krok 4.  
}
```

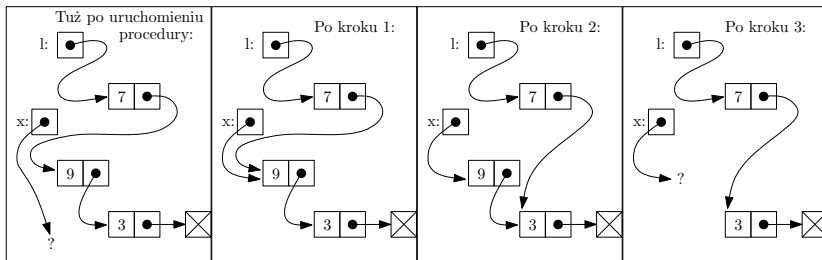
- Załóżmy, że wywołujemy `add_after( l, 9 )`, gdzie `l` wskazuje na pewien element listy.
- Funkcja “wstawi” nowy element listy tak, by był następnikiem elementu wskazywanego przez `l`.
- Nowy element będzie miał wartość `value` równą 9.
- **Uwaga:** funkcję warto uzupełnić o sprawdzenie czy wywołanie `malloc` powiodło się.
- Kolejne etapy działania procedury można przedstawić następująco:



## Przykład implementacji listy jednokierunkowej – usuwanie elementu

```
void remove_after( list_t *l ) {  
    if ( l->next != NULL ) {  
        void* x = l->next;           // Krok 1.  
        l->next = l->next->next;      // Krok 2.  
        free( x );                    // Krok 3.  
    }  
}
```

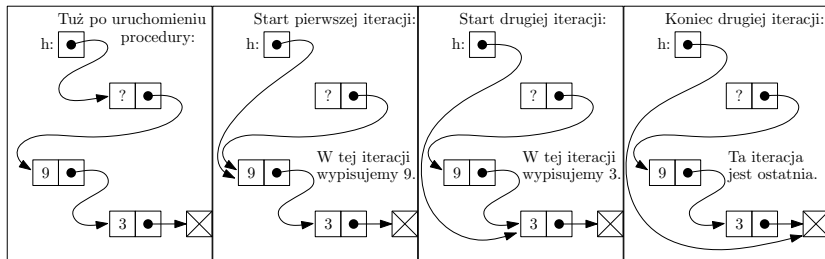
- Załóżmy, że wywołujemy `remove_after( l )`, gdzie `l` wskazuje na pewien element listy.
- Funkcja “usunie” z listy element wskazywany przez `l->next`, czyli “następnik” `l`, o ile takowy istnieje. Należy pamiętać o zwolnieniu pamięci.
- Zauważ, że nie możemy od razu wywołać `free(l->next)`, gdyż wówczas nie moglibyśmy “powiązać” elementów listy.
- Kolejne etapy działania procedury można przedstawić następująco:



## Przykład implementacji listy jednokierunkowej – przeglądanie listy

```
void print( list_t *h ) {  
    h = h->next;  
    while ( h != NULL ) {  
        printf ( "%d ", h->value);  
        h = h->next;  
    }  
}
```

- Załóżmy, że wywołujemy `print( h )`, gdzie `h` wskazuje na “głową” listy.
- Zgodnie z naszą implementacją, głowa listy zawiera nieokreśloną wartość pola `value`.
- Zauważ, że funkcja dysponuje kopią adresu głowy listy, więc modyfikując `h` nie zmieniamy wartości adresu przechowanej w zmiennej przekazanej jako argument.
- Kolejne etapy działania procedury można przedstawić następująco:



# Stos

**Stosem** nazywamy strukturę danych, udostępniającą dwie operacje:

- Operacja **push(x)** powoduje wstawienie wartości x na wierzchołek stosu.
- Operacja **pop()** zwraca wartość znajdującą się na wierzchołku stosu oraz usuwa tę wartość ze stosu.

Uwagi:

- Wartości zawsze zdejmowane są ze stosu w kolejności odwrotnej do kolejności ich umieszczania na stosie (LIFO – last in, first out).
- Wektor może zostać użyty do implementacji stosu (**push\_back** realizuje operację **push**, **pop\_back** realizuje **pop**).
  - Zaleta implementacji stosu na wektorze: jeden ciągły blok pamięci.
  - Wada: operacje **push\_back**/**pop\_back** średnio rzecz biorąc działają szybko, ale czasem mogą trwać bardzo długo (gdy wykonuje się realokacja pamięci).
- Lista jednokierunkowa może zostać użyta do implementacji stosu:
  - Każdy element stosu jest umieszczany w odrębnym, alokowanym dla niego bloku pamięci. Wada: wielokrotne alokacje/realokacje małych bloków pamięci.
  - Czas wykonania każdej operacji **push**/**pop** jest wówczas stały.

Implementacja stosu z wykorzystaniem listy jednokierunkowej: program w10p07.c – do pobrania ze strony przedmiotu.



# Kolejka

**Kolejką** nazywamy strukturę danych, udostępniającą dwie operacje:

- Operacja **add(x)** powoduje wstawienie wartości x na koniec kolejki.
- Operacja **remove()** zwraca wartość znajdującą się na początku kolejki oraz usuwa tę wartość z kolejki.

Uwagi:

- Wartości zawsze usuwane z kolejki w kolejności ich umieszczania w kolejce (FIFO – first in, first out).
- Lista jednokierunkowa może zostać użyta do implementacji kolejki:
  - Każdy element kolejki jest umieszczany w odrębnym, alokowanym dla niego bloku pamięci. Wada: wielokrotne alokacje/realokacje małych bloków pamięci.
  - Operacja **add** dodaje element na koniec listy – powinniśmy dysponować adresem ostatniego elementu, aby operację tą wykonywać szybko.
  - Operacja **remove** usuwa pierwszy element listy.
  - Czas wykonania każdej operacji **add/remove** jest wówczas stały.
- Wektor nie nadaje się do implementacji kolejki.
- Implementacja kolejki przy użyciu tablicy jest możliwa. Stosuje się wówczas tzw. mechanizm cykliczny.