

Свързани списъци

Пламен Танов
Ненко Табаков

Технологично училище „Електронни системи“
Технически университет – София

версия 0.1

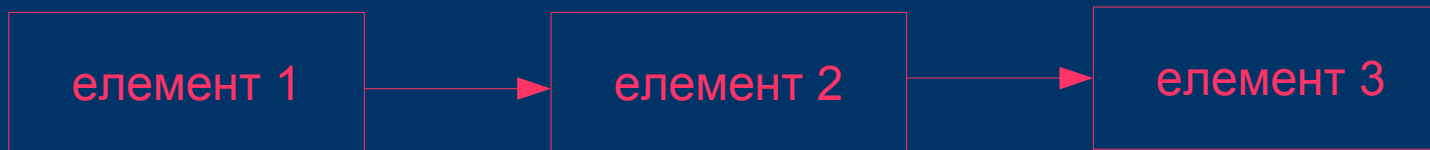
Въведение

- Свързани списъци се използват, когато е необходимо да се обхоят последователно (един по един) съвкупност от елементи
 - Свързаният списък е структура от данни, в която всеки елемент съдържа информацията, необходима за достигане на следващ елемент
 - Главното предимство на свързаните списъци е че дават възможност ефективно да се пренареждат елементите им
-
-

Дефиниция₁

Свързан списък е множество от елементи, при което всеки елемент е част от възел, който също съдържа връзка към възел

„Възел“ представлява указател към следващия елемент, така че към свързаните списъци понякога може да се отнасяме и като към самосвързани структури



Дефиниция₂

Свързаните списъци представляват последователна подредба на множество елементи. Започваме от даден възел (неговият елемент се разглежда като пръв в редицата) ако последваме неговата връзка към друг възел то стигаме до втория елемент в списъка и т.н. до края му
В зависимост от крайния елемент списъците биват:

- Цикличен – последната връзка сочи към първия елемент
 - Последната връзка сочи към нулев елемент
 - Последната връзка сочи към фалшив елемент
-
-

Елементи на свързан списък

Един възел се състои от връзка към следващия възел и елемент.

Връзката е указател към възел, а елементът представлява някаква променлива, която съдържа самите данни.

```
typedef int Item;//или някакъв друг тип (float, struct, ...)
typedef struct node * Link;//Link е указател към struct node
struct node {
    Item item;//съдържанието на дадения елемент
    Link next;//връзка към следващия елемент
};
```

Създаване на свързан списък

Заделянето на памет е главно условие за ефективната употреба на свързани списъци. В общия случай не се знае броят на възлите. Винаги когато трябва да добавим нов възел е необходимо да се създаде нов екземпляр на структурата **node**

```
typedef int Item; //или някакъв друг тип (float, struct, ...)
typedef struct node * Link; //Link е указател към struct node
struct node {
    Item item; //съдържанието на дадения елемент
    Link next; //връзка към следващия елемент
};
...
//x е указател към struct node
Link x = (Link) malloc(sizeof(*x));
//(*x) е struct node, т.е. sizeof(*x) е sizeof(struct node)
```

Действия със свързан списък

Свързаните списъци предоставят лесен и удобен начин за триене и добавяне на елементи към тях. Това е и основната причина за съществуването им. Съответните операции при масиви са неестествени и неудобни, защото изискват преместване на цялото съдържание на масива в ляво или дясно.

Обратно, свързаните списъци не са добре пригодени за намиране на k -тия елемент в даден списък, докато в масив k -тия елемент се намира като просто се напише $a[k]$

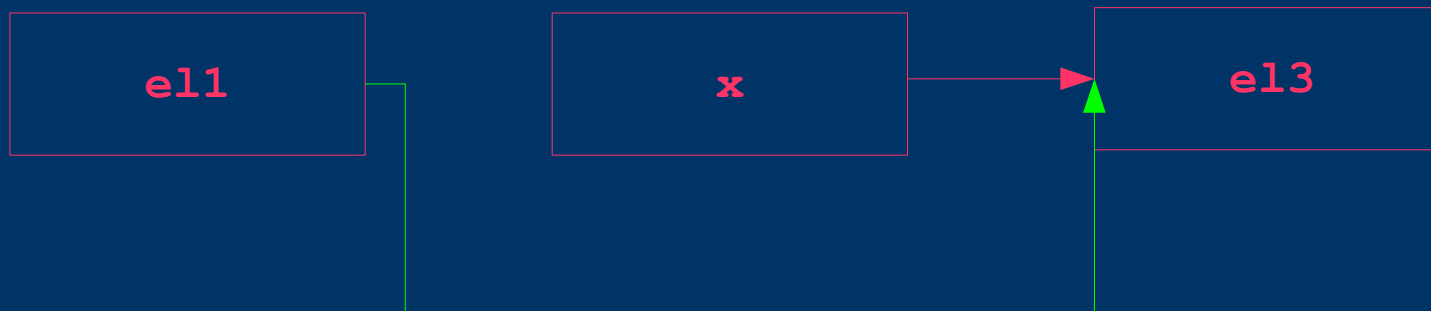
Изтриване в свързан списък₁

- За да изтрием елемент от списъка (в случая **x**) трябва да **знаем** кой е елементът **преди** него (в случая **e11**)!
- Последователността е от значение!



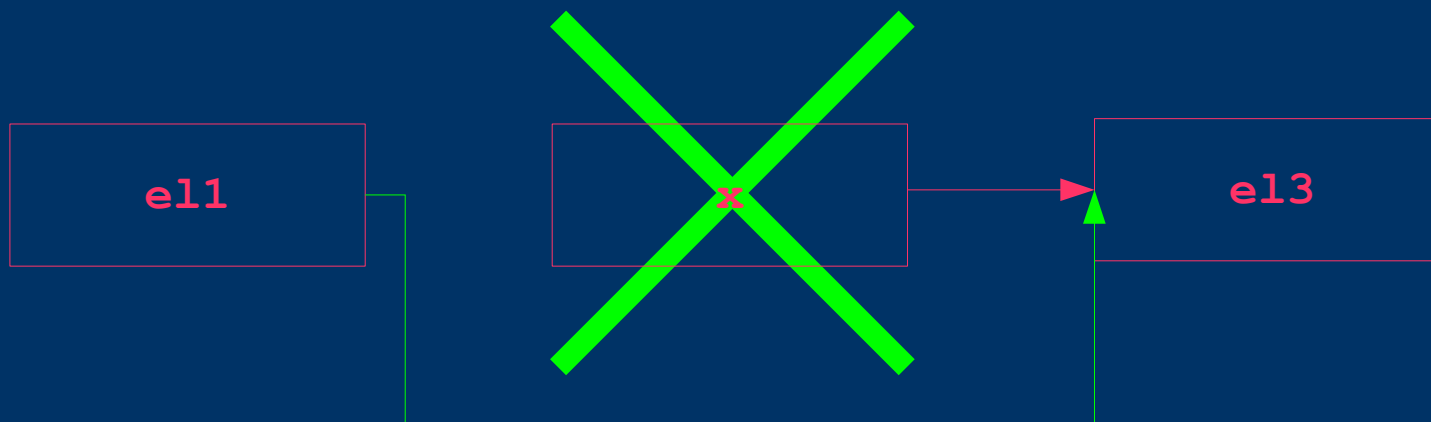
Изтриване в свързан списък₂

- Насочваме `e11->next` към `x->next`



Изтриване в свързан списък₃

- Освобождаваме мястото заделено за **x**

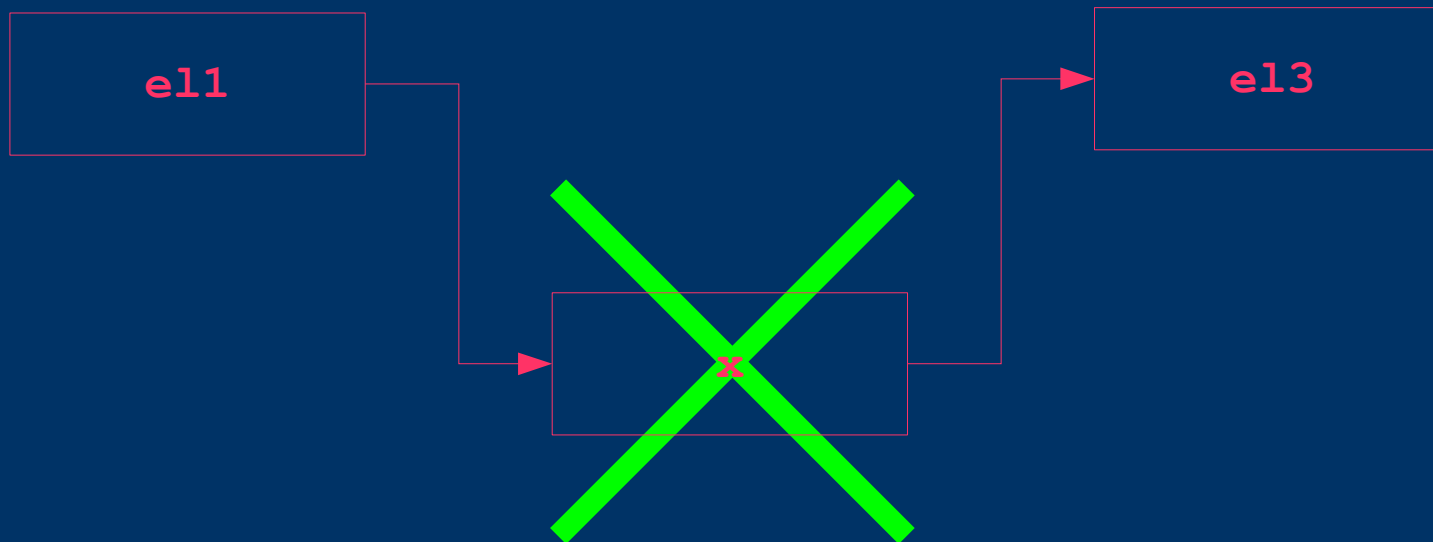


Изтриване в свързан списък₄



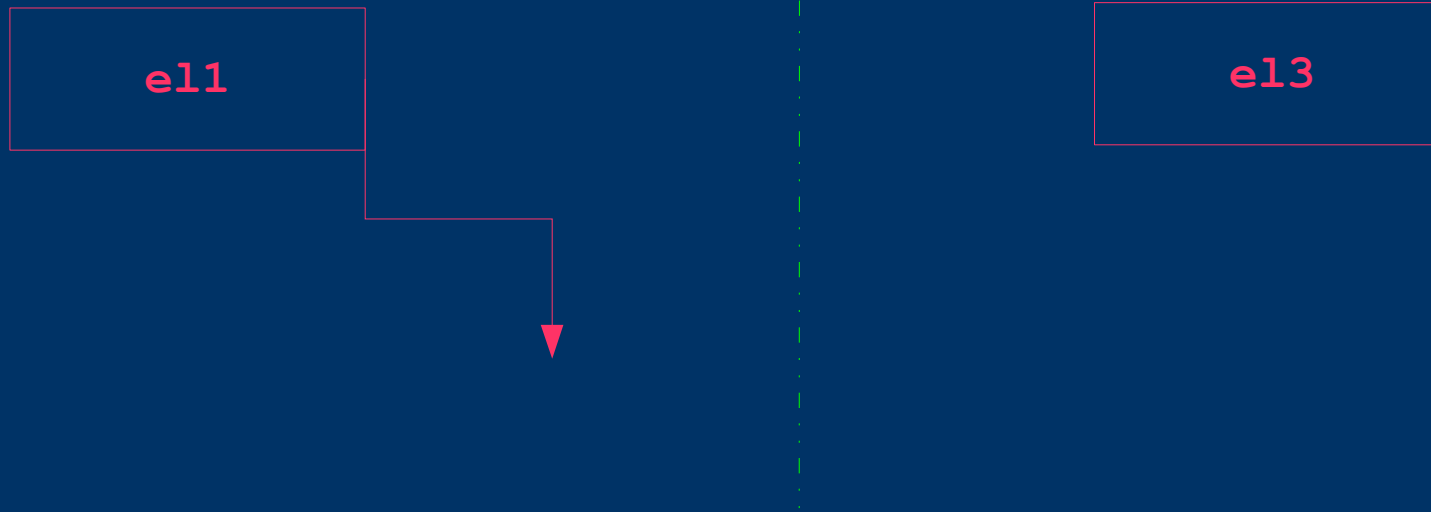
Изтриване в свързан списък₅

- Ако не се спази тази последователност списъкът ще се **разкъса** понеже ще загубим информация за **e13**. Това е често срещана **грешка!**



Изтриване в свързан списък₆

- Ако не се спази тази последователност списъкът ще се **разкъса** понеже ще загубим информация за **e13**. Това е често срещана **грешка!**



Пример

```
#include <stdlib.h>
typedef int Item;//или някакъв друг тип (float, struct, ...)
typedef struct node * Link;//Link е указател към struct node

struct node {
    Item item;//съдържанието на дадения елемент
    Link next;//връзка към следващия елемент
};

/* e11, e12 и e13 са елементи от свързан списък.
Свързани са по следния начин: e11 -> e12 -> e13
e12 (по-късно x) е този, който ще изтрием */
int main () {
    ...

    Link x = e11->next;//x сочи елемента след e11 (т.е. e12)
    e11->next = x->next;//e11 сочи към елемента след x (т.е. e13)
    free(x);//изтриваме динамично заделения елемент e12
}
```

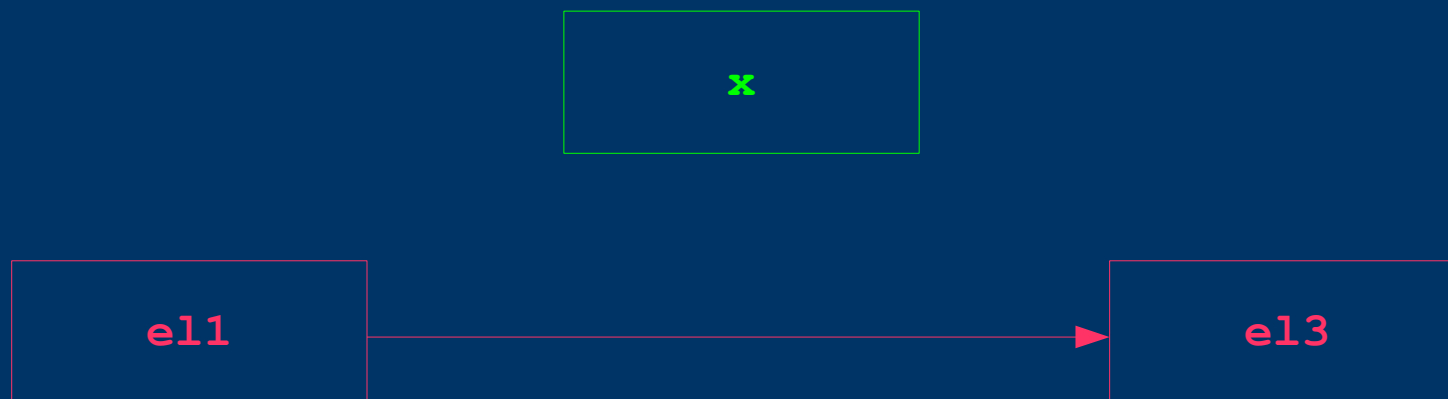
Добавяне в свързан списък₁

- За да добавим елемент в списъка (в случая **x**) трябва да **знаем** кой е елементът преди него (в случая **e11**)! Добавяме **след** него (**e11**)
- Последователността е от значение!



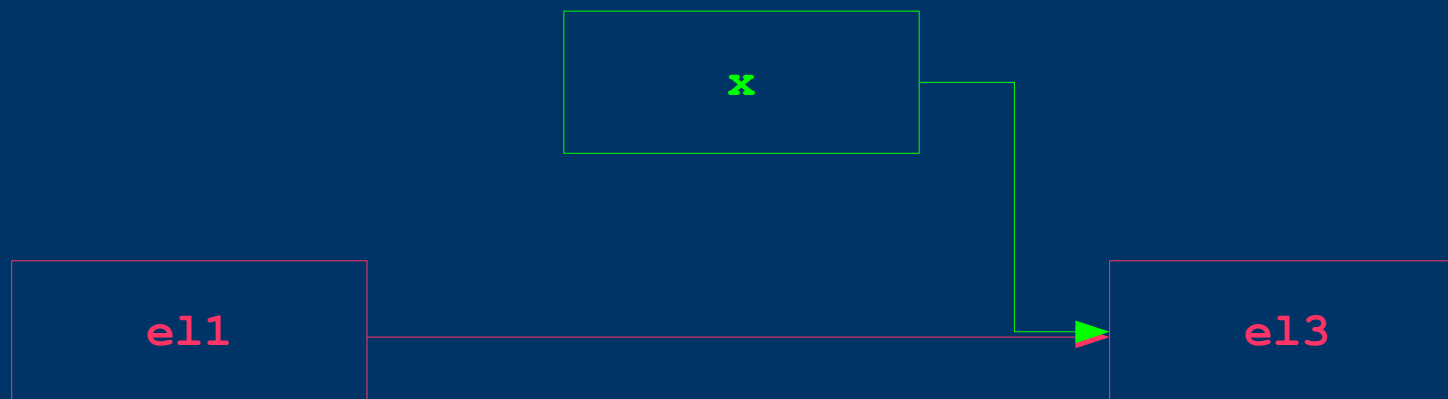
Добавяне в свързан списък₂

- **Заделяме** памет за новия елемент
- **Инициализираме** стойността му



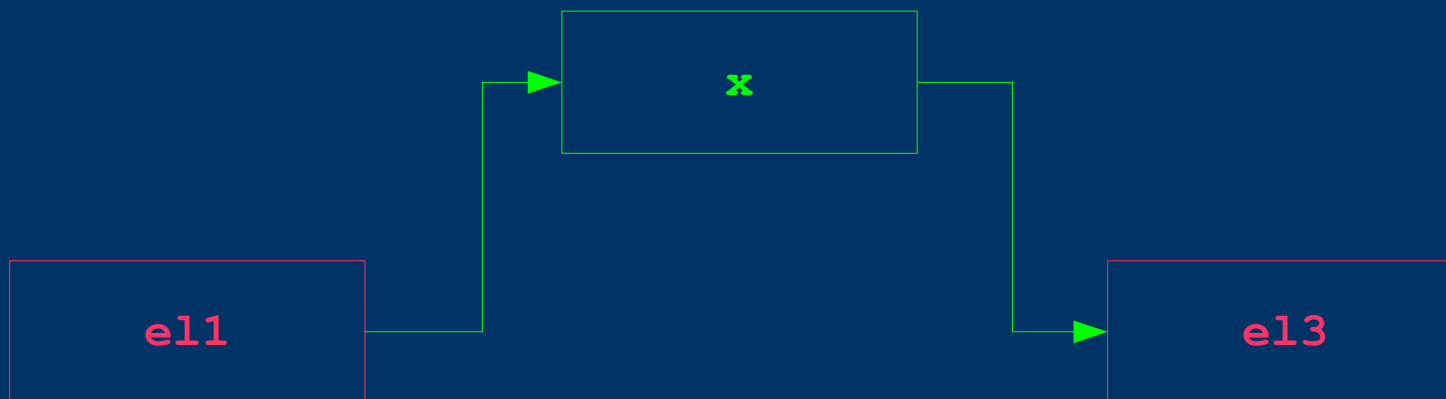
Добавяне в свързан списък₃

- Насочваме `x->next` към `e11->next`



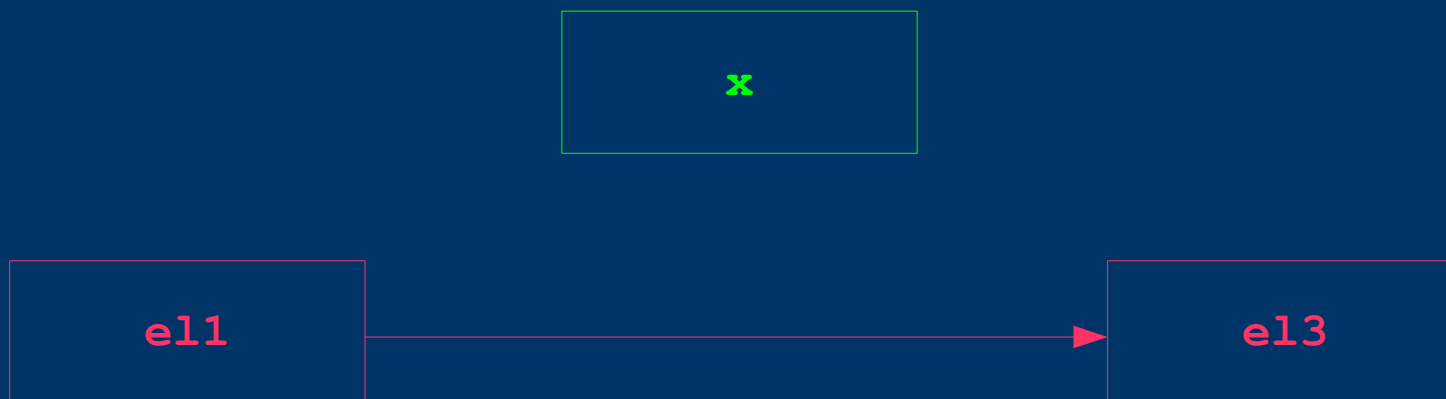
Добавяне в свързан списък₄

- Насочваме `e11->next` към `x`



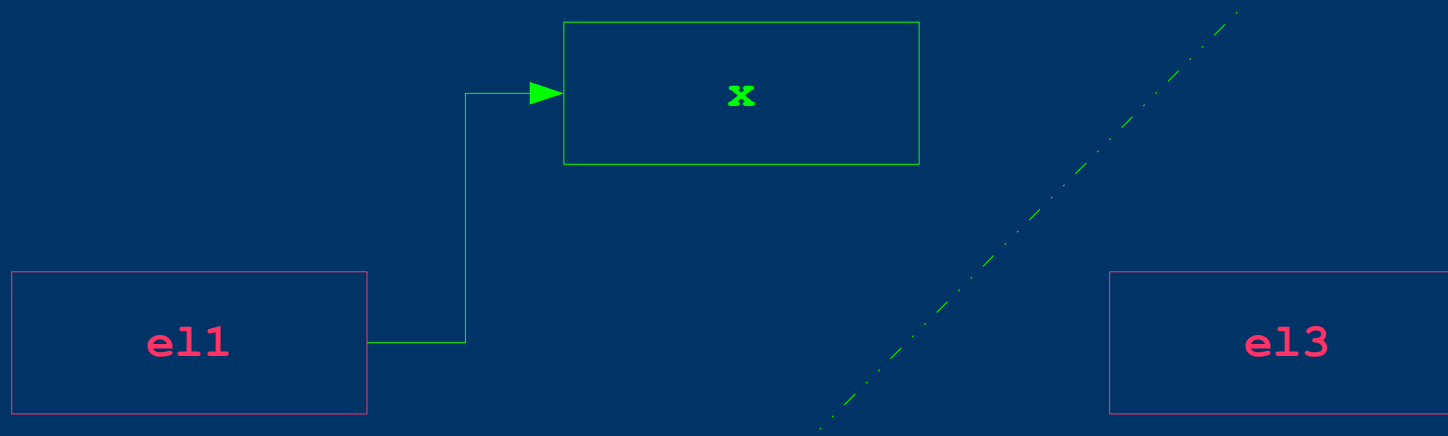
Добавяне в свързан списък₅

- Ако не се спази тази последователност списъкът ще се **разкъса** понеже ще загубим информация за **e13**. Това е често срещана **грешка!**



Добавяне в свързан списък₆

- Ако не се спази тази последователност списъкът ще се **разкъса** понеже ще загубим информация за **e13**. Това е често срещана **грешка!**



Пример

```
#include <stdlib.h>
typedef int Item; //или някакъв друг тип (float, struct, ...)
typedef struct node * Link; //Link е указател към struct node

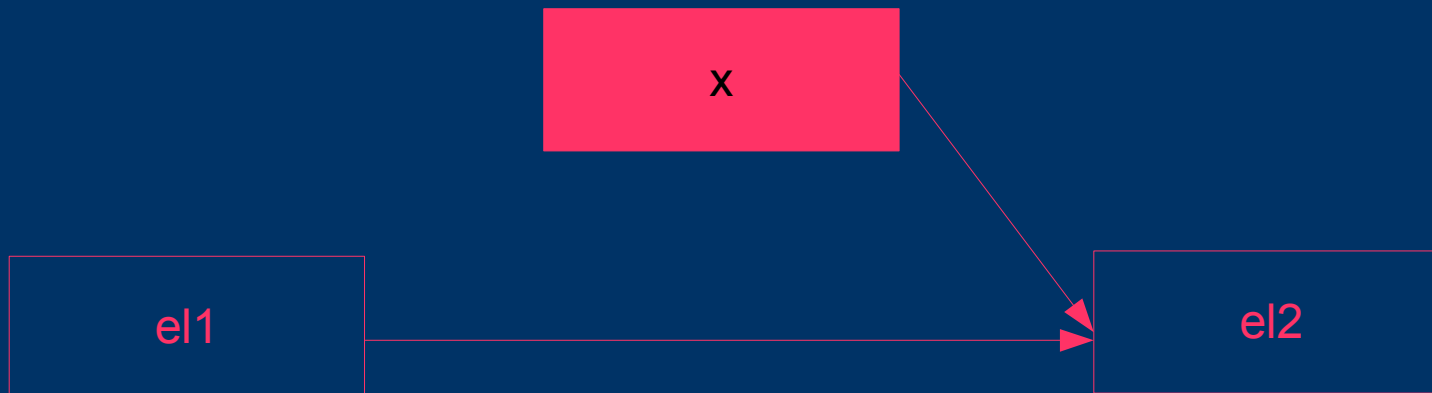
struct node {
    Item item; //съдържанието на дадения елемент
    Link next; //връзка към следващия елемент
};

/* e11 и e13 са елементи от свързан списък.
Свързани са по следния начин: e11 -> e13
x е този, който ще добавим след e11 */
int main () {
    ...

    Link x = (Link) malloc(sizeof(*x)); //заделяме памет за x
    x->item = 2; //инициализираме стойността в елемента x
    x->next = e11->next; //x сочи към елементът, към който сочи e11
    e11->next = x; //e11 сочи към x
}
```

Други често срещани грешки

- Обръщение към ненасочен указател ($e1 \rightarrow next$ да не е инициализиран правилно)
- Неволно да променим указателя **next** на даден елемент, вместо **next** на следващия елемент
- Няколко указателя да сочат към един и същи елемент: $e11 \rightarrow next$ и $x \rightarrow next$ да сочат към $e12$, а никой не сочи към x – пропуснали сме да насочим $e11$ към x



Обхождане на свързан списък

Свързан списък е или нулева връзка (за край на списък), или връзка към възел, който съдържа елемент и връзка към свързан списък

```
//за списъци
for(t = x; t != NULL; t = t->next) {
//докато не се срещне връзка, която сочи към NULL
//указател към NULL - край на списъка
    visit(t->item); //правим нещо с поредния елемент
}

//за масиви:
for(i = 0; i < N; i++) { //докато не обходим и последния ел
    visit(arr[i]); //правим нещо с поредния елемент
}
```



Обръщане на списък



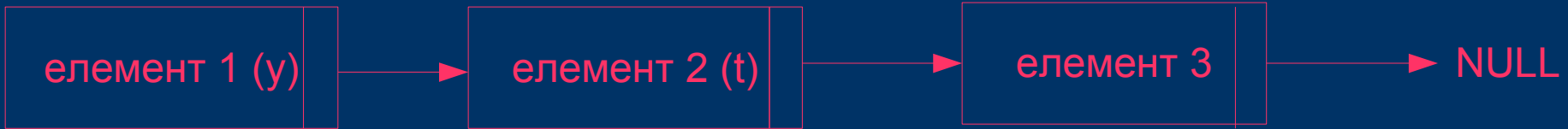
Пример

```
Link reverse(Link x) {
    Link t, y = x, r = NULL;
    while (y != NULL) {
        t = y->next; // t е елементът след y
        y->next = r; // y се насочва към r (предишния елемент)
        r = y; // r става текущият елемент (y)
        y = t; // y става елементът след y (виж 3 реда нагоре t)
    }

    return r;
}
```

NULL (r) - инициализация

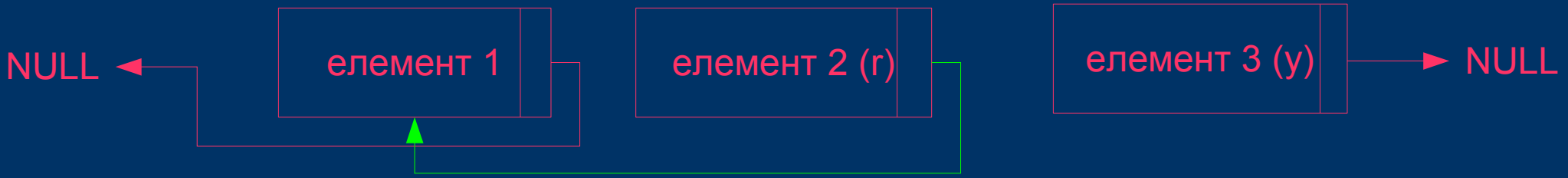
Обръщане на списък



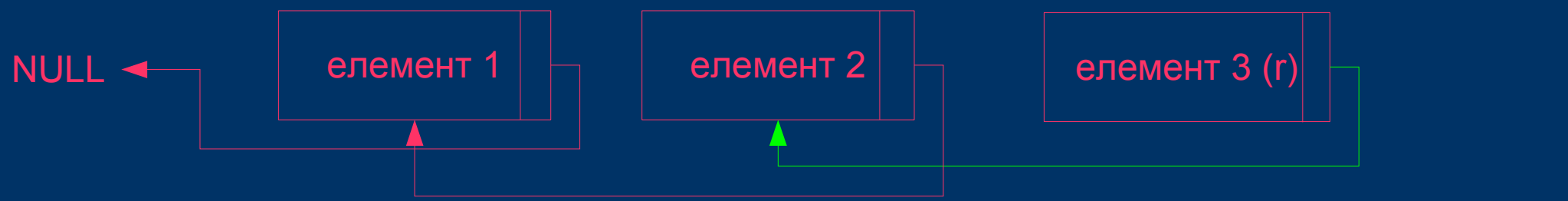
• Итерация 1:



• Итерация 2:



• Итерация 3:



Конвенции за главата и опашката на свързани списъци₁

- Кръгов, никога празен

първо вмъкване

```
head->next = head;
```

вмъкване на t след x

```
t->next = x->next; x->next = t;
```

изтриване след x

```
x->next = x->next->next;
```

обхождащ цикъл

```
t = head;
```

```
do{... t=t->next;}while(t!=head);
```

проверка дали е един елемент

```
if(head->next == head)
```

Конвенции за главата и опашката на свързани списъци₂

- Указател към началото, нулева опашка
инициализиране

```
head = NULL;
```

вмъкване на t след x

```
if(x==NULL) {  
    head = t; head->next = NULL;  
} else {  
    t->next = x->next; x->next = t;  
}
```

изтриване след x

```
t = x->next; x->next = t->next;
```

обхождащ цикъл

```
for(t = head; t!=NULL; t=t->next)
```

проверка дали е празен

```
if(head == NULL)
```

Конвенции за главата и опашката на свързани списъци₃

- Фиктивен водещ възел , нулева опашка

инициализиране

```
head = malloc(sizeof (*head));  
head->next = NULL;
```

вмъкване на t след x

```
t->next = x->next; x->next = t;
```

изтриване след x

```
t = x->next; x->next = t->next;
```

обхождащ цикъл

```
for (t = head->next; t!=NULL; t=t->next)
```

проверка дали е празен

```
if (head->next == NULL)
```

Конвенции за главата и опашката на свързани списъци₄

- Фиктивни водещи и завършващи възли
инициализиране

```
head = malloc(sizeof (*head));  
z = malloc(sizeof (*z));  
head->next = z; z->next = z;
```

вмъкване на t след x

```
t->next = x->next; x->next = t;
```

изтриване след x

```
t = x->next; x->next = t->next;
```

обхождащ цикъл

```
for (t = head->next; t != z; t = t->next)
```

проверка дали е празен

```
if (head->next == z)
```

Използване на водещ възел

- Още един възел в началото (повече памет)
- По-лесно се вмъква в началото на списък (не проверяваме дали има елемент или списъкът е празен)
- Когато искаме да подадем указател към списък като аргумент на функция, която може да променя списъка, в частност да го изтрива (**`head->next = NULL;`**)

Пример

```
void empty(Link x) { //със водещ възел
    x->next = NULL; //списъкът вече е празен
}
void empty(Link x) { //без водещ възел
    x = NULL; //ГРЕШКА - x не се променя в извикващата функция
} //тъй като аргументите се предават по-стойност (копие)

/* без водещ възел не е възможно, тогава се налага да се
ползва друг механизъм - като в примера с reverse
(резултатният списък се връща като резултат от функцията) */
Link reverse(Link x) { //без водещ възел
    Link t, y = x, r = NULL;
    while (y != NULL) {
        t = y->next;
        y->next = r;
        r = y;
        y = t;
    }
    return r;
}
```

Интерфейс за обработка на списъци₁

```
typedef struct node * Link;
struct node {
    ItemType item; // ItemType да е дефинирана някъде другаде...
    Link next;
};

void initNodes (ItemType) ; // инициализиране на списък

Link newNode (ItemType) ; // създаване на нов елемент
void freeNode (Link) ; // освобождаване на паметта за елемент

void insertNext (Link, Link) ; // вмъкване на елемент в списъка
Link deleteNext (Link) ; // изтриване на елемент от списъка

Link next (Link) ; // следващ елемент в списъка
ItemType item (Link) ; // стойност на елемент (Link->item)
```

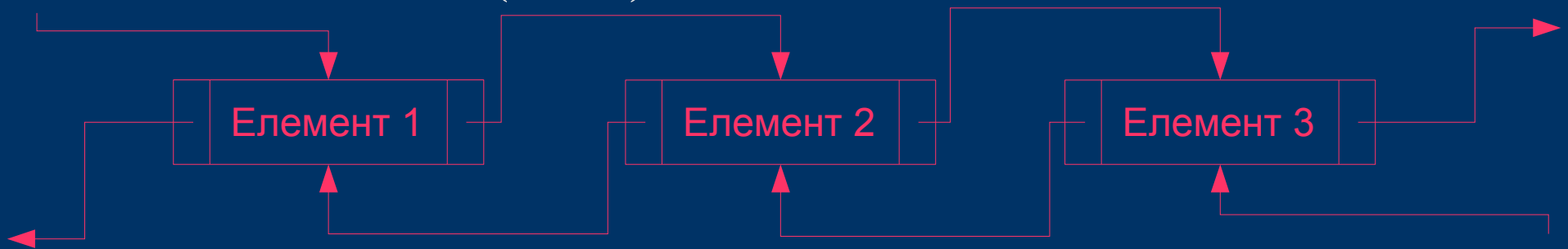
Интерфейс за обработка на списъци₂

Целта е да се направи библиотека, в която се дава възможност за работа със свързани списъци (файлове list.c и list.h).

След това потребителя ще може да я ползва многократно без значение коя точно конвенция за главата и опашката е използвана в библиотеката. По-късно алгоритмите в нея може да се променят с цел оптимизация (но без да се изменят прототипите на функциите от предния слайд) и въпреки това потребителската програма няма да е необходимо да се променя

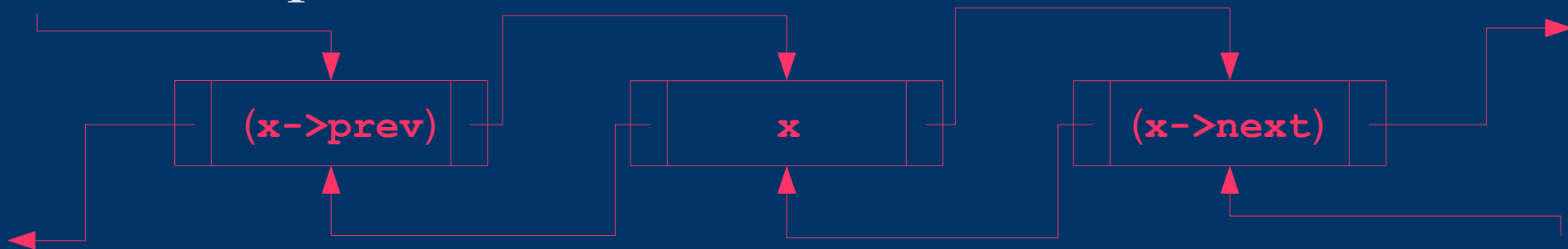
Двусвързани списъци₁

- Чрез добавяне на повече връзки можем да добавим възможността да се **движим обратно** през свързан списък.
- Така можем да поддържаме операция „намери елемент, предходен на даден елемент“.
- Във всеки възел поддържаме **две връзки** – към предшестващия елемент (`prev`) и към следващия (`next`)



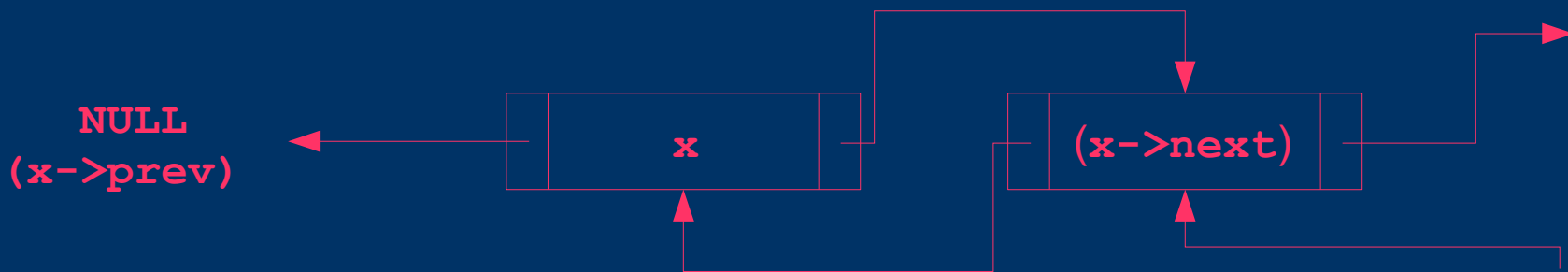
Двусвързани списъци₂

- Двусвързаният списък заема **повече** място в паметта, защото всеки негов елемент има още една връзка (`prev`). Поради това се ползва само когато има реална полза от това (да е двусвързан)
- За разлика от едносвързан списък при изтриване **не се нуждаем** от допълнителна информация за предходния възел (ниतो следващия) – тя се съдържа в самия възел!



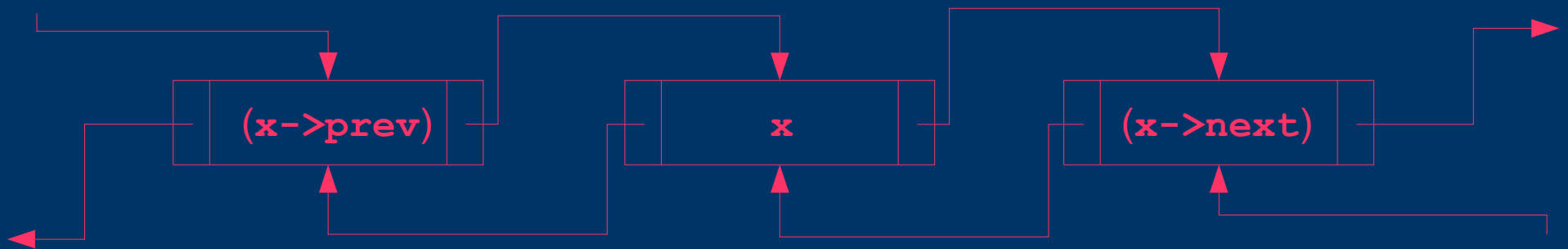
Двусвързани списъци₃

- С фиктивни възли или кръгов списък можем да **гарантираме**, че възлите **x**, **x->next->prev** и **x->prev->next** са едно и също нещо за всеки възел в двусвързан списък. Ако нямаше фиктивен възел в началото, за първия елемент имаме: **x->prev = NULL**, **x->prev->next** е грешно (**NULL->next** е грешно!)



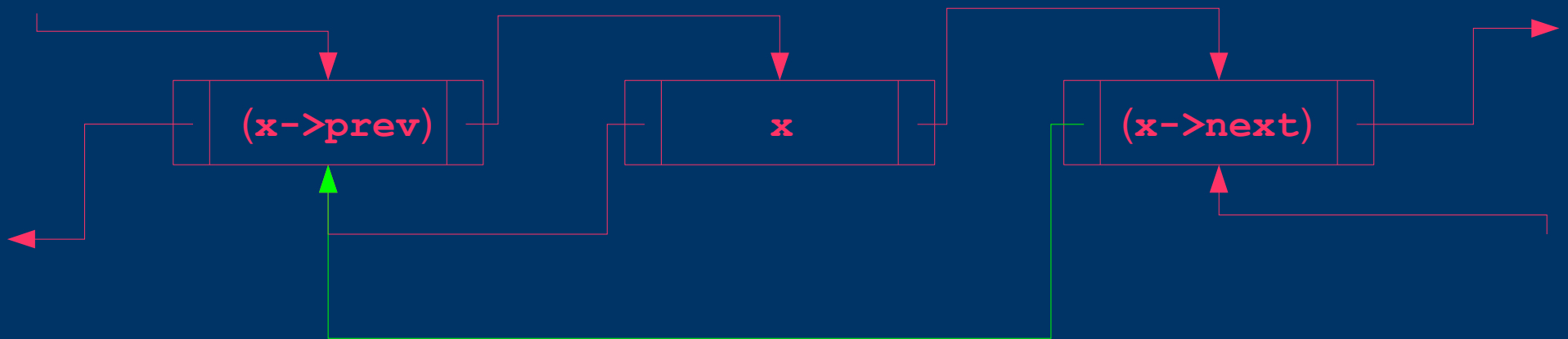
Изтриване в двусвързан списък₁

- Указателят към възела е **достатъчна** информация за да можем да го изтрием от списъка (за **разлика** от едносвързан списък)!
- Последователността е от значение!



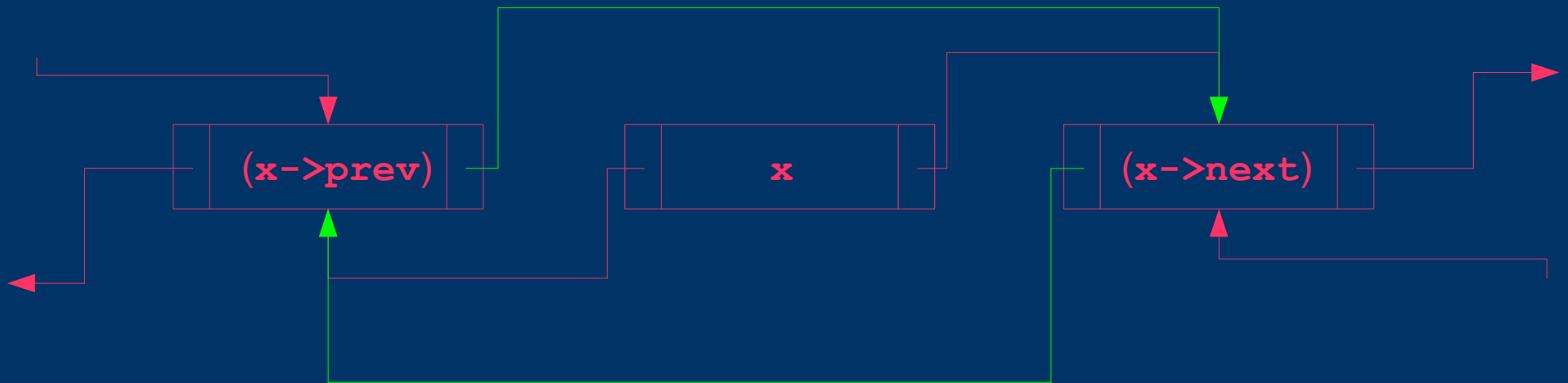
Изтриване в двусвързан списък₂

- Насочваме `x->next->prev` към `x->prev`



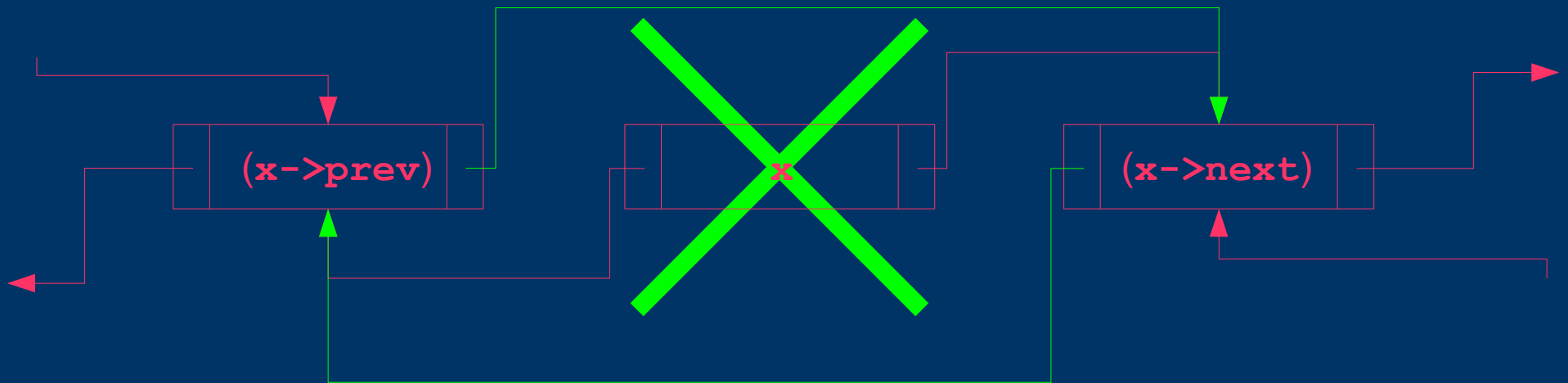
Изтриване в двусвързан списък₃

- Насочваме `x->prev->next` към `x->next`



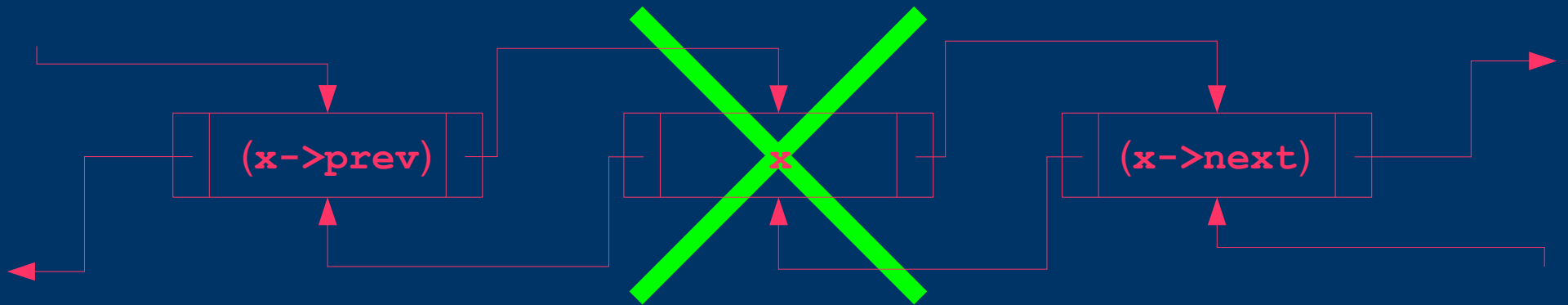
Изтриване в двусвързан списък₄

- Освобождаваме мястото заделено за **x**



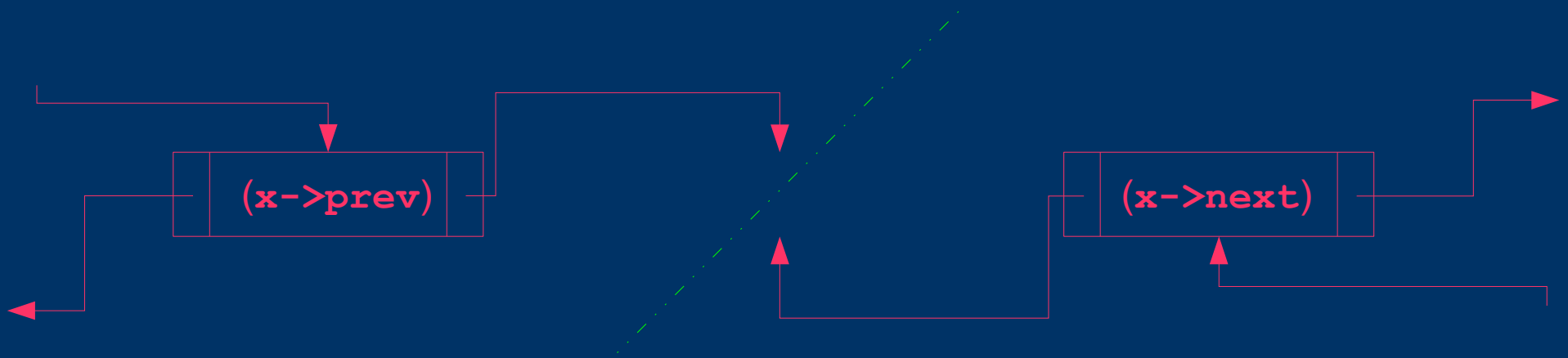
Изтриване в двусвързан списък₆

- Ако не се спази тази последователност списъкът ще се **разкъса** понеже ще загубим информация за следващия елемент. Това е често срещана **грешка!**



Изтриване в двусвързан списък₇

- Ако не се спази тази последователност списъкът ще се **разкъса** понеже ще загубим информация за следващия елемент. Това е често срещана **грешка!**



Пример

```
#include <stdlib.h>
typedef int Item; //или някакъв друг тип (float, struct, ...)
typedef struct node * Link; //Link е указател към struct node

struct node {
    Link prev; //връзка към предишния елемент
    Item item; //съдържанието на дадения елемент
    Link next; //връзка към следващия елемент
};

/* e11, x и e13 са елементи от свързан списък.
Свързани са по следния начин: e11 -> x -> e13
x е този, който ще изтрием */
int main () {
    ...
    x->next->prev = x->prev; //казваме на елементът след x (x->next), че
                           //преди него е елементът преди x (x->prev)
    x->prev->next = x->next; //казваме на елементът преди x (x->prev), че
                           //след него е елементът след x (x->next)
    free(x); //изтриваме динамично заделената памет
}
```

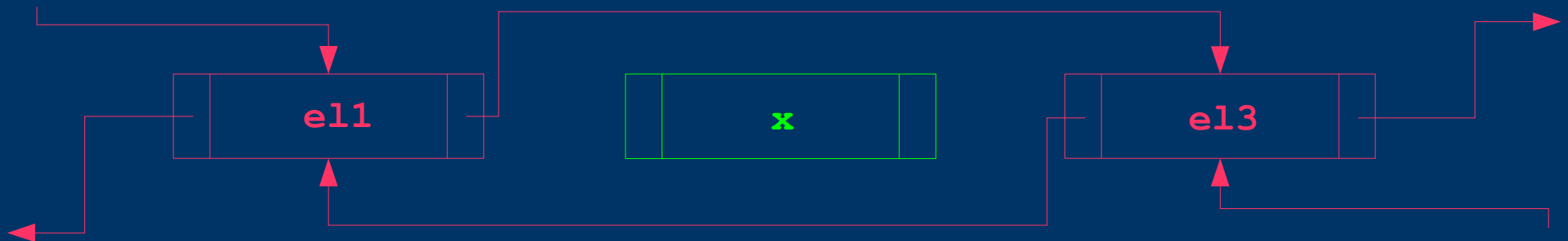
Добавяне в двусвързан списък₁

- За да добавим елемент в списъка (в случая **x**) трябва да **знаем** кой е елементът преди него (в случая **e11**)! Добавяме **след** него (**e11**)
- За разлика от едносвързания списък **може** да се добавя и **преди** даден възел
- Последователността е от значение!



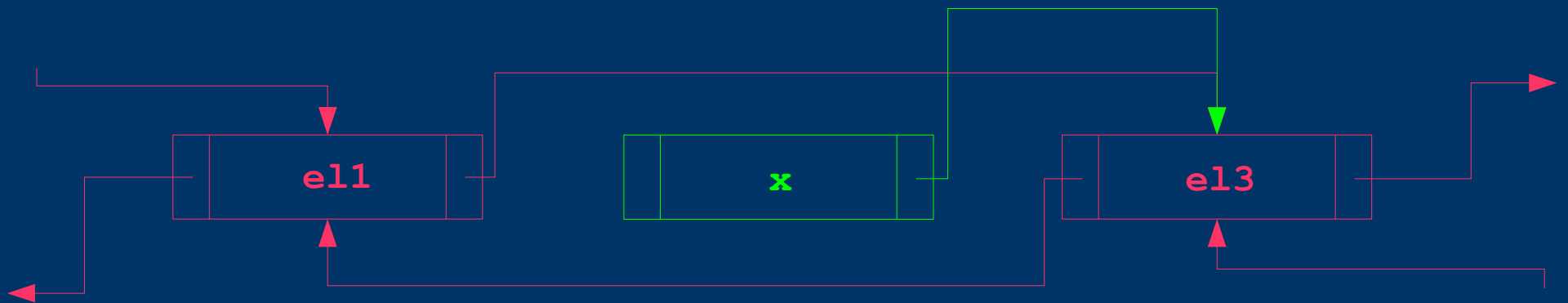
Добавяне в двусвързан списък₂

- **Заделяме** памет за новия елемент
- **Инициализираме** стойността му



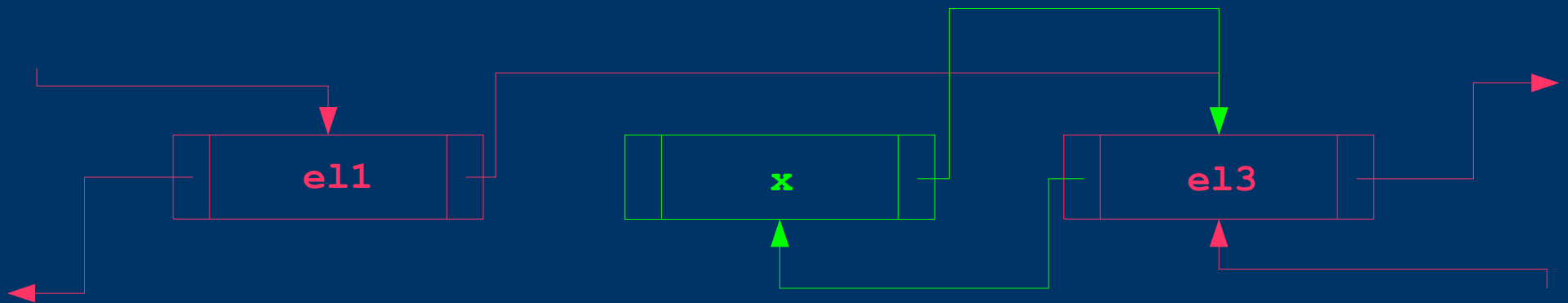
Добавяне в двусвързан списък₃

- Насочваме `x->next` към `e11->next`



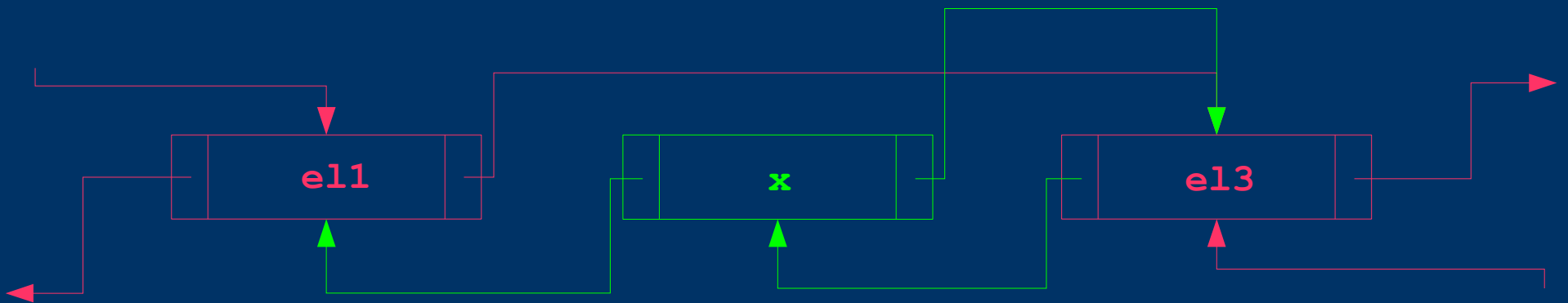
Добавяне в двусвързан списък₄

- Насочваме $x \rightarrow next \rightarrow prev$ към x



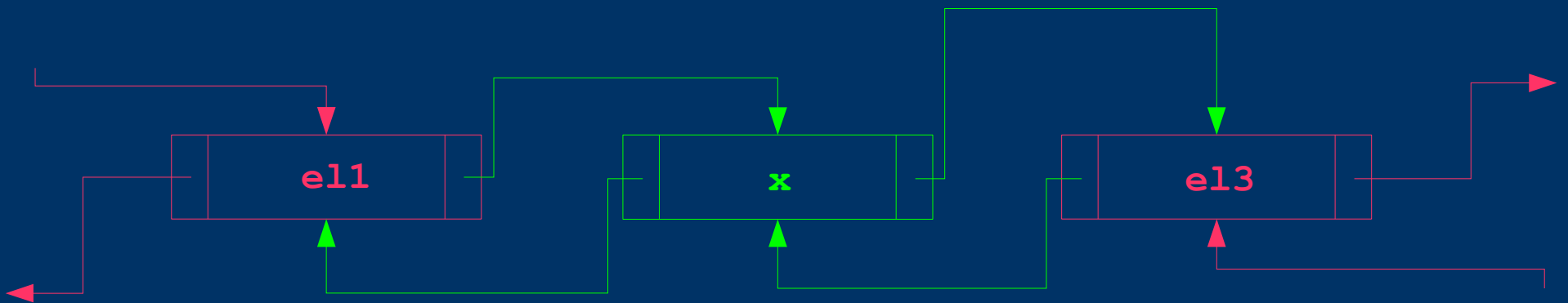
Добавяне в двусвързан списък₅

- Насочваме $x \rightarrow prev$ към $e11$



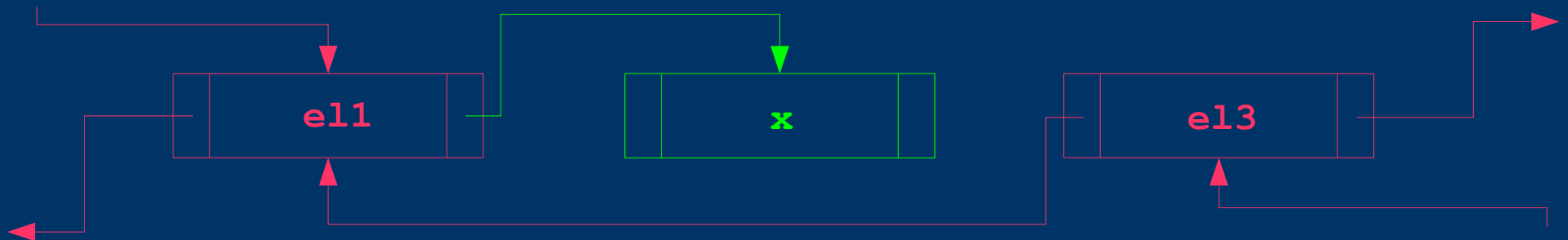
Добавяне в двусвързан списък₆

- Насочваме `e11->next` към `x`



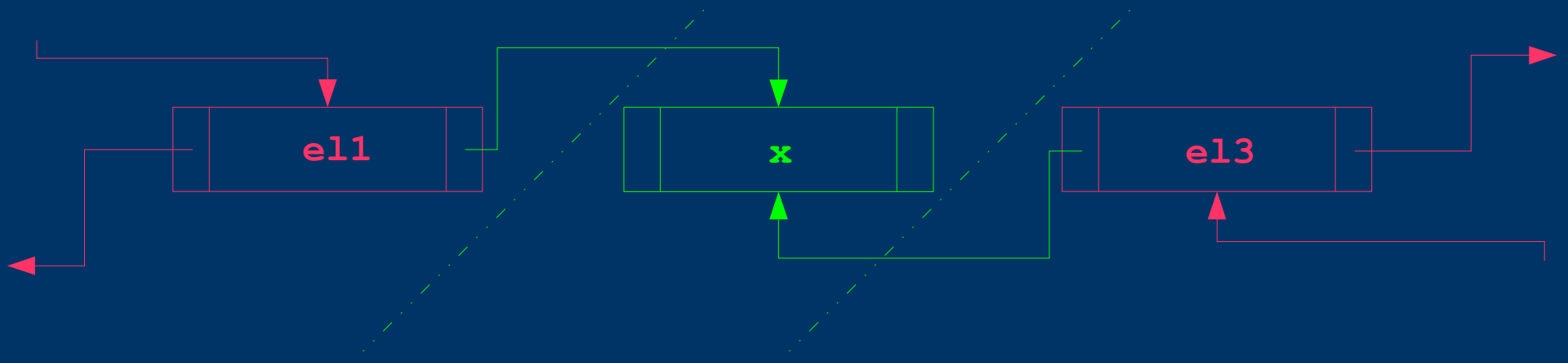
Добавяне в двусвързан списък₇

- Ако не се спази тази последователност списъкът ще се **разкъса** понеже ще загубим информация за **e11** или **e13**. Това е често срещана **грешка!**



Добавяне в двусвързан списък₈

- Ако не се спази тази последователност списъкът ще се **разкъса** понеже ще загубим информация за **e11** или **e13**. Това е често срещана **грешка!**



Пример

```
#include <stdlib.h>
typedef int Item; //или някакъв друг тип (float, struct, ...)
typedef struct node * Link; //Link е указател към struct node
struct node {
    Link prev; //връзка към предишния елемент
    Item item; //съдържанието на дадения елемент
    Link next; //връзка към следващия елемент
};
/* e11 и e13 са елементи от свързан списък. (e11 -> e13)
х е този, който ще добавим след e11 */
int main () {
    ...
    Link x = (Link) malloc(sizeof(*x)); //заделяме памет за х
    x->item = 6; //инициализираме стойността в елемента х

    x->next = e11->next; //х сочи на където сочи и e11
    x->next->prev = x; //e13 научава, че преди него е х
    x->prev = e11; //преди х е e11
    e11->next = x; //e11 започва да сочи към х
}
```