

Wstęp do informatyki

Wykład 10

Abstrakcyjne typy danych, listy wiązane

Instytut Informatyki UW

Temat wykładu

- Abstrakcyjne typy danych
- Listy (liniowe/wiązane), w tym uporządkowane – implementacja
- Stos, kolejka – przykłady abstrakcyjnych struktur danych, implementowanych za pomocą list wiązanych

Abstrakcyjny typ danych

Co to jest „abstrakcyjny typ danych”?

- Zadany przez
 - zbiór dopuszczalnych wartości i jego własności,
 - zestaw dopuszczalnych operacji.
- Definicja typu **nie** opisuje sposobu jego implementacji

Po co abstrakcyjne typy danych?

- **Modularyzacja** programów (implementacja operacji na typie danych „ukryta” w funkcjach/procedurach/metodach, „bibliotekach”)
- Prostszy zapis programów
- Możliwość dodawania nowych operacji

Porównaj z: programowanie strukturalne, obiektowe.

Przykład abstr. typu danych – **zbiór**

Opis typu definiującego zbiory liczb całkowitych:

- **Zbiór wartości** – podzbiory zbioru liczb całkowitych (z określonego zakresu $[0, n]$)
- **Operacje:**
 - Utwórz zbiór pusty
 - Sprawdź czy element x jest w zbiorze S
 - Dodaj x do zbioru S
 - Usuń x ze zbioru S
 - Operacje teoriomnogościowe: suma zbiorów, różnica zbiorów, przecięcie zbiorów, zawieranie zbiorów

Przykład abstr. typu danych – zbiór

Implementacja (przykładowa):

- Reprezentacja zbioru: tablica zer i jedynek a taka, że $a[x] == 1$ gdy x należy do zbioru (tzw. wektor charakterystyczny)
- Operacje:
 - Utwórz zbiór pusty: wypełnij tablicę zerami
 - Sprawdź czy element x jest w zbiorze: $a[x] == 1$?
 - Dodaj x : $a[x] = 1$
 - Usuń x z S : $a[x] = 0$
 - Suma zbiorów a i b zapisana w zbiorze c :
Dla $i=0, 1, \dots, n$:
jeśli $a[i] == 1$ lub $b[i] == 1$: $c[i] = 1$
wpp $c[i] = 0$
 - Inne operacje – zad. domowe...

Abstrakcyjny typ danych – cele implementacji

Cele:

- **sposób wyboru definicji typu i operacji:**
zagwarantować szeroki zakres zastosowań;
- **cele implementacji:**
 - mała pamięć?
 - mały czas realizacji operacji? ... *których?*

Przykład – podzbiory zbioru $\{0, 1, \dots, n\}$

Implementacja (przykładowa – wektor charakterystyczny):

- Pamięć – każdy zbiór wymaga n komórek (**dużo!**)
- Czas:
 - utwórz zbiór pusty: $O(n)$,
 - Sprawdź czy element x jest w zbiorze: $O(1)$
 - Dodaj x : $O(1)$
 - Usuń x ze zbioru: $O(1)$
 - suma zbiorów a i b w zbiorze c : $O(n)$ **można szybciej?**

Typ abstrakcyjny – lista liniowa („wiązana”)

Typ abstrakcyjny – lista liniowa („wiązana”)

Opis typu definiującego listę liniową złożoną z elementów typu T:

- zbiór wartości – **ciągi** elementów typu T
- Operacje na liście liniowej L:
 - Utwórz listę z jednym elementem o wartości x (utworz(x))
 - Pobierz pierwszy element (pierwszy(L)),
 - Zwróć listę bez pierwszego elementu (ogon(L)),
 - Dodaj x na początek (wstawPocz(x,L))
 - Usuń element o wartości x (usun(x,L))
 - Wypisz zawartość listy (wypisz(L))

Typ abstrakcyjny – lista liniowa („wiązana”)

Lista L – przykład:

Operacje na liście liniowej L:

- | | |
|-----------------------------------|-----------------------|
| – $L \leftarrow \text{utworz}(5)$ | 5 |
| – $\text{wstawPocz}(17, L)$ | 17, 5 |
| – $\text{wstawPocz}(2, L)$ | 2, 17, 5 |
| – $\text{ogon}(L),$ | 17, 5 |
| – $\text{wstawPocz}(3, L)$ | 3, 17, 5 |
| – $\text{pierwszy}(L)$ | 3 [lista bez zmian] |
| – $\text{usun}(17, L)$ | 3, 5 |
| – $\text{wypisz}(L)$ | |

Typ abstrakcyjny – lista liniowa („wiązana”)

Listy w Python a lista wiązana....:

- Listy dostępne w Pythonie zapewniają realizację operacji dla list wiązanych, a nawet **więcej**....
- ale **nie** są zoptymalizowane pod kątem wykonywania operacji wstawiania i usuwania elementów z początku/końcu listy (o czym się przekonamy...)
- ... dlatego poznamy inny sposób implementacji

Typ abstrakcyjny – **uporządkowana** lista liniowa

Opis typu definiującego uporządkowaną listę liniową złożoną z elementów typu porządkowego T (np. int):

- zbiór wartości – **niemalejące** ciągi elementów typu T
- Operacje na liście liniowej L:
 - Utwórz listę z elementem o wartości x (utworz(x))
 - Pobierz pierwszy element (pierwszy(L)),
 - Zwróć listę bez pierwszego elementu (ogon(L)),
 - Dodaj x (wstawPorz(x,L)) do listy L (**z zachowaniem porządku**)
 - Usuń element o wartości x (usun(x,L))
 - Wypisz zawartość listy (wypisz(L))

Lista liniowa – implementacja tablicowa

Reprezentacja listy L złożonej z elementów typu T:

- tablica L
- zmienna n określająca aktualny rozmiar listy
- $L[0], \dots, L[n - 1]$ to elementy listy

Wady:

- Rozmiar tablicy ograniczeniem na długość listy,
- Rozmiaru tablicy nie można łatwo zmieniać (?) więc zajmowana pamięć jest proporcjonalna do najdłuższej dopuszczalnej listy
- Usuwanie elementu „ze środka” kosztowne, podobnie dodawanie („przesuwanie” elementów)

Lista liniowa i wskaźniki w C

Lista – struktura z wyróżnionym polem oznaczającym **wskaźnik na następny element listy**.

Przykład

```
struct elem {  
    int val;        //dane właściwe  
    struct elem *next; // wskaźnik na następny element  
};
```

Lista liniowa i wskaźniki w Python

Lista - klasa z wyróżnionym polem oznaczającym **wskaźnik na następny element** listy (p. pakiet **wdi** na stronie przedmiotu na skos!).

```
class ListItem:  
    def __init__(self, value) :  
        self.val = value #dane właściwe  
        self.next = None #następny element
```

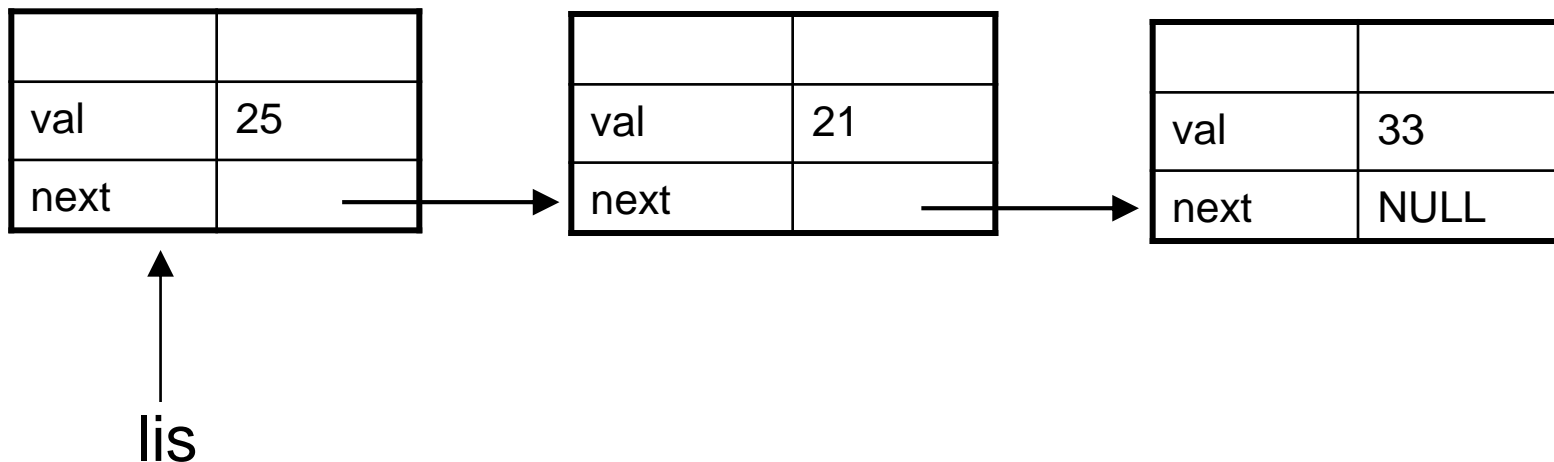
Lista liniowa i wskaźniki

Dostęp do listy – wskaźnik na jej pierwszy element:

- **struct** elem *lis;

UWAGA: zmienna **lis** ma taki sam **typ** jak pole **next** w strukturze struct elem.

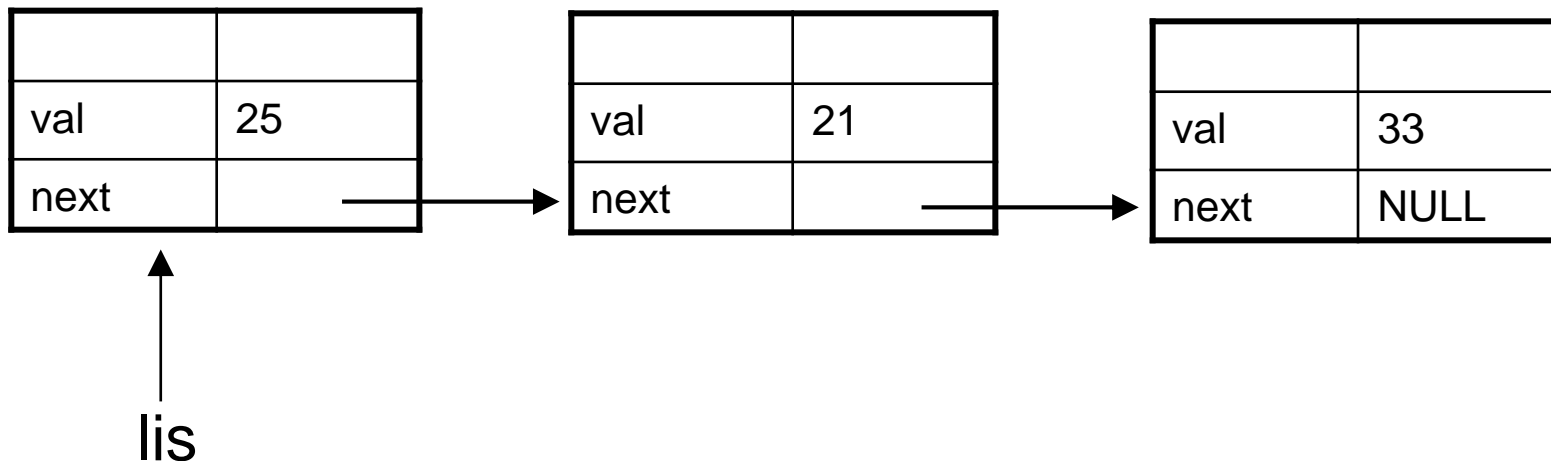
Przykład



Lista – wypisz elementy listy

```
void wypisz(struct elem *lis)
{
    while (lis!=NULL) {
        printf("%d\n",lis->val);
        lis = lis->next;
    }
}
```

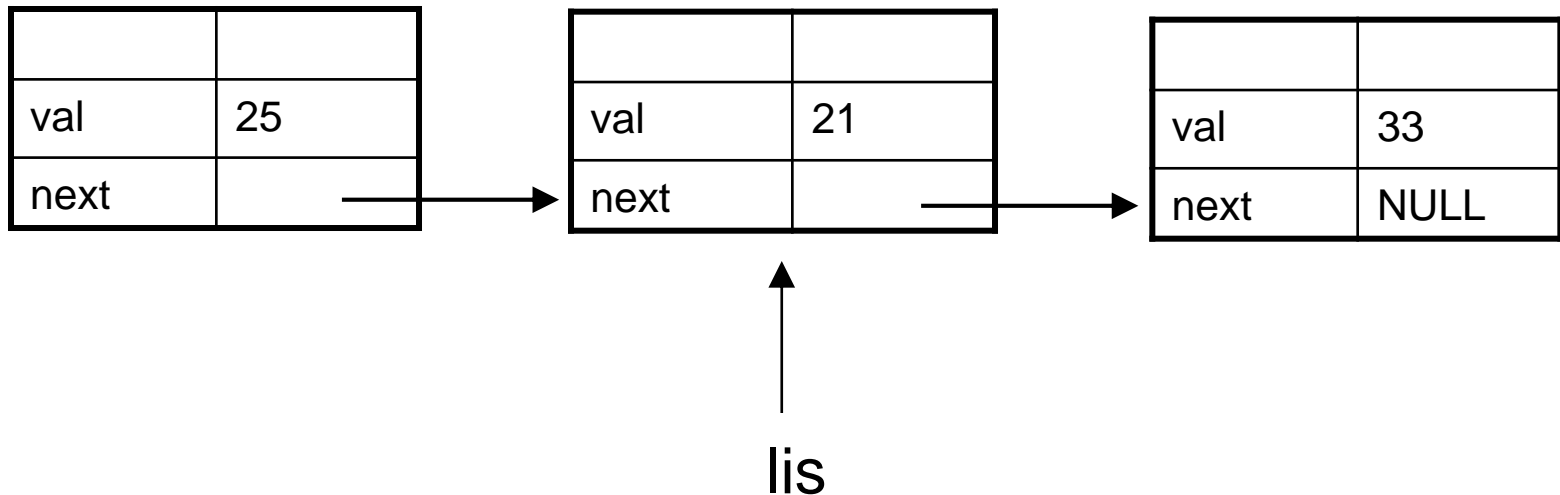
```
def wypisz(lis):
    while lis!=None:
        print lis.val
        lis = lis.next
```



Lista – wypisz elementy listy

```
void wypisz(struct elem *lis)
{
    while (lis!=NULL) {
        printf("%d\n",lis->val);
        lis = lis->next;
    }
}
```

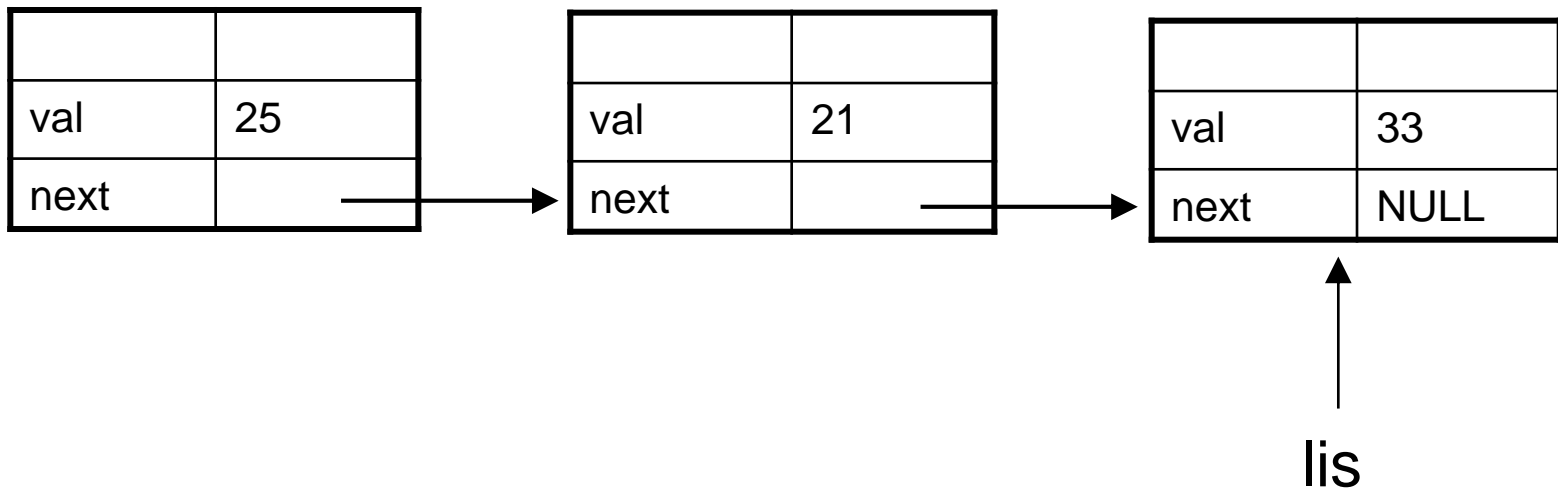
```
def wypisz(lis):
    while lis!=None:
        print lis.val
        lis = lis.next
```



Lista – wypisz elementy listy

```
void wypisz(struct elem *lis)
{
    while (lis!=NULL) {
        printf("%d\n",lis->val);
        lis = lis->next;
    }
}
```

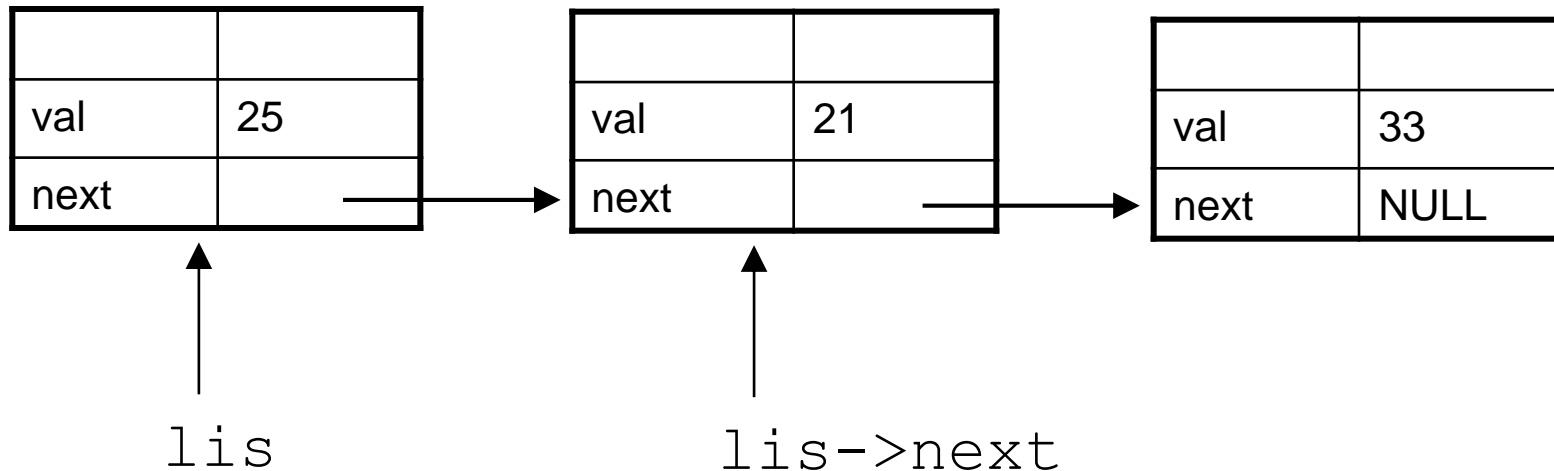
```
def wypisz(lis):
    while lis!=None:
        print lis.val
        lis = lis.next
```



Lista – bez pierwszego

```
struct elem *ogon(struct elem *lis)
{
    if (lis!=NULL) return lis->next;
    else return NULL;
}
```

```
def ogon(lis):
    if lis!=None:
        return lis.next
    else: return None
```



Lista / wskaźniki: tworzenie

Dotychczas:

- wykonywaliśmy operacje na istniejącej liście

Tworzenie listy:

- Każdy element tworzony osobno („rezerwujemy” dla niego pamięć)
- Deklaracja zmiennej wskaźnikowej w języku C **nie** powoduje utworzenia elementu, na który ta zmienna wskazuje!
- Łączenie elementów w listę możliwe poprzez pola next

Lista / wskaźniki: tworzenie w C

Zadeklarowanie zmiennej

```
struct elem *lis;
```

nie powoduje utworzenia struktury typu **struct** elem a jedynie wskaźnik na **struct** elem.

Utworzenie nowej struktury wymaga zaalokowania pamięci (na sterckie):

```
lis = (struct elem *) malloc(sizeof(struct elem));
```

Lista / wskaźniki: tworzenie w C

Utworzenie struktury wymaga zaalokowania pamięci:

```
lis = (struct elem *) malloc(sizeof(struct elem));
```

gdzie:

- `malloc(i)` rezerwuje pamięć o rozmiarze `i` „jednostek pamięci”
- `sizeof(TYP)` podaje liczbę jednostek pamięci zajmowanych przez elementy typu `TYP`
- `(struct elem *)` to rzutowanie typu, które powoduje, że wynik funkcji `malloc(...)` traktowany będzie jako wskaźnik na element typu `struct elem`.

Lista / wskaźniki: tworzenie Python

Utworzenie nowego elementu:

```
lis = ListItem( <WARTOSC> )
```

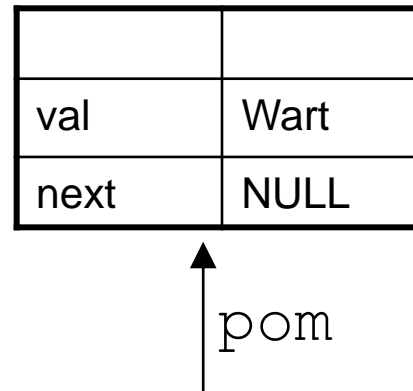
tworzy element, w którym pole **val** jest równe **<WARTOSC>** a wskaźnik na następny element jest pusty (None).

```
class ListItem:  
    def __init__(self,value):  
        self.val = value  
        self.next = None
```


Lista / wskaźniki: tworzenie

```
struct elem *utworz(int wart)
{ //utworzenie nowej jednoelementowej listy
  struct elem *pom;

  pom=(struct elem *) malloc(sizeof(struct elem));
  pom->val = wart;
  pom->next = NULL;
  return pom;
}
```



Python: `lis = ListItem(wart)`

C: `lis = utworz(wart)`

Lista : nowy element na początek

```
struct elem *wstawPocz(struct elem *lista, int nval)
{ // val dodajemy na poczatek listy lista
    struct elem *nowy;
    nowy=utworz(nval);
    nowy->next = lista;
    return nowy;
}
```

```
def wstawPocz(lista, nval):
    # val dodajemy na poczatek listy
    nowy=ListItem(nval)
    nowy.next = lista
    return nowy
```

val	nval
next	NULL

↑
nowy

val	21
next	→

↑
lista

val	33
next	NULL

Lista : nowy element na początek

```
struct elem *wstawPocz(struct elem *lista, int nval)
```

```
{ // val dodajemy na poczatek listy lista
```

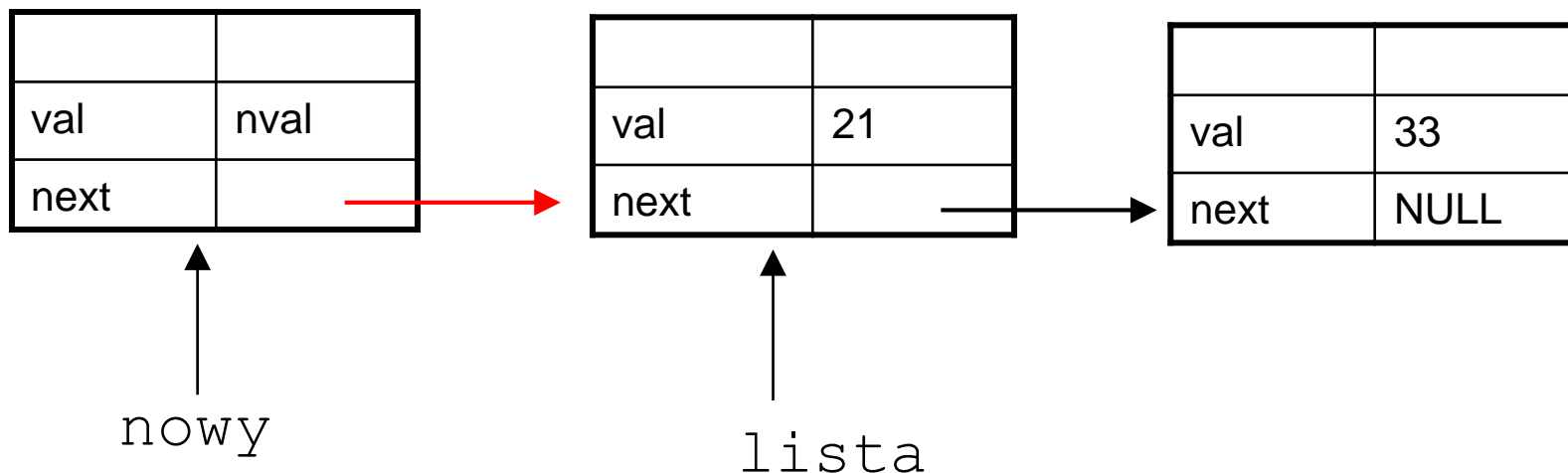
```
    struct elem *nowy;
```

```
    nowy=utworz(nval);
```

```
    nowy->next = lista;
```

```
    return nowy;
```

```
def wstawPocz(lista, nval):  
    # val dodajemy na poczatek listy  
    nowy=ListItem(nval)  
    nowy.next = lista  
    return nowy
```



Lista : znajdź sval

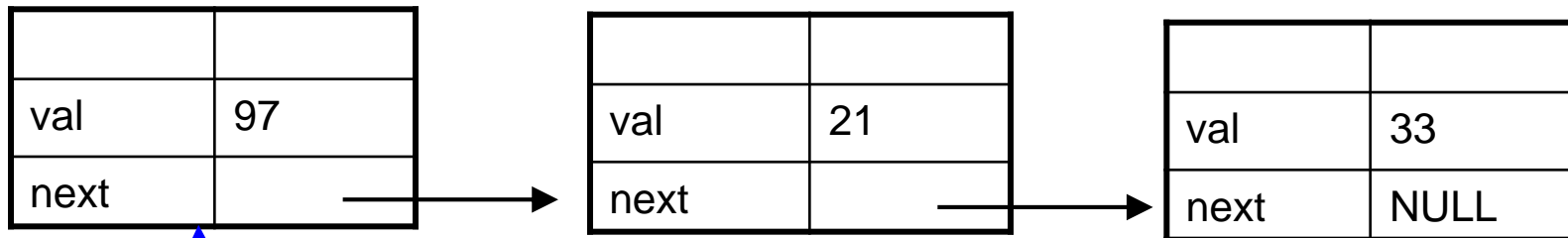
```
struct elem *znajdz(struct elem *lista, int sval)
{ // zwraca wskaźnik na element zawierający sval
  // zwraca NULL gdy brak takiego elementu
  while (lista!=NULL) {
    if (lista->val==sval) return lista;
    lista = lista->next;
  }
  return NULL;
}
```

```
def znajdz(lista, sval):
  # zwraca wskaźnik na el. zawierający sval
  # zwraca None gdy brak takiego elementu
  while lista!=None:
    if (lista.val==sval): return lista
    lista = lista.next
  return None
```

Lista : znajdź sval

```
struct elem *znajdz(struct elem *lista, int sval)
{
    // zwraca wskaźnik na element zawierający sval
    // zwraca NULL wpp
    while (lista!=NULL) {
        if (lista->val==sval)
            return lista;
        lista = lista->next;
    }
    return NULL;
}
```

```
def znajdz(lista, sval):
    # zwraca wskaźnik na el. zawierający sval
    # zwraca None gdy brak takiego elementu
    while lista!=None:
        if (lista.val==sval): return lista
        lista = lista.next
    return None
```



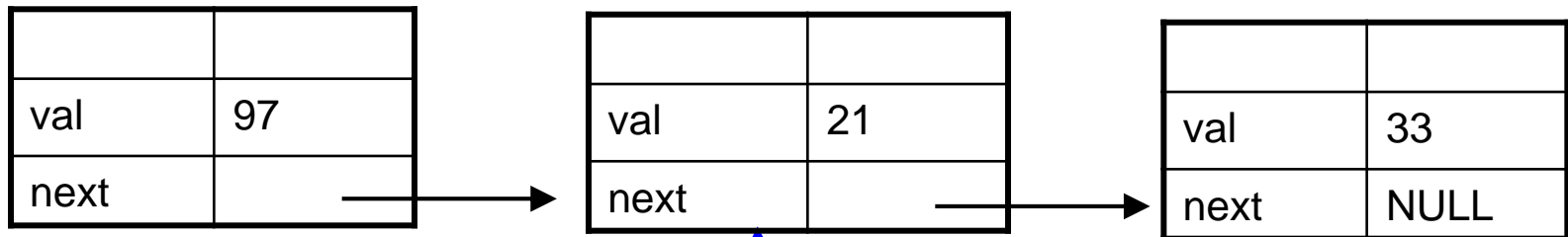
lista

znajdz(lista, 21)

Lista : znajdź sval

```
struct elem *znajdz(struct elem *lista, int sval)
{
    // zwraca wskaźnik na element zawierający sval
    // zwraca NULL wpp
    while (lista!=NULL) {
        if (lista->val==sval)
            return lista;
        lista = lista->next;
    }
    return NULL;
}
```

```
def znajdz(lista, sval):
    # zwraca wskaźnik na el. zawierający sval
    # zwraca None gdy brak takiego elementu
    while lista!=None:
        if (lista.val==sval): return lista
        lista = lista.next
    return None
```



lista

znajdz(lista, 21)

Lista uporządkowana

Lista uporządkowana: nowy element

Problem (opisuje `wstawPorz(lis, nval)`)

Wejście:

`lis` – wskaźnik na listę z kluczami uporządkowanymi niemalejąco

`nval` – nowy klucz do wstawienia

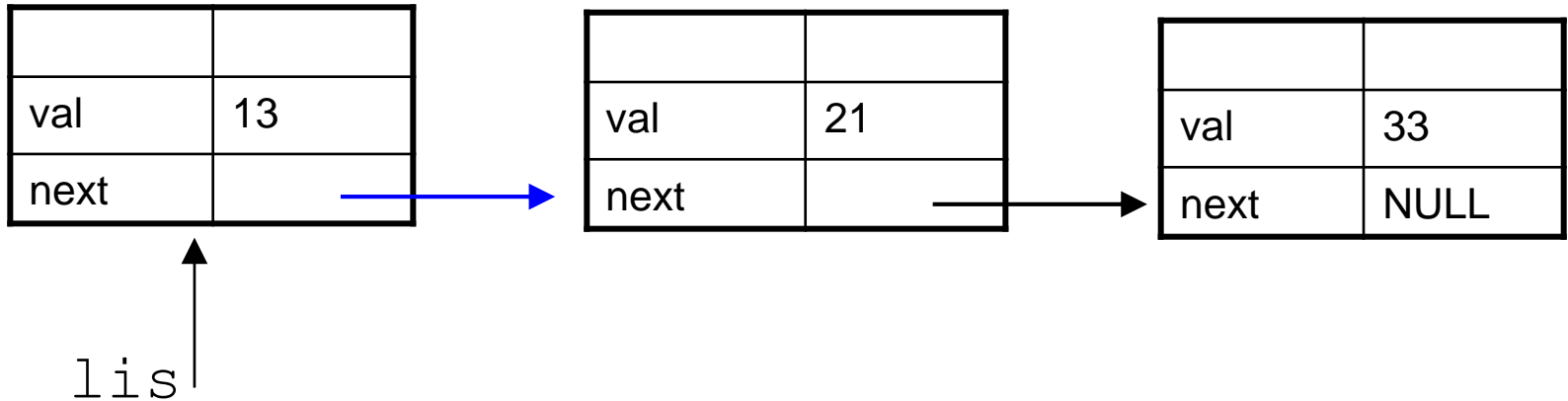
Wyjście:

lista otrzymana przez dodanie `nval` do `lis`, z zachowaniem porządku kluczy

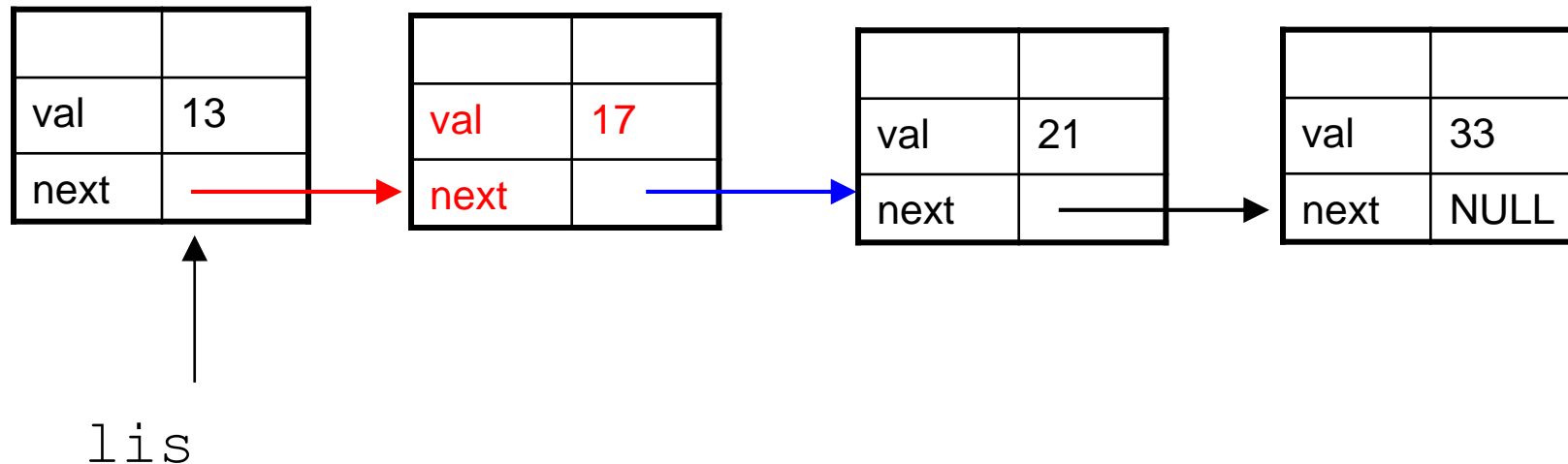
Lista uporządkowana: nowy element

wstawPorz(lis, 17)

Przed:



Po:



Lista uporządkowana: nowy element

Analiza przypadków:

1. Nowy element jest najmniejszy (zmiana „głowy” listy)
2. Nowy element jest największy (za ostatnim)
3. Nowy element pomiędzy dwoma innymi

Lista uporządkowana: nowy element

```
struct elem *wstawPorz(struct elem *lis,
                        int nval)
{ struct elem *nowy, *pom;

  nowy = utworz(nval);
  if (lis==NULL || lis->val >= nval){
    nowy->next = lis;
    return nowy;}
  else {
    pom=lis;
    while (pom->next != NULL &&
           pom->next->val < nval)
      pom=pom->next;
    nowy->next=pom->next;
    pom->next=nowy;
    return lis;
  }
}
```

```
def wstawPorz(lis, nval):
    nowy = ListItem(nval)
    if lis==None or lis.val >= nval:
        nowy.next = lis
        return nowy
    else:
        pom=lis
        while (pom.next != None and
               pom.next.val < nval):
            pom=pom.next
        nowy.next=pom.next
        pom.next=nowy
        return lis
```

Lista uporządkowana: nowy element

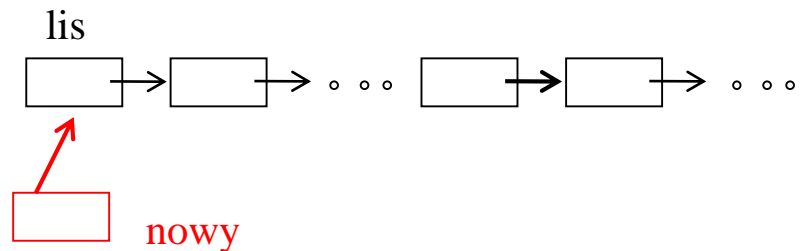
```
struct elem *wstawPorz(struct elem *lis,
    int nval)
{ struct elem *nowy, *pom;

    nowy = utworz(nval);
    if (lis==NULL || lis->val >= nval){
        nowy->next = lis;
        return nowy;}
    else {
        pom=lis;
        while (pom->next != NULL
            && pom->next->val < nval)
            pom=pom->next;
        nowy->next=pom->next;
        pom->next=nowy;
        return lis;
    }
}
```

Komentarz:

(lis==NULL || lis->val >= nval)

gdy nowy element powinien być umieszczony na początku listy.



Lista uporządkowana: nowy element

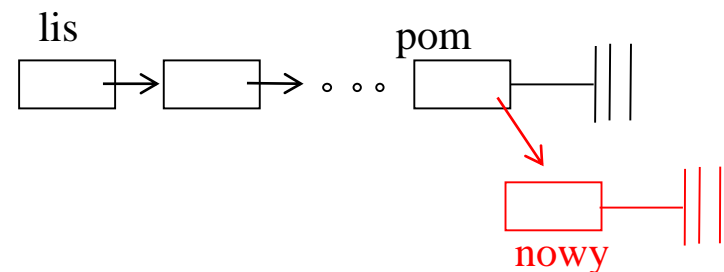
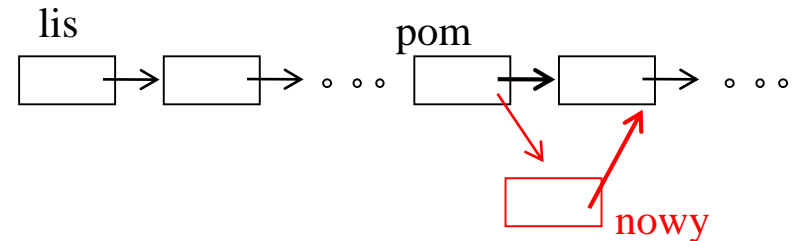
```
struct elem *wstawPorz(struct elem *lis,
                        int nval)
{ struct elem *nowy, *pom;

  nowy = utworz(nval);
  if (lis==NULL || lis->val >= nval){
    nowy->next = lis;
    return nowy;}
  else {
    pom=lis;
    while (pom->next != NULL
        && pom->next->val < nval)
      pom=pom->next;
    nowy->next=pom->next;
    pom->next=nowy;
    return lis;
  }
}
```

Komentarz dla „else”:

„zatrzymujemy” przeglądanie listy na elemencie

poprzedzającym miejsce, w które należy wstawić nowy element (jego następnikiem będzie **nowy**).



Lista uporządkowana: usuwanie

Problem (opisuje `usun(lis, uval)`)

Wejście:

`lis` – wskaźnik na listę

`uval` – klucz do usunięcia (usuwamy pierwsze wystąpienie)

Wyjście:

lista otrzymana przez usunięcie `uval` z `lis` (usuwamy pierwsze wystąpienie)

Lista: usuwanie

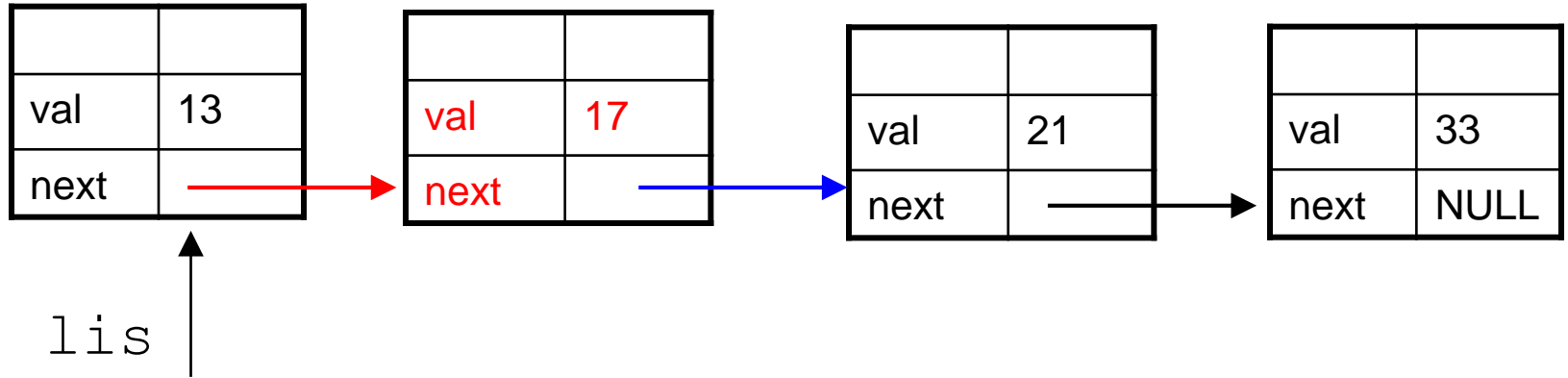
Analiza przypadków:

1. uval nie występuje na liście
2. uval pierwszy na liście
3. uval występuje, ale nie jest pierwszy na liście

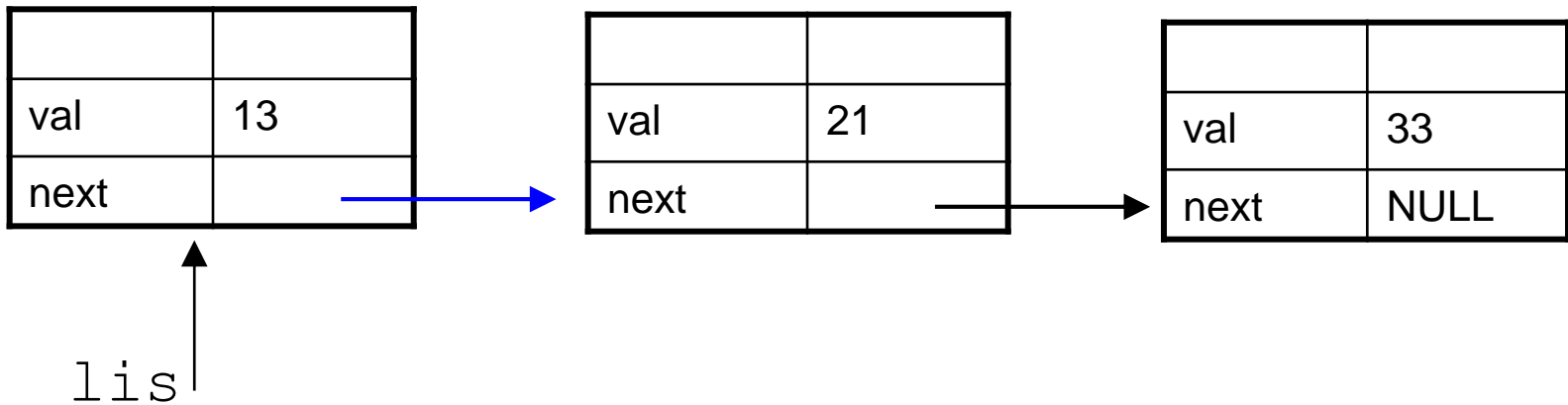
Lista: usuwanie

usun(lis, 17)

Przed:



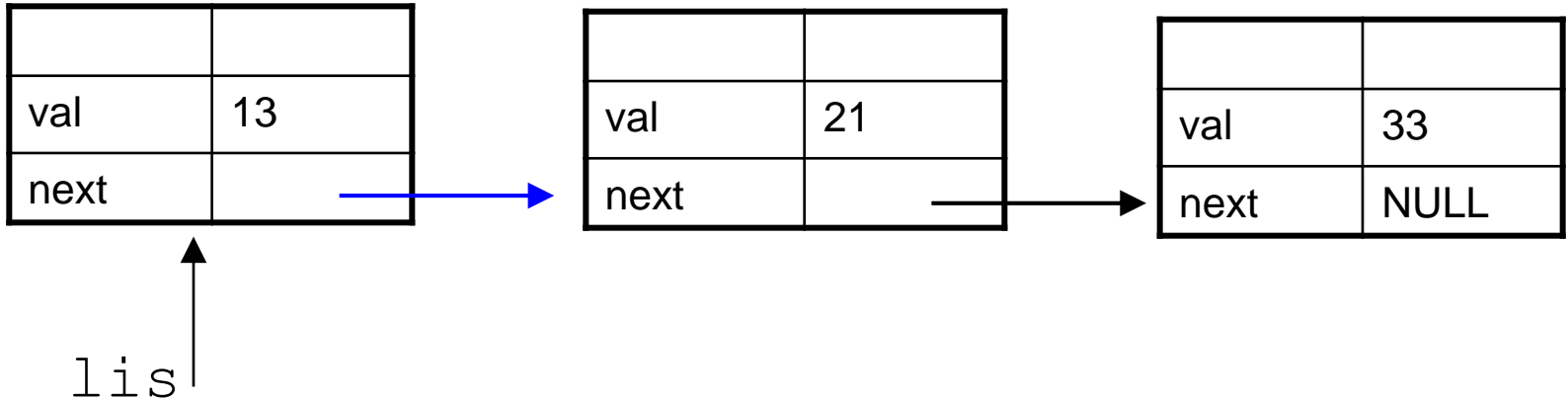
Po:



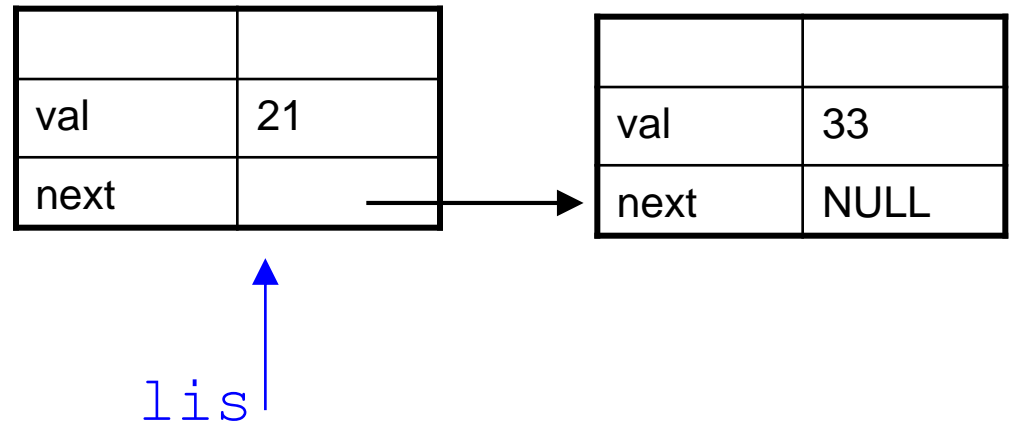
Lista: usuwanie

usun(lis, 13)

Przed:



Po:

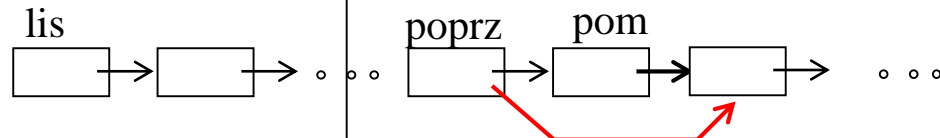


Lista: usuwanie

```
struct elem *usun(struct elem *lis, int uval)
{
    struct elem *pom, *poprz;
    pom = lis;
    while (pom != NULL && pom->val != uval){
        //poszukiwanie elementu uval
        poprz = pom;
        pom = pom->next;
    }
    if (pom != NULL) // na liscie wystapil uval
        if (pom == lis) { //do usuniecia pierwszy element
            lis=lis->next;
            free(pom);
        }
        else { // tworzymy powiazanie omijajace usuw.el.
            poprz->next = pom->next;
            free(pom);
        }
    return lis;
}
```

Komentarz:

- pierwsza pętla przechodzi przez listę w poszukiwaniu elementu o wartości pola val równej uval;
- usunięcie elementu z listy wymaga **zmiany następnika jego poprzednika**, dlatego pierwsza pętla przechowuje wskaźnik na element poprzedni w zmiennej poprz
- w sytuacji, gdy usuwamy pierwszy element, zmianie musi też ulec wartość wskaźnika na początek listy.

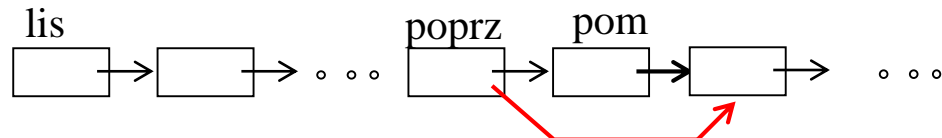


Lista: usuwanie

```
def usun(lis, uval):  
    pom = lis;  
    while pom != None and pom.val != uval:  
        #poszukiwanie elementu uval  
        poprz = pom  
        pom = pom.next  
    if pom != None: # na liscie wystapil uval  
        if pom == lis: #do usuniecia pierwszy  
            lis=lis.next  
        else: #tworzymy powiaz. omijajace  
            poprz.next = pom.next  
    return lis
```

Komentarz:

- pierwsza pętla przechodzi przez listę w poszukiwaniu elementu o wartości pola val równej uval;
- usunięcie elementu z listy wymaga **zmiany następnika jego poprzednika**, dlatego pierwsza pętla przechowuje wskaźnik na element poprzedni w zmiennej poprz
- w sytuacji, gdy usuwamy pierwszy element, zmianie musi też ulec wartość wskaźnika na początek listy.

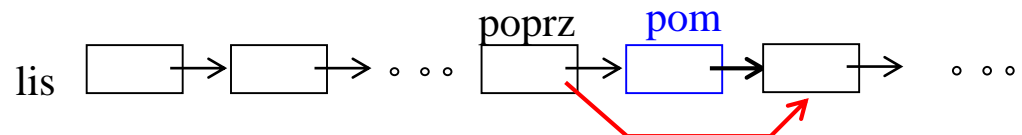


Lista: usuwanie

```
struct elem *usun(struct elem *lis, int uval)
{
    struct elem *pom, *poprz;
    pom = lis;
    while (pom != NULL && pom->val != uval){
        //poszukiwanie elementu uval
        poprz = pom;
        pom = pom->next;
    }
    if (pom != NULL) // na liscie wystapil uval
        if (pom == lis) { //do usuniecia pierwszy element
            lis=lis->next;
            free(pom);
        }
        else { // tworzymy powiazanie omijajace usuw.el.
            poprz->next = pom->next;
            free(pom);
        }
    return lis;
}
```

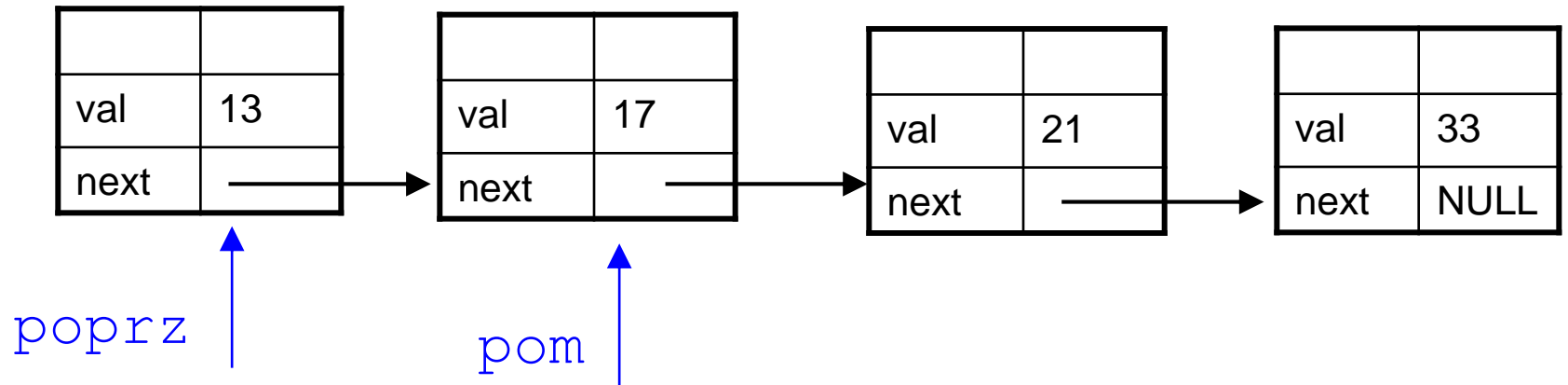
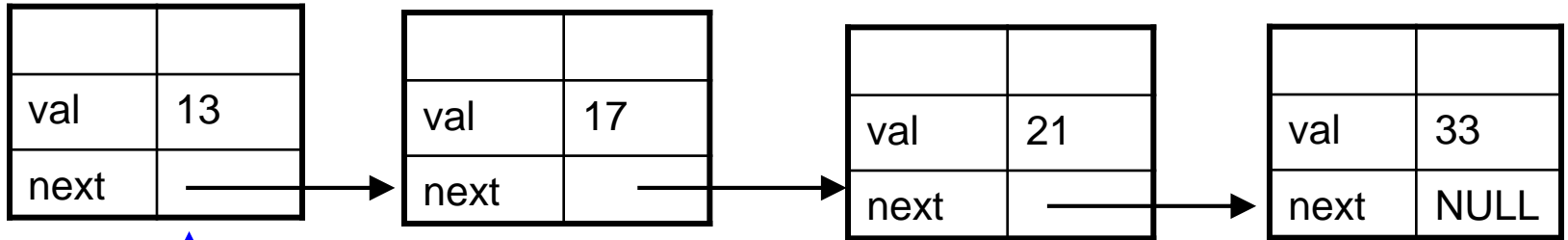
Komentarz:

- dynamicznie tworzone obiekty zajmują obszar „sterty”
- **C:** `free(pom)` zwalnia pamięć na stercie zajmowaną przez usuwany element
- **Python:** automatyczne „odśmiecanie” **sterty**



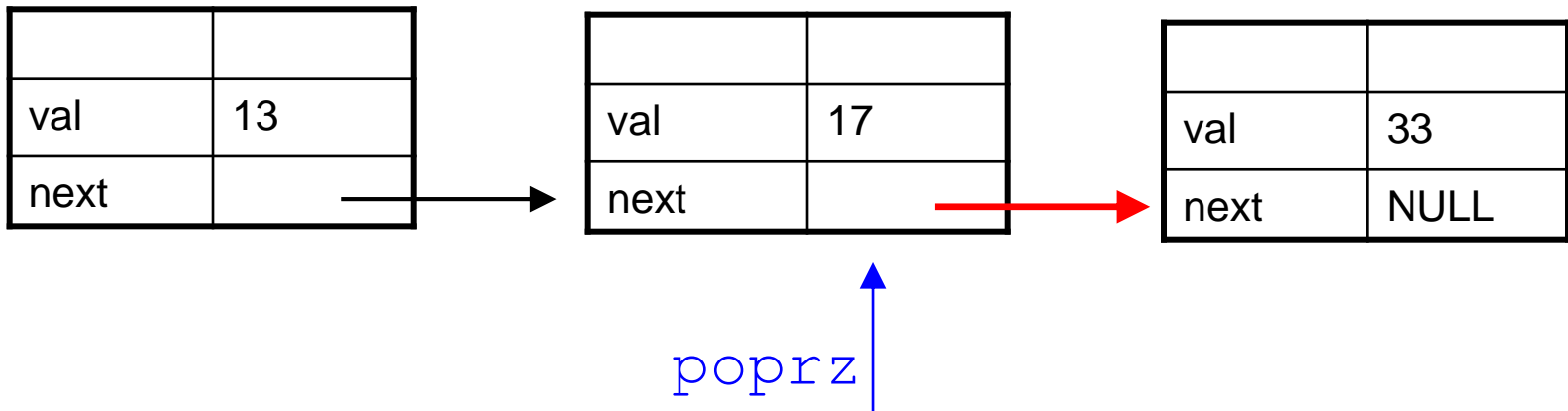
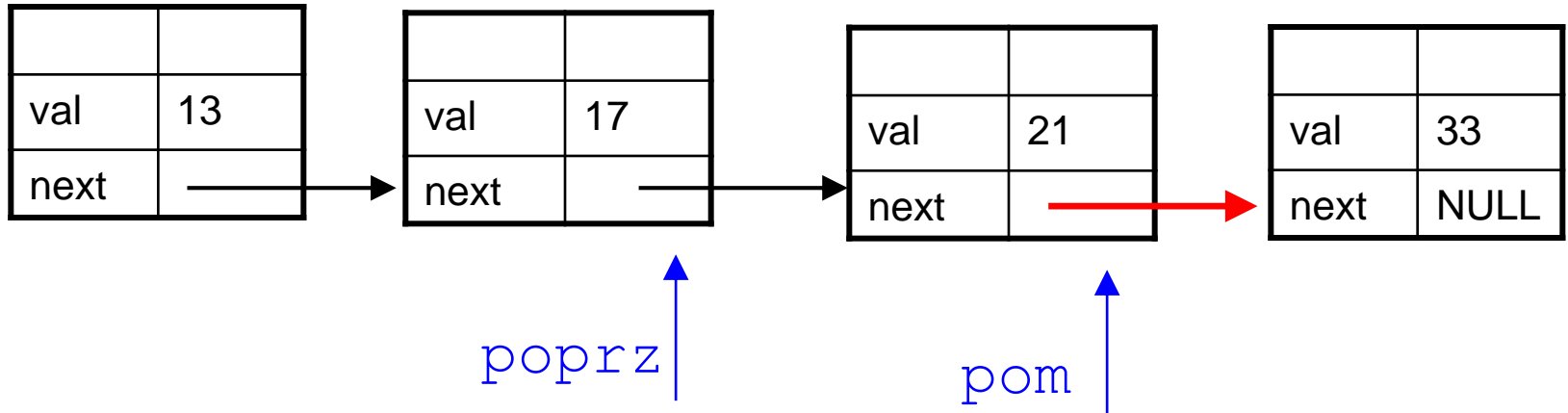
Lista: usuwanie

usun(lis, 21)



Lista: usuwanie

usun(lis, 21)



Lista – rekurencyjnie

Wypisanie elementów:

```
void wypiszR(struct elem *lis)
{
    if (lis != NULL){
        printf("%d\n", lis->val);
        wypiszR(lis->next);
    }
}
```

```
def wypiszR(lis):
    if lis != None:
        print lis.val
        wypiszR(lis.next)
```

Lista – rekurencyjnie

Wstawienie wartości nval do listy uporządkowanej lis:

- Jeśli lis pusta lub pierwszy element większy (lub równy) nval – wstaw nval na początek;
- W przeciwnym razie: wstaw nval do lis->next (lis.next)

Lista – rekurencyjnie

Wstawienie do listy uporządkowanej:

```
struct elem *wstawPorzR(struct elem *lis, int nval)
{ struct elem *nowy, *pom;

  if (lis==NULL || lis->val >= nval){
    nowy = utworz(nval);
    nowy->next = lis;
    return nowy;}
  else {
    pom = wstawPorzR(lis->next,nval);
    lis->next = pom;
    return lis;
  }
}
```

Lista – rekurencyjnie

Wstawienie do listy uporządkowanej:

```
def wstawPorzR(lis, nval):  
    if lis==None or lis.val >= nval:  
        nowy = ListItem(nval)  
        nowy.next = lis  
        return nowy  
    else:  
        pom = wstawPorzR(lis.next,nval)  
        lis.next = pom  
        return lis
```

Lista – rekurencyjnie

Usunięcie elementu o wartości uval z lis:

1. Jeśli lis pusta – zwróć list
2. Jeśli uval równy elementowi w głowie lis: zwróć lis->next (lis.next)
W przeciwnym razie: usuń uval z lis->next (lis.next)

Lista – rekurencyjnie

Usunięcie elementu z listy:

```
struct elem *usunRek(struct elem *lis, int uval)
{ struct elem *pom;

  if (lis != NULL)
    if (lis->val == uval) { //do usuniecia pierwszy element
      pom = lis;
      lis = lis->next;
      free(pom); }
    else{
      pom = usunRek(lis->next, uval);
      lis->next = pom;
    }
  return lis;
}
```

Lista – rekurencyjnie

Usunięcie elementu z listy:

```
def usunRek(lis, uval):  
    if lis != None:  
        if (lis.val == uval): #do usuniecia pierwszy element  
            lis = lis.next  
        else:  
            pom = usunRek(lis.next, uval)  
            lis.next = pom  
    return lis
```

Stos, Kolejka

Stos jako abstrakcyjna struktura danych

STOS (LIFO – last in, first out):

- Wartości: ciągi elementów określonego typu
- Operacje:
 - Wstaw na **szczyt** stosu (**początek ciągu**)
 - Odczytaj element ze **szczytu** stosu
 - Usuń element ze **szczytu** stosu
 - Zainicjuj stos (utwórz stos pusty)

Metody implementacji:

- Tablica?
- Lista liniowa?

Kolejka jako abstrakcyjna struktura danych

KOLEJKA (FIFO – first in, first out):

- Wartości: ciągi elementów określonego typu
- Operacje:
 - Dodaj element na **koniec** kolejki
 - Odczytaj element z **początku** kolejki
 - Usuń element z **początku** kolejki
 - Zainicjuj kolejkę (utwórz pustą kolejkę)

Metody implementacji:

- Tablica?
- Lista liniowa? Z bezpośrednim dostępem do pierwszego i **ostatniego** elementu listy!

Podsumowanie

- **Abstrakcyjny** typ danych jako narzędzie programowania strukturalnego/obiektowego
- Listy **wiązane** – efektywne narzędzie implementacji w zastosowaniach, gdzie wstawianie/usuwanie głównie na koniec/początek
- **Kolejka, stos** – popularne abstrakcyjne struktury danych implementowane za pomocą list wiązanych