Struktury

Struktury pozwalają na grupowanie zmiennych różnych typów pod wspólną nazwą. To istotnie ułatwia organizacje danych, które okazują się w jednym miejscu kodu programu. To jest bardzo ważne dla dużych programów, dla tego że pozwala uporządkować kod. Jest koniecznym warunkiem (ale nie wystarczającym) obiektowego oprogramowania – dane, zebrane jako elementy struktury, tworzą obiekty danych (nie wystarcza im metod – funkcji, które zabezpieczają działania nad danymi).

Składowymi struktur mogą być zmienne dowolnych typów, tablice i struktury również.

Przykład W8_1

Sposoby definicji struktur

1.

```
struct STUDENT_ACCOUNT  //definicja struktury
{
        int rok;
        char name[128];
        char lastname[128];
};

//deklaracja obiektów o nazwie s1, s2  typu struct STUDENT_ACCOUNT
struct STUDENT_ACCOUNT s1, s2;
```

2.

```
struct STUDENT_ACCOUNT
{
    int rok;
    char name[128];
    char lastname[128];
}s1, s2;
//definicja struktury oraz deklaracja obiektów o nazwie s1, s2  typu
struct STUDENT_ACCOUNT
```

3. Deklarujemy wskaźnik do struktury typu STUDENT_ACCOUNT

```
STUDENT_ACCOUNT  //definicja struktury
{
     int rok;
     char name[128];
     char lastname[128];
};

//deklaracja wskaźnika – wydzielono 4 B dla wskaźnika.
//dla struktury pamięć nie pozostała zaalokowaną.
STUDENT_ACCOUNT *ptr_s = NULL;

//alokujemy pamięć
ptr_s = (STUDENT_ACCOUNT *)malloc(sizeof(STUDENT_ACCOUNT));
if(!ptr_s)
{
 …….
}

ptr_s->rok = 1988;
.....
free(ptr_s);
ptr_s = NULL;
```

1. Odwołanie do elementów struktury:

```
STUDENT_ACCOUNT s1;
s1.rok = 2007;
printf("name = %s\n", s1.name);
sprintf(s1.lastname, "Kowalski");
```

2. Odwołanie do elementów struktury przez wskaźnik do struktury:

```
STUDENT_ACCOUNT *ptr_s = NULL;
ptr_s = (STUDENT_ACCOUNT *) malloc(sizeof(STUDENT_ACCOUNT));
if(!ptr_s)
{
.....
}

ptr_s->rok = 2007;
printf("name = %s\n", ptr_s-> name);
sprintf(ptr_s-> lastname, "Kowalski");

free(ptr_s);
ptr_s = NULL;
```

3. Odwołanie do elementów tablicy struktur:

```
STUDENT_ACCOUNT *ptr_s = NULL;
ptr_s = (STUDENT_ACCOUNT *) malloc(10*sizeof(STUDENT_ACCOUNT));
if(!ptr_s)
{
.....
}

for(i=0; i<10; i++)
{
    ptr_s[i].rok = 2007;
    printf("name = %s\n", ptr_s[i].name);
    sprintf(ptr_s[i].lastname, "Kowalski");
}

free(ptr_s);
ptr_s = NULL;

//id_wskaźnik[element_tablicy].id_pola
```

Odwołanie do elementów tablicy typu STUDENT_ACCOUNT za pomocą wskaźnika.

```
STUDENT_ACCOUNT tab[5], *ptr = NULL;
….
for(ptr = tab, int i = 0; ptr != tab + 5; ++ptr, ++i)
{
        sprintf(ptr->name, “my_name”);  //definiujemy dane
        …
        ptr->rok = 20 + i;
}
```

Operator przypisania = , a również funkcja *memcpy, memset* są bardzo niebezpieczne dla struktur.

Przykład WW_8.3.

Na poziomie języka C trzeba pisać funkcje dla kopiowania pól struktur takiego samego typu.

W8

- **Przekazywanie obiektu do funkcji przez wartość. Osobliwości C++.**
- **Przekazywanie obiektu do funkcji przez wskaźnik.**
- **Funkcja zwraca obiekt. Osobliwości C++.**

Struktury danych: *Tablica, Kolejka, Stos, Lista, Drzewo binarne.*

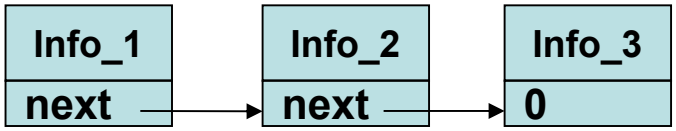
- **Tablica – zajmuje ciągłe bajty pamięci. Odbywa się bezpośredni (najszybszy) dostęp do elementów (obiektów) tablicy.**

- **Inne struktury – tylko sekwencyjny dostęp. Dla dostępu do elementu o numerze *n* trzeba sekwencyjnie pobrać poprzednie *n – 1* elementów.**

- **Kolejka – dostęp do elementów na podstawie FIFO (first – in, first – out). Element, który był pierwszym umieszczony w kolejce, pierwszym będzie pobrany. Dostęp do danych tego elementu jest możliwy tylko przy pobraniu. Pobranie jest możliwe tylko przy usunięciu elementu z kolejki. Chronimy tylko wskaźnik do First. Dostęp mamy tylko do First. Żeby pobrać kolejny element, musimy najpierw pobrać First, a dalej przestawić First do kolejnego elementu.**

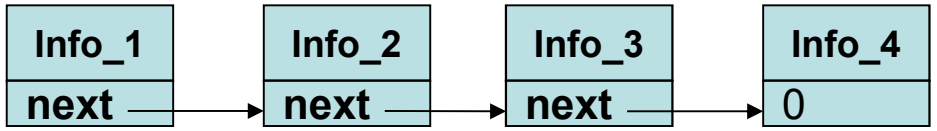
Przykład Queue !!

Kolejka



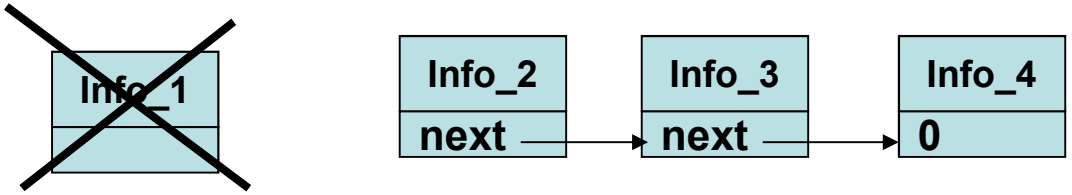
First

Dodajemy element do kolejki (push)



First

usuwamy element z kolejki (pop)

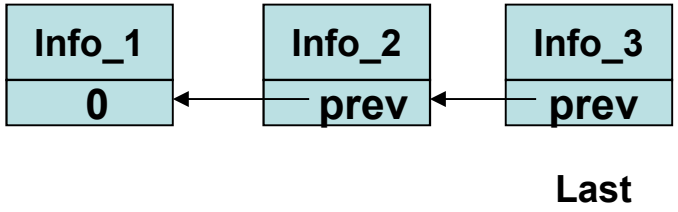


First

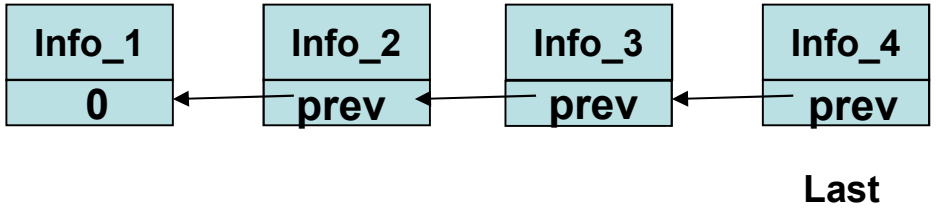
Stos - dostęp do elementów na podstawie LIFO (last – in, first – out). Element, który był umieszczony w stosie ostatnim, pierwszym będzie pobrany. Dostęp do danych tego elementu jest możliwy tylko przy pobraniu. Pobranie jest możliwe tylko przy usunięci elementu ze stosu.

Chronimy tylko wskaźnik do Last. Dostęp mamy tylko do Last. Żeby pobrać kolejny element, musimy najpierw pobrać Last, a dalej przestawić Last do kolejnego elementu.

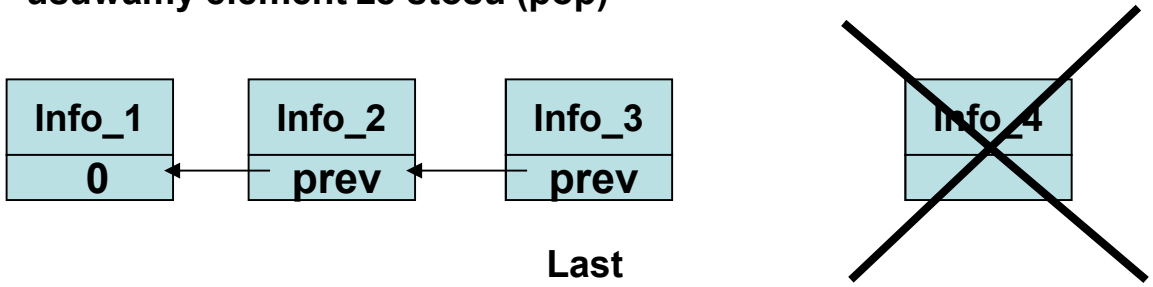
Stos



Dodajemy element do stosu (push)



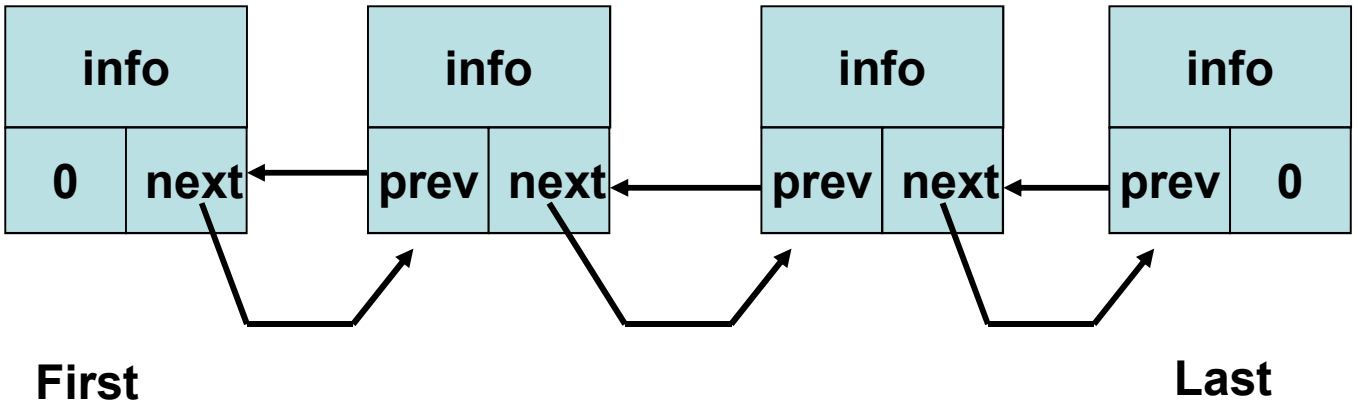
usuwamy element ze stosu (pop)

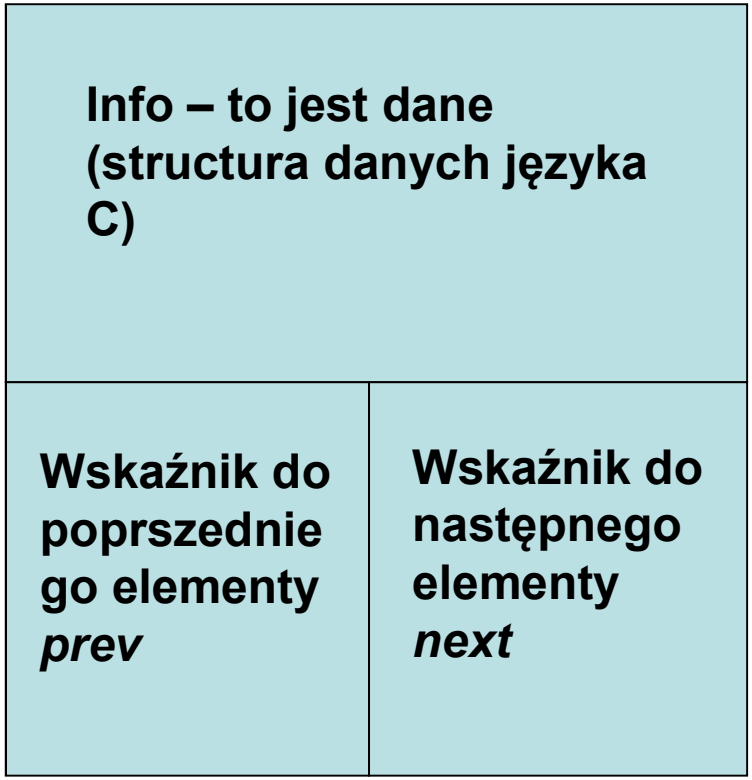


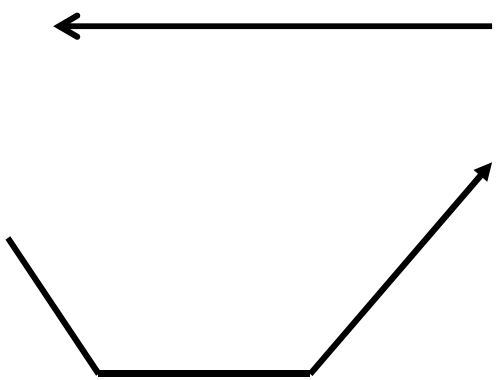
Lista dwustronna. Każdy element ma referencje do poprzedniego i do następnego. Pobranie dowolnego elementu nie oznacza usunięcie jedo z listy. Dla usunięcia elementu z listy trzeba osobne działanie.

- **Listę dwustronną można przeglądać jako od początku do końca, tak i odwrotnie. (Na różnice od tablicy lista nadaje możliwość tylko sekwencyjnego dostępu do elementów).**
- **Przy uszkodzeniu danych łatwiej odnowić.**
- **Łatwiejsze wykonanie wielu działań.**

To jest tak zwana dynamiczna struktura danych – ilość elementów staje wiadoma tylko przy wykonaniu programu.







Wskazanie do elementów listy

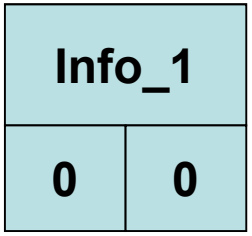
Każdy element zawiera:
- **Dane**
- **Wskaźnik do poprzedniego elementu**
- **Wskaźnik do następnego elementu**

Przykład W8_2_1

First – wskaźnik do początku listy. Last - wskaźnik do końca listy

Dodawanie elementów listy

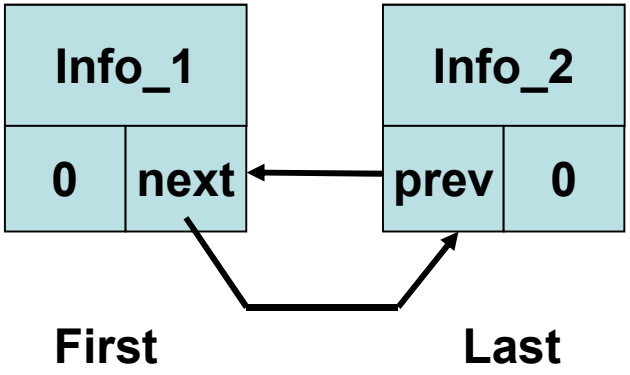
1. **Lista pusta. First = NULL, Last = NULL.**
2. **Dodajemy pierwszy element:**



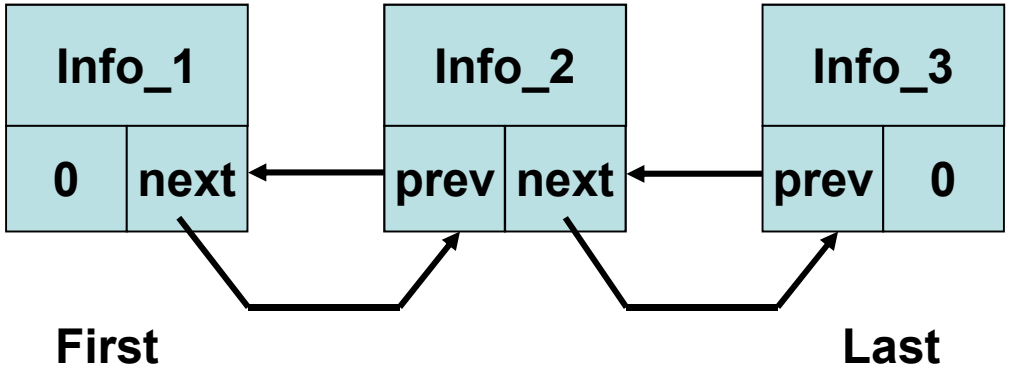
First
Last

Teraz First i Last wskazują do pierwszego elementu

3. Dodajemy drugi element:

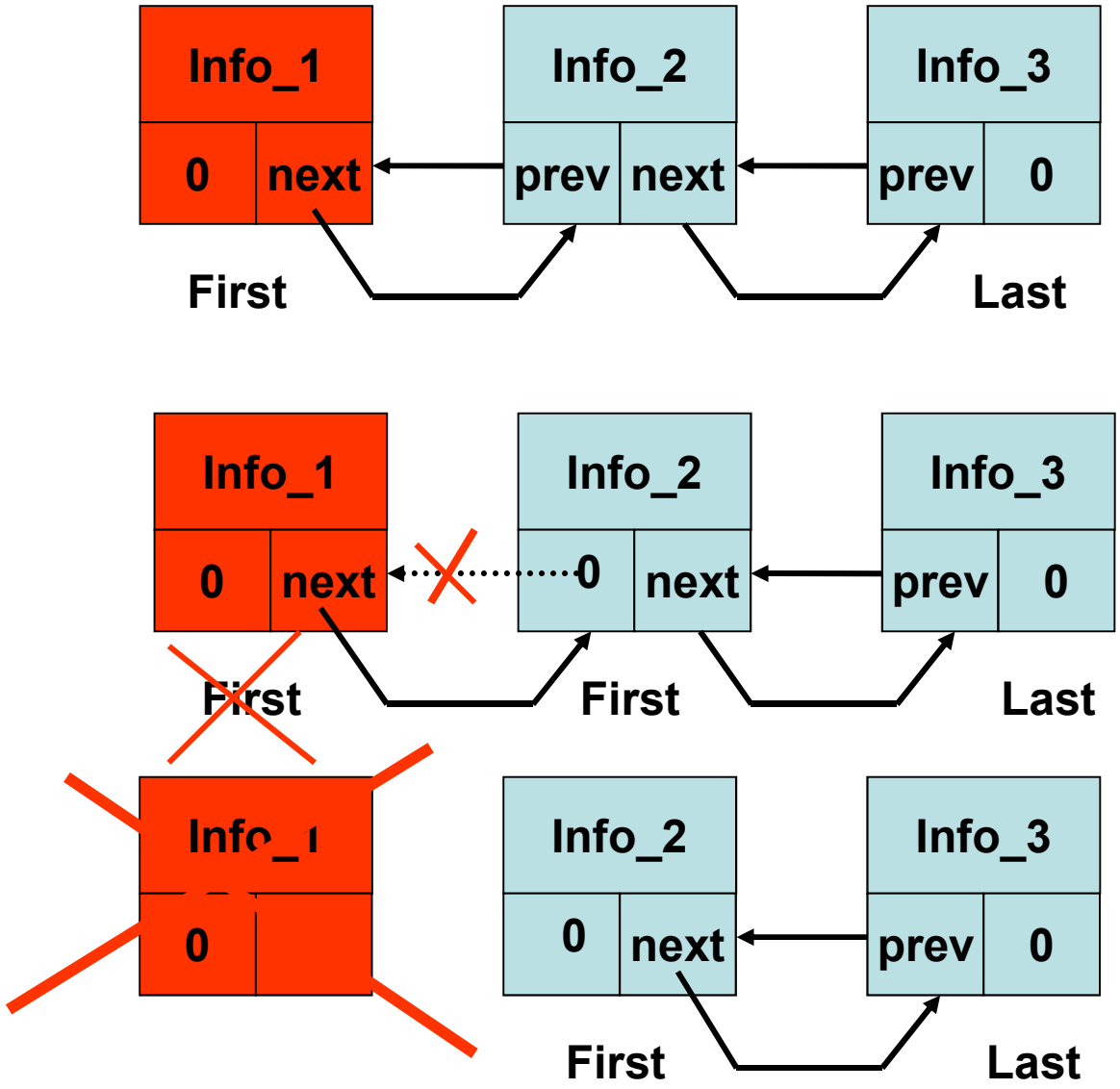


4. Dodajemy trzeci element:



Usunięcie elementów listy

1. Usunięcie elementu First



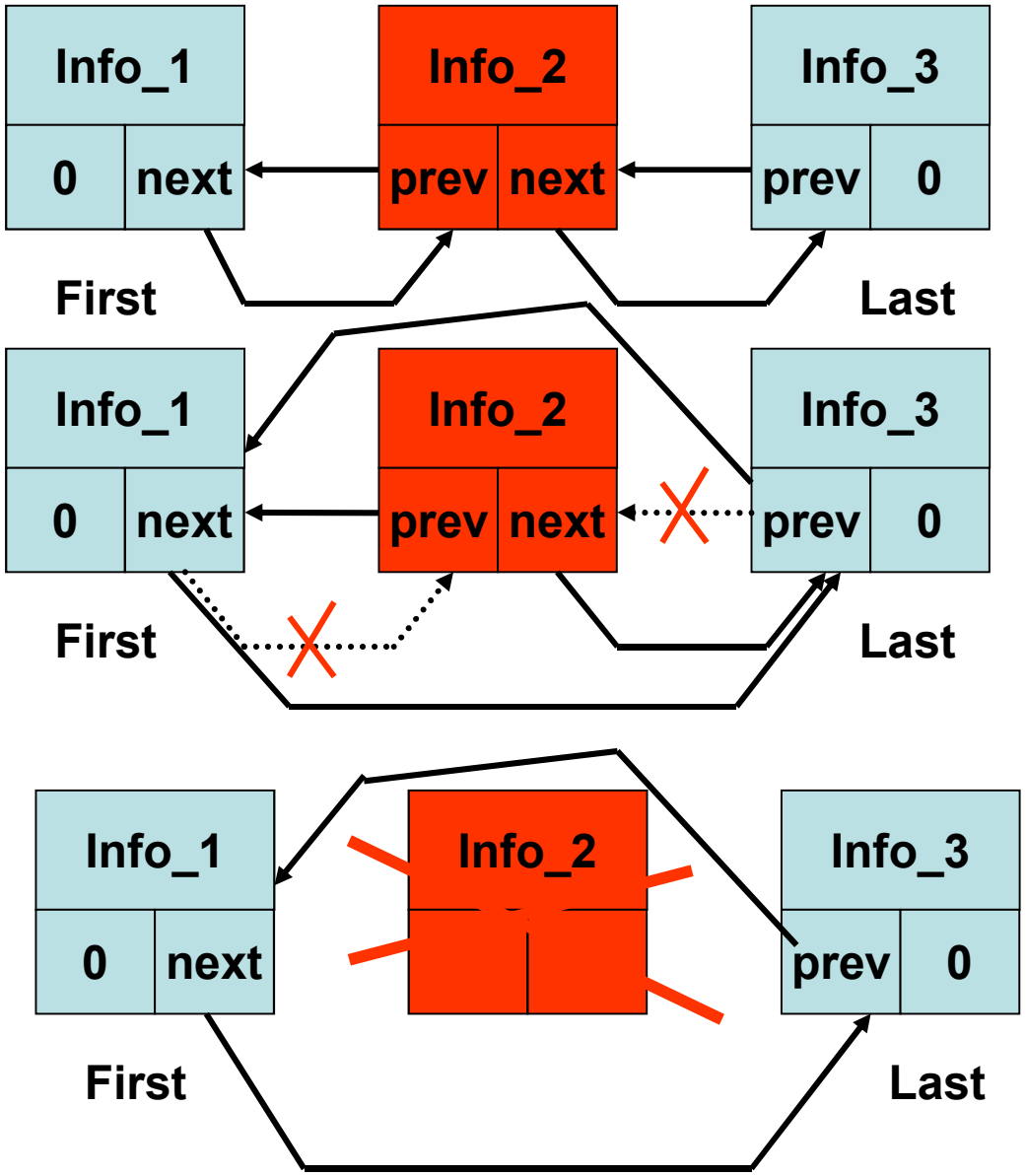
cur - pointer to current element of List.

cur is a First.

1. cur->next->prev = NULL;
2. First = cur->next;
3. Free data in cur and remove cur;

Usunięcie elementów listy

1. Usunięcie elementu !First i !Last



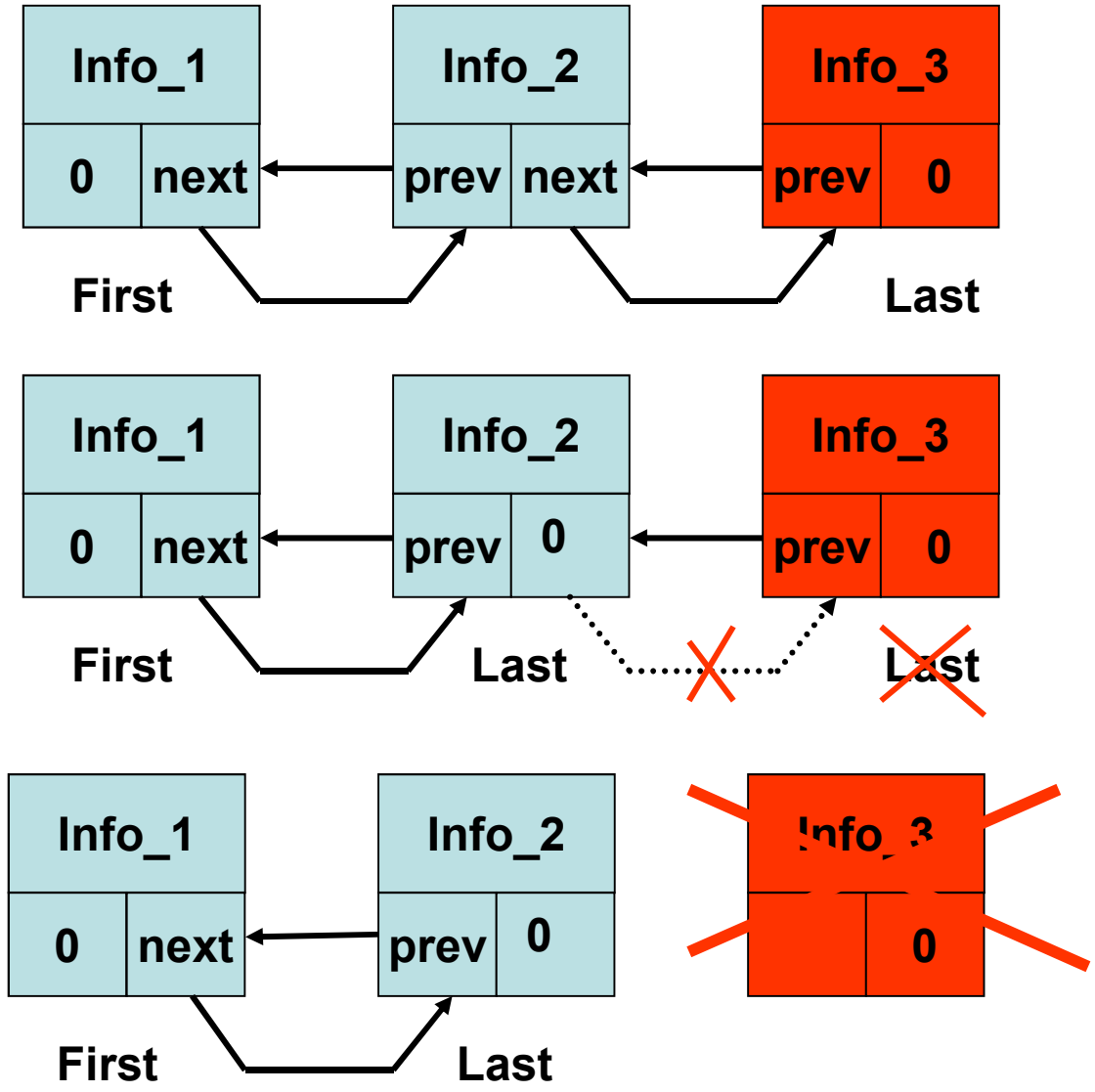
cur - pointer to current element of List.

cur is not First and cur is not Last.

1. cur->prev->next = cur->next;
2. cur->next->prev = cur->prev;
3. Free data in cur and delete cur;

Usunięcie elementów listy

1. Usunięcie elementu Last



cur - pointer to current element of List.

cur is a Last.

1. Last = cur->prev;
2. Last->next = NULL;
3. Free data in cur and delete cur;

Algorytmy (metody) obsługiwania listy:

```
struct STUDENT_ACCOUNT
{
        int rok;        //rok urodzenia
        char name[256];  //imie
        char sname[256]; //nazwisko
        STUDENT_ACCOUNT *prev;
        STUDENT_ACCOUNT *next;
};
int LIST_Add(STUDENT_ACCOUNT s);
void LIST_Remove(STUDENT_ACCOUNT *s);
STUDENT_ACCOUNT *LIST_Find(STUDENT_ACCOUNT s);
void LIST_LookForward();
int LIST_StoreToDisk();
int LIST_ReadFromDisk();
void LIST_Erase();
void LIST_Free();
```

Funkcje obsługi plików binarnych

Zapis danych do pliku binarnego:

size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);

buffer - bufor, ktory piszemy na dysk
size - rozmiar jednostki zapisu w B
count - ilosc zapisow
stream - wskaznik do pliku
zwraca ilosc zapisow, ktore aktualne pozostale zapisane

Przykład: piszemy 10 zapisów danych typu STUDENT_ACCOUNT w plik, na który wskazuje pf

```
size_t count_written;
count_written = fwrite((const void *)tmp, sizeof(STUDENT_ACCOUNT), 10, pf);
if(count_written != 10)
        return 0;  // błąd!
```

Funkcje obsługi plików binarnych

Czytanie danych z pliku binarnego:

size_t fread(void *buffer, size_t size, size_t count, FILE *stream);

buffer - bufor, ktory piszemy na dysk
size - rozmiar jednostki zapisu w B
count - ilosc zapisow
stream - wskaznik do pliku
zwraca ilosc zapisow, ktore aktualne pozostale zapisane

Przykład: czytamy 10 zapisów danych typu STUDENT_ACCOUNT z pliku, na który wskazuje pf, w bufor tmp;

```
STUDENT_ACCOUNT *tmp = (STUDENT_ACCOUNT *)malloc(10*sizeof(STUDENT_ACCOUNT ));
if(!tmp)
        return 0;  //błąd !

size_t count_read;
count_read = fread((void *)tmp, sizeof(STUDENT_ACCOUNT), 10, pf);
count_read!= 10)
        return 0;  // błąd!
```

Funkcje obsługi plików binarnych

Ustawienie wskaźnika pozycji pliku:

int fseek(FILE *stream, long offset, int origin);

Przesuwa wskaźnik pliku na *offset* bajtów od *origin*.

Origin:

SEEK_CUR - od bieżącej pozycji
SEEK_END - od końca pliku
SEEK_SET - od początku plku

Zwraca 0, jeśli OK.

long ftell(FILE *stream);

Zwraca wskaźnik bieżącej pozycji pliku.

Funkcje obsługi plików binarnych

Przykład:

```
//otworzymy pusty plik na zapis i czytanie. Jeśli plik już istnieje, będzie
//wyczyszczony.
FILE *pf = fopen("my_plik_data.inf", "w+b");
.....
long plik_pos[10];
size_t count_written, count_read;
plik_pos[0] = ftell(pf);   // plik_pos[0] wskazuje na początek zapisu obj_1 w pliku

count_written = fwrite((const void *)&obj_1, sizeof(OBJEKT_1), 1, pf);
if(count_written != 10)
        return 0;  // błąd!
plik_pos[1] = ftell(pf);   // plik_pos[1] wskazuje na początek następnego  zapisu
.....
//przesuwamy wskaźnik pozycji pliku do czytania (zapisu) obj_1
if(fseek(pf, plik_pos[0], SEEK_SET) != 0)
        return 0;  //błąd !

count_read = fread((void *)&obj_1, sizeof(OBJEKT_1), 1, pf);
if(count_read != 1)
     return; //błąd !
```

typedef *type-declaration synonym*;

Deklaracja typedef nadaje możliwość wprowadzenia swoich własnych nazw istniejących typów danych. Nadaje możliwość zmian istniejących typów danych bez istotnych zmian w kodzie. Oprócz tego robi kod bardziej czytelnym z punktu widzenia merytorycznego.

type-declaration **- jeden z istniejących typów danych.**

Synonym **- nowa nazwa.**

Przykład 1.

```
typedef long GRAPHNODE;
.....................................
GRAPHNODE it;              //to jest long it;
GRAPHNODE arr[512];        //to jest long arr[512];
GRAPHNODE *ptr_a;          //to jest long *ptr_a;
```

To oznacza, że wszędzie, gdzie używana definicje typu GRAPHNODE, faktycznie jest zastosowany typ long. GRAPHNODE – to synonim long. Jeśli wynika konieczność zmiany typu bazowego long na __int64, wystarczy tylko w jednym miejscu programy zmienić

```
typedef __int64 GRAPHNODE;
.....
GRAPHNODE it;          //to jest __int64 it;
GRAPHNODE arr[512];    //to jest __int64 arr[512];
GRAPHNODE *ptr_a;      //to jest __int64 *ptr_a;
```

Przykład 2.

```
typedef struct MY_TYP_DATA
{
        int i;
        double a;
} MYSTRUCT;

int main()
{
        MYSTRUCT ms;
        ms.i = 10;
        ms.a = -1.0e-4;
        .....
        return 0;
}
```

Przykład 3. (To jest to samo, co przykład 2)

```
struct MY_TYP_DATA
{
        int i;
        double a;
};

typedef   struct   MY_TYP_DATA   MYSTRUCT;

int main()
{
        MYSTRUCT ms;
        ms.i = 10;
        ms.a = -1.0e-4;
        ..........................
        return 0;
}
```

Wskaźnik do funkcji.

Jest możliwe tylko dwa działania z funkcjami – wywołanie i pobranie adresu.

Pobranie adresu:

```
void fun(char *p);      //prototyp funkcji
//deklaracja wskaźnika do funkcji, która zwraca void i ma listę argumentów
//char *p:
void (*efct)(char *p);

int main()
{
        efct = &fun;            //efct ma teraz adres funkcji fun
        (*efct)("błąd.....");   //wywołanie funkcji przez wskaźniku
        ..........................................................................
}

void fun(char *p)
{
.....................
}
```

Przekazanie argumentów przy wywołaniu funkcji przez wskaźnik jest dokładnie takim samym, jak i przy wywołaniu zwykłym.

Przykład W8_2

Definicja tablicy wskaźników do funkcji:

```
//prototypy funkcji, które mają taką samą listę argumentów formalnych i
//zwracają ten samy typ

int fun(double a);
int fun_1(double a);

//definiujemy tablice wskaźników i inicjalizujemy
int (*tabfun[ ])(double a) = {&fun, &fun_1};

.....
//wywołanie:
int ret  =  (*tabfun[0])(10.0);    //wywołujemy fun
int ret1 =  (*tabfun[1])(20.0);    //wywołujemy fun_1
```

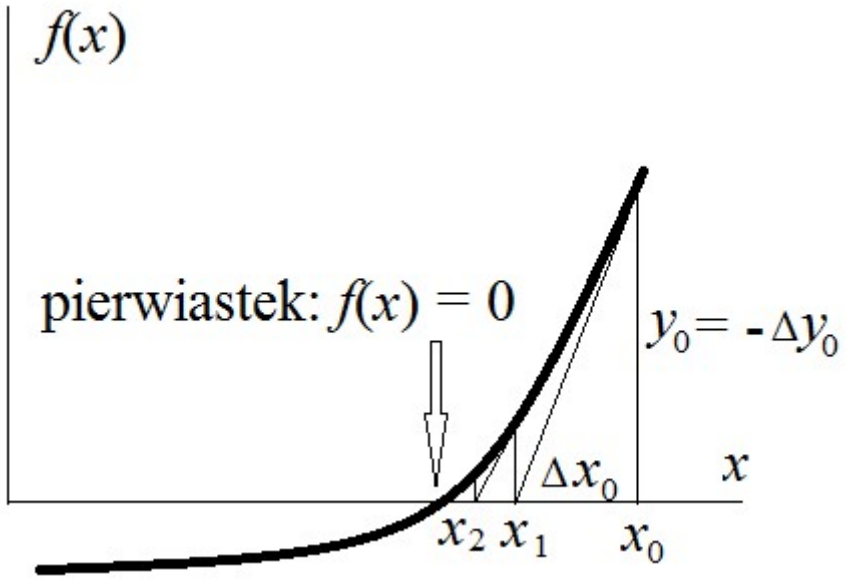
Zagadnienie:

Dla podanej funkcji

$y = f(x)$

odnaleźć takie x**,**

dla których $f(x) = 0$.



Metoda Newtona:

zakładamy, że my znamy wartość x, **bliską do rozwiązania** $x' = x + \Delta x$.

Linearyzacja:

$$y + \Delta y = f(x + \Delta x) \approx f(x) + \left.\frac{df(x)}{dx}\right|_x \cdot \Delta x = 0;$$

$$y + \Delta y = 0 \quad \rightarrow \quad \Delta x = -\frac{f(x)}{\left.\frac{df(x)}{dx}\right|_x};$$

Algorytm:

$x_0 = x0; \quad i = 0$

while(err > tol)

$y = f(x_i)$

$err = |y|$

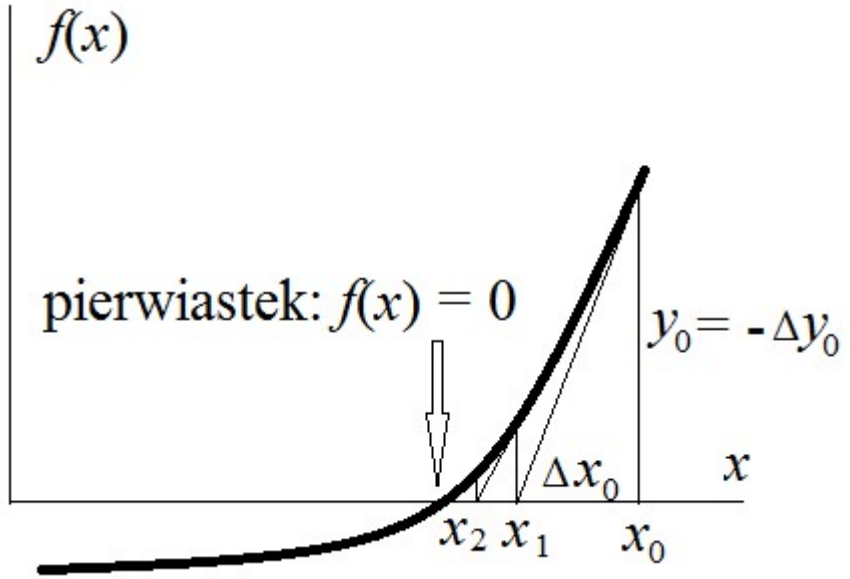
$if(err \le tol) \quad break;$

$\Delta y = -f(x_i)$

$$\Delta x = \frac{\Delta y}{\left(\frac{df}{dx}\Big|x_i\right)}$$

$x_{i+1} = x_i + \Delta x$

end _ while



Przykład W8_2_3