Algorytmy i SD

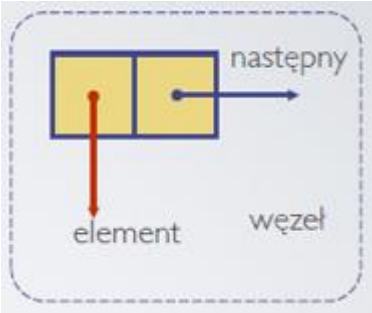
Struktury danych - Listy, kolejki

Piotr Ciskowski, Łukasz Jeleń
Wrocław, 2023

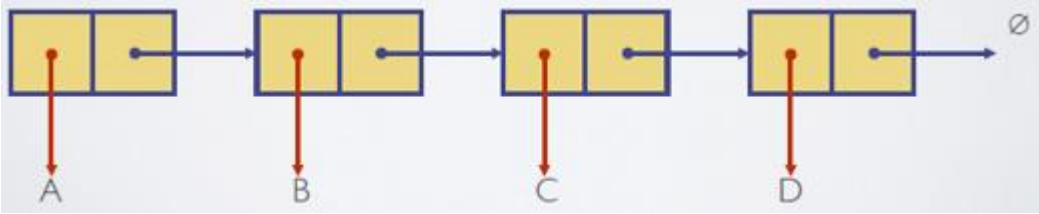
ADT **lista jednokierunkowa:**

przechowuje dowolne obiekty
jest konkretna strukturą danych, składającą się z sekwencji węzłów

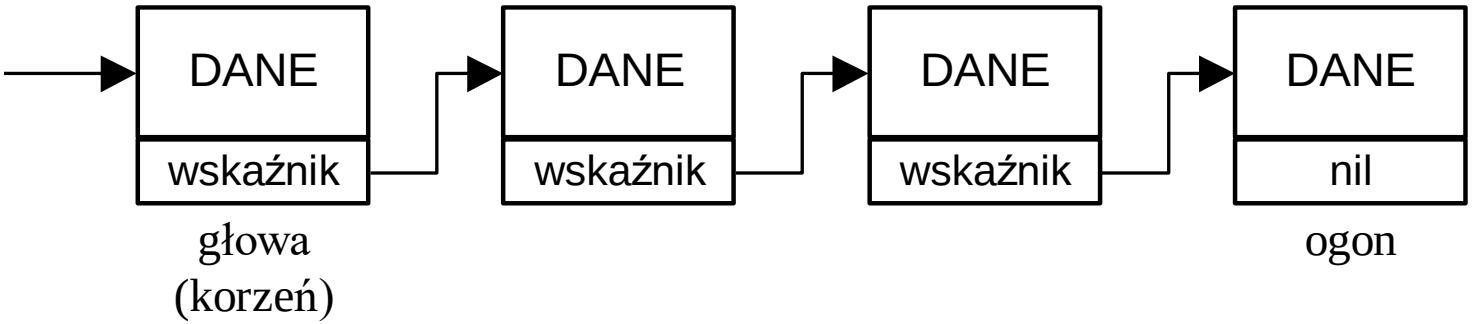
każdy węzeł przechowuje:
- element
- link do następnego węzła



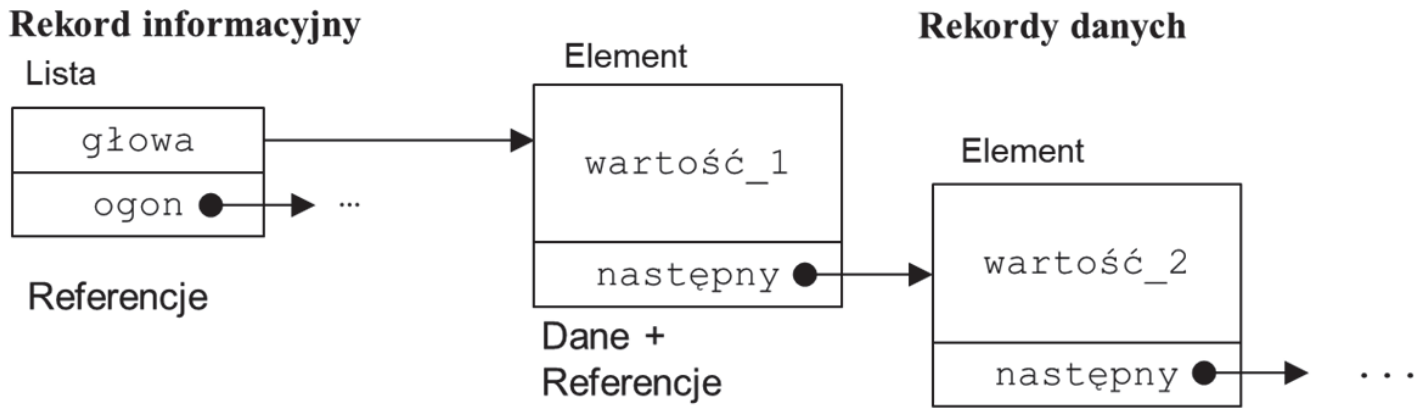
lista ma dodatkowy węzeł, w którym pamięta:
- *head* – adres pierwszego węzła
- a czasem też ostatniego



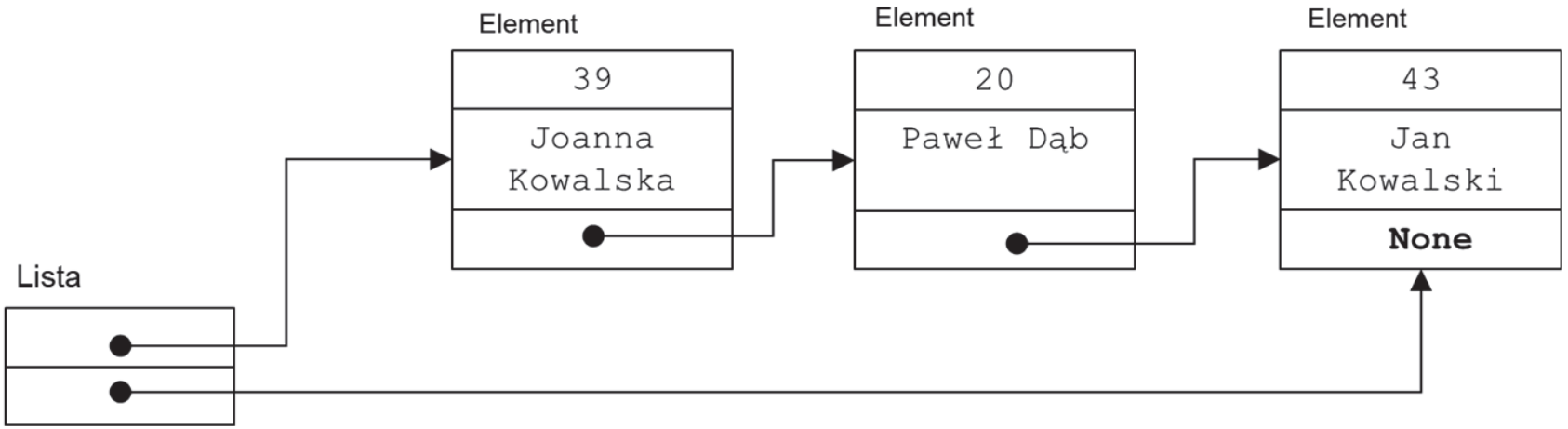
rysunek: Łukasz Jeleń


DANE
wskaźnik
DANE
wskaźnik
DANE
wskaźnik
DANE
nil
głowa
(korzeń)
ogon

struktury danych – lista jednokierunkowa

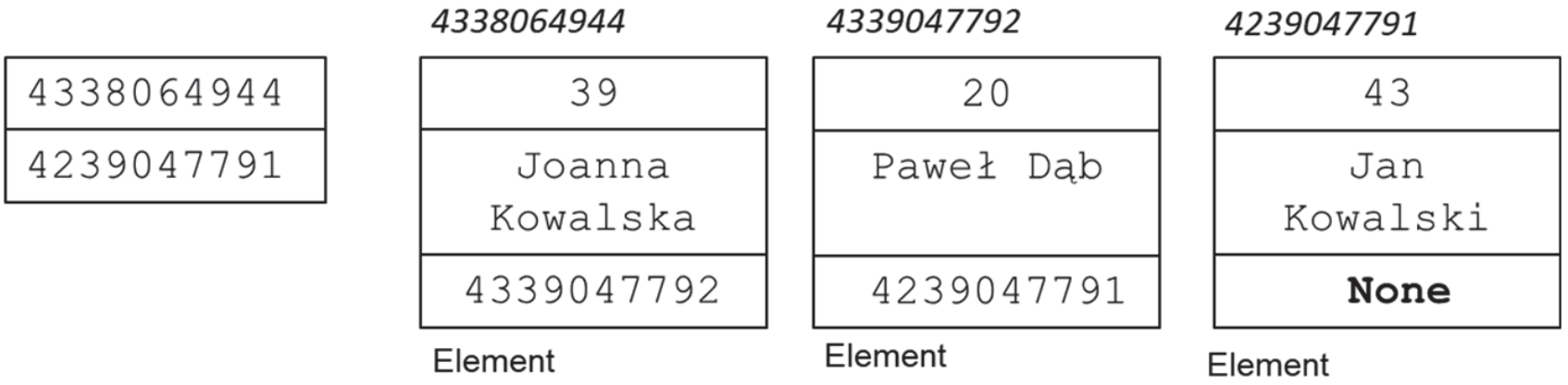


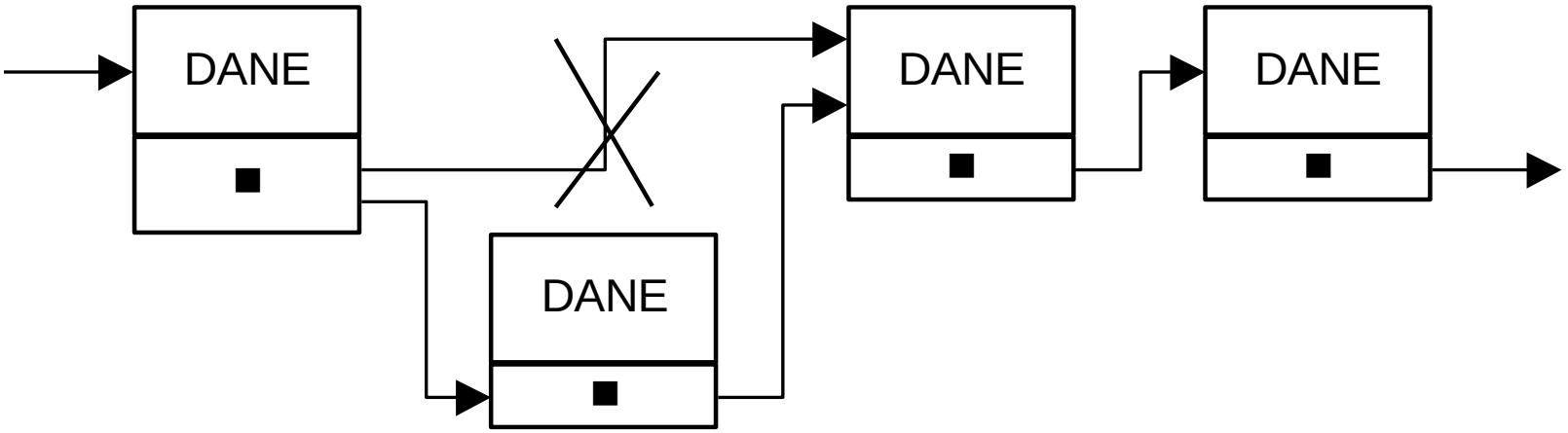
struktury danych – lista jednokierunkowa



rysunek: Piotr Wróblewski. Algorytmy w Pythonie

struktury danych – lista jednokierunkowa

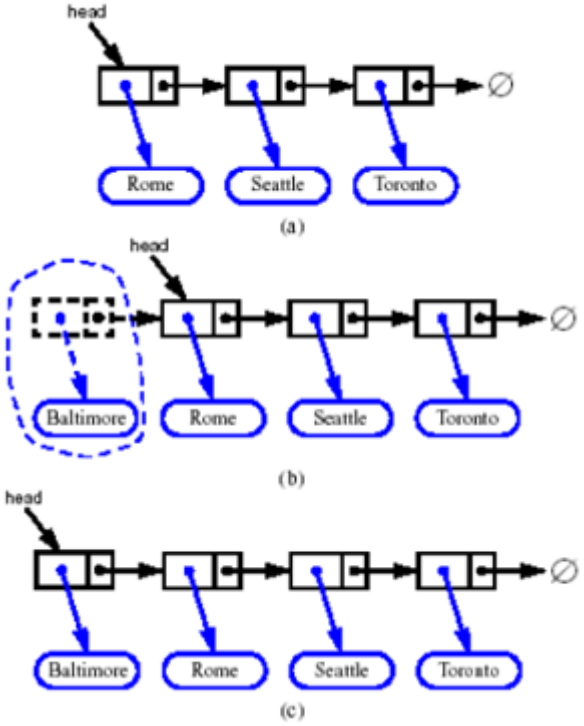



DANE
DANE
DANE
DANE

lista jednokierunkowa - operacje**:**

- **dodawanie na początku**

  1. alokacja nowego węzła
  2. dodanie nowego elementu
  3. dodanie powiązania
     - nowy węzeł wskazuje na początek listy (dotychczasowy *head*)
  4. aktualizacja *head*
     - wskazuje na nowy węzeł



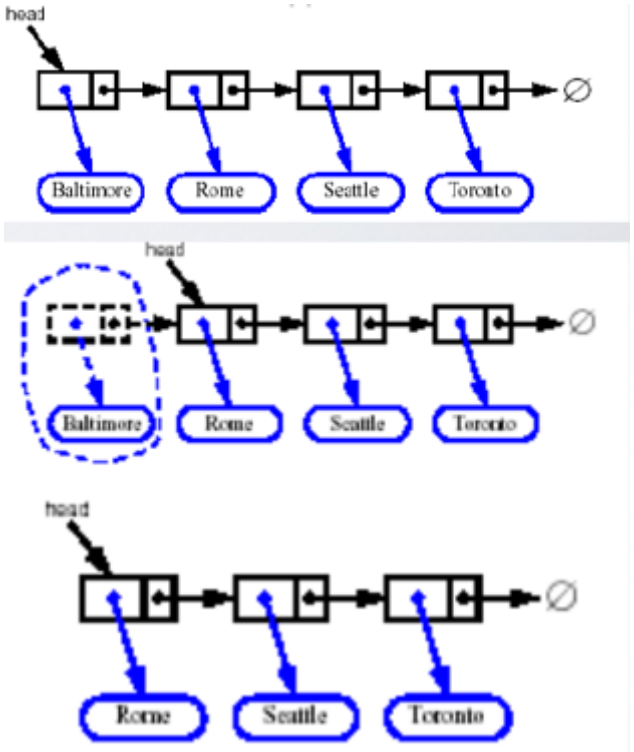
lista jednokierunkowa - operacje**:**

- **usuwanie z początku**

  1. zapamiętanie *head*
     - adres pierwszego węzła

  2. aktualizacja *head*
     - wskazuje na drugi węzeł

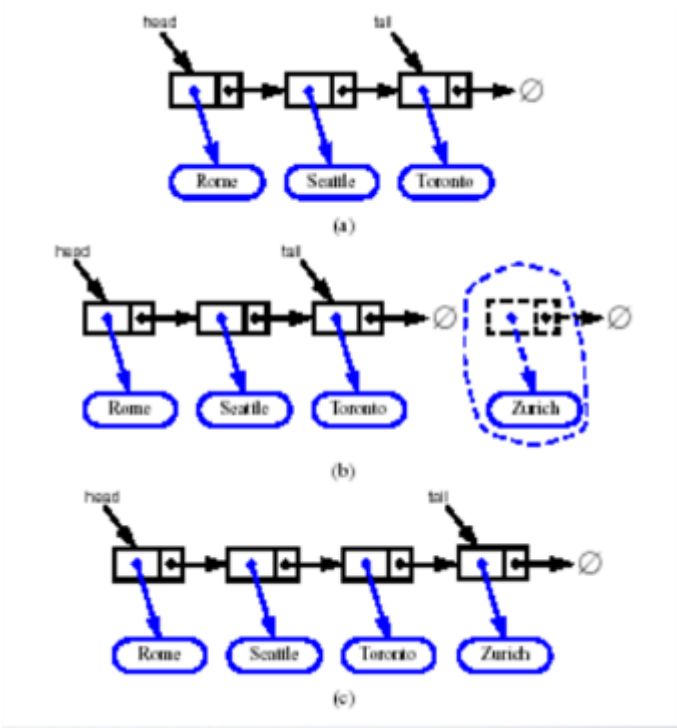
  3. usunięcie pierwszego węzła



lista jednokierunkowa - operacje**:**

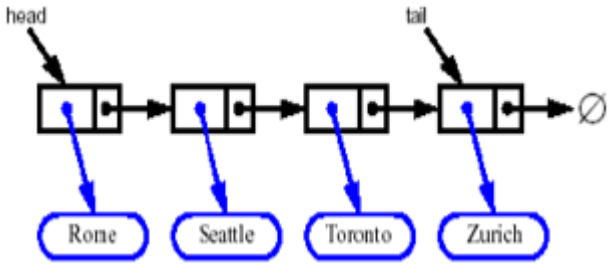
- **dodawanie z tyłu**

  1. alokacja nowego węzła
  2. dodanie nowego elementu
  3. dodanie powiązania
     - nowy węzeł wskazuje na *null*
  4. aktualizacja *tail*
     - wskazuje na nowy węzeł



lista jednokierunkowa - operacje**:**

- **usuwanie z tyłu**
  - sami sobie wykombinujcie kroki ;-)
  - nie jest wydajne
  - nie da się uaktualnić *tail* w stałym czasie
    - trzeba przejrzeć listę od początku
      (aby znaleźć poprzedni element)



przykładowa implementacja w Pythonie

```
class Element:                  # Rekord danych
  def __init__(self, pDane = None, pNastepny=None):
    self.dane = pDane
    self.nastepny = pNastepny

class Lista:          # Nowa lista jest pusta (referencja 'None')
  def __init__(self):
    self.glowa = None
    self.ogon = None
    self.dlugosc=0
```

struktury danych – lista jednokierunkowa

```
l = Lista()      # Tworzymy pustą listę
q = Element(3)
l.glowa = q      # Wstawiamy nowy element 'q' na koniec listy
l.ogon  = q
r = Element(5)
q.nastepny= r    # Wstawiamy kolejny element na koniec...
l.ogon    = r    # Aktualizujemy odsyłacz do końca listy
```

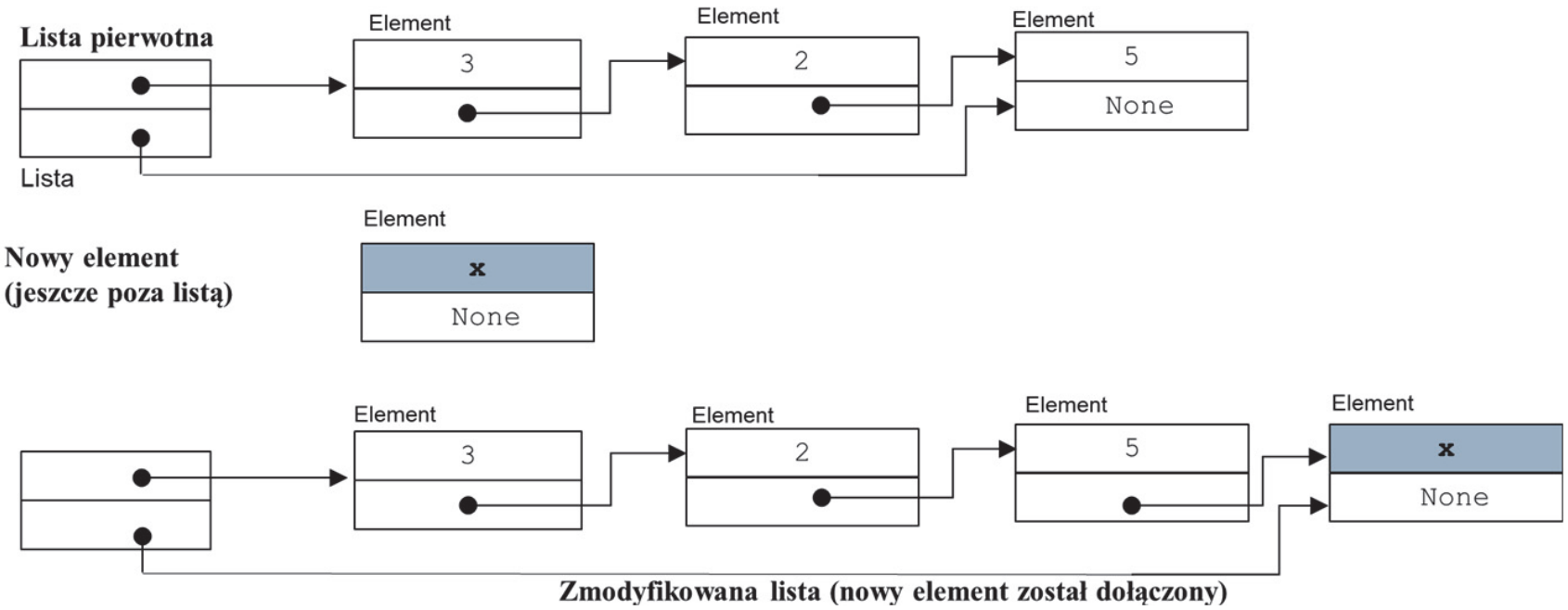
```
x=5
adres_tmp=l.glowa                # Idziemy na początek listy
while adres_tmp!= None:
  if adres_tmp.dane==x:
      print("Znalazłem poszukiwany element")
      break
  adres_tmp=adres_tmp.nastepny   # Idziemy do kolejnego elementu listy

if adres_tmp == None:
    print("Nie znaleziono poszukiwanego elementu")
```

struktury danych – lista jednokierunkowa

```
def wypisz(self):
    tmp=self.glowa
    if tmp == None:
        print("\n *Lista pusta*")
        return
    while tmp != None: # Pętla przechodząca przez sekwencję danych
        print(tmp.dane, end=" ")
        tmp = tmp.nastepny
    print("\n")        # Znak końca linii
```

```
def wstawNaKoniec(self, pDane):
    x = Element(pDane)
    if (self.glowa==None): # Lista pusta
        self.glowa=x
        self.ogon =x

    else:
        self.ogon.nastepny=x
        self.ogon=x
    self.dlugosc = self.dlugosc + 1 # Aktualizujemy umowne pole zawierające liczbę
                                    # elementów na liście
```

struktury danych – lista jednokierunkowa



struktury danych – lista jednokierunkowa

```
def szukaj(self, x):
    tmp = self.glowa
    while tmp != None:
        if tmp.dane == x:
            print("\nZnalazłem poszukiwany element ", x)
            break
        tmp = tmp.nastepny
    if tmp == None:
        print("\nNie znaleziono poszukiwanego elementu ", x)
```

struktury danych – lista jednokierunkowa

```
def szukajRef(self, x): # Odszukaj element na liście i zwróć jego pozycję
    biezacy = self.glowa
    poprzedni=None
    while biezacy != None:
        if biezacy.dane == x:
            return poprzedni, biezacy, True  # Znaleziono element
        poprzedni=biezacy
        biezacy = biezacy.nastepny
    return poprzedni, biezacy, False         # Nie znaleziono elementu
```

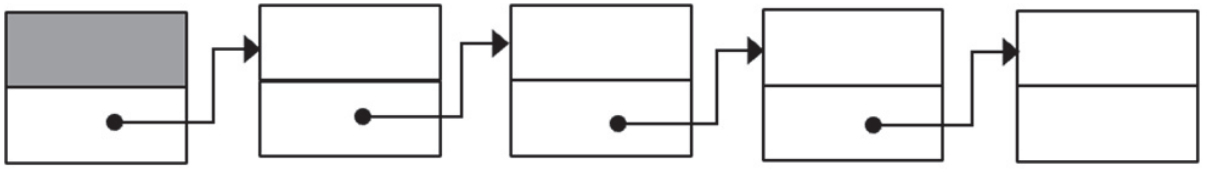
```
poprzedni, biezacy, znaleziono = self.szukajRef(x)
```

struktury danych – lista jednokierunkowa

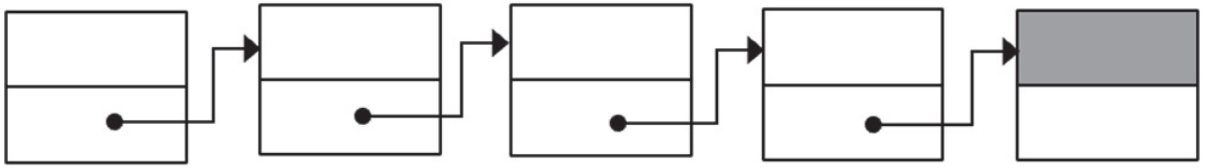
```
def wstawSort(self, x):
    nowy=Element(x)
    self.dlugosc = self.dlugosc + 1
# Poszukiwanie właściwej pozycji na wstawienie elementu:



# Poszukiwanie miejsca na wstawienie:



# Wstawiamy, analizując wartości zapamiętane w 'przed' i 'po'
```

struktury danych – lista jednokierunkowa

a) Wstawiamy na początek listy (*przed* = None)

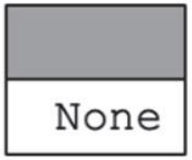


b) Wstawiamy na koniec listy (*po* = None)

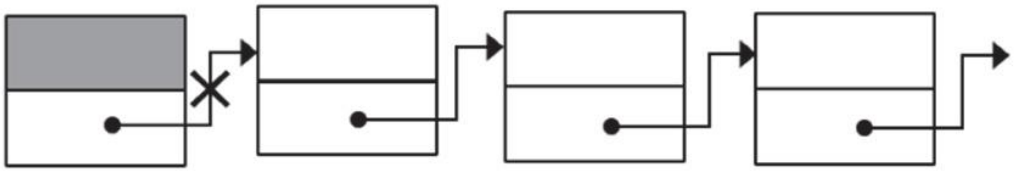


c) Wstawiamy w środek listy (*przed* ≠ None, *po* ≠ None)

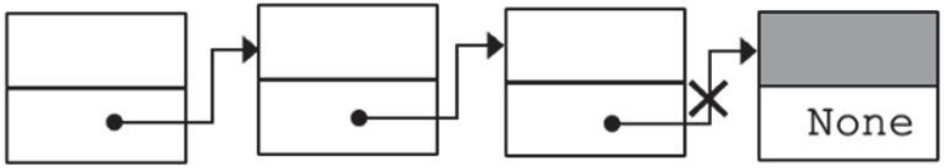
a) Lista ma długość 1



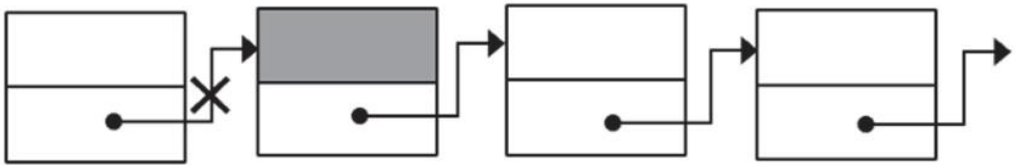
b) Lista ma długość >1, usuwamy z *przodu*



c) Lista ma długość >1, usuwamy z *końca*



d) Lista ma długość >1, usuwamy gdzieś ze *środka*



struktury danych – lista jednokierunkowa

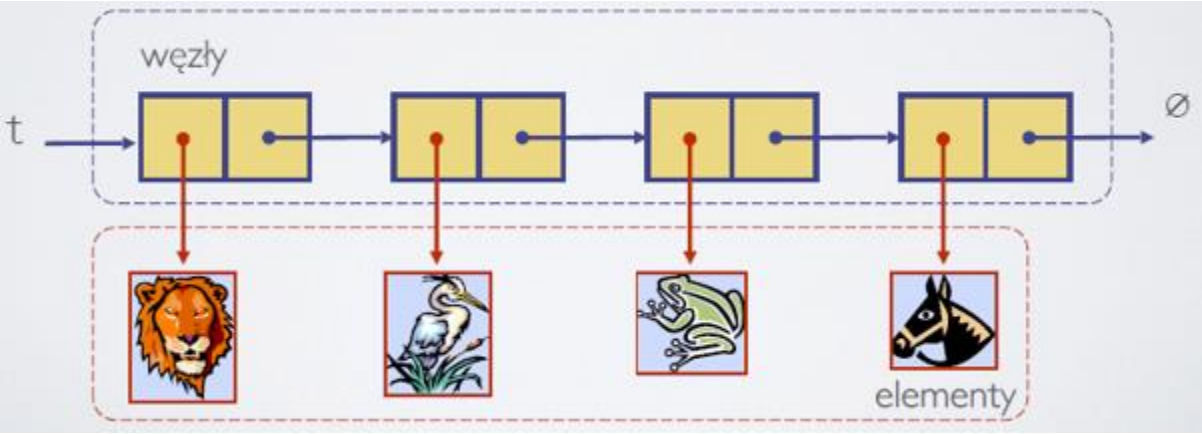
```
def usunWybrany(self, x): # Odszukaj i usuń element z listy
    poprzedni, biezacy, znaleziono = self.szukajRef(x) # Szukamy elementu i jego pozycji:

    if znaleziono==False:
        print("Nie znaleziono elementu")
        return
    self.dlugosc = self.dlugosc - 1      # Skracamy parametr opisujący długość listy o 1

    if self.dlugosc == 0:                # Przypadek a) – lista jednoelementowa
        self.glowa = None
        self.ogon  = None
        self.dlugosc=0
        return

    if self.glowa==biezacy:              # Przypadek b) – usuwamy z przodu
        self.glowa = biezacy.nastepny.   # Przestawiamy wskaźnik "glowa"
        return

    if self.ogon==biezacy:                # Przypadek c) – usuwamy z tyłu
        self.ogon = poprzedni             # Przestawiamy wskaźnik "ogon"
        poprzedni.nastepny=None           # Zaznaczamy nowy koniec listy
        return
                                          # Przypadek d) – usuwamy ze środka:
    poprzedni.nastepny = biezacy.nastepny # Przestawiamy wskaźniki "ogon"
```

lista jednokierunkowa – jako stos**:**

- możemy zaimplementować stos
  za pomocą listy jednokierunkowej

- górny element stosu przechowywany jest
  w pierwszym węźle listy

- wykorzystanie miejsca: *O*(*n*)
- każda operacja stosu wykonywana w czasie *O*(1)



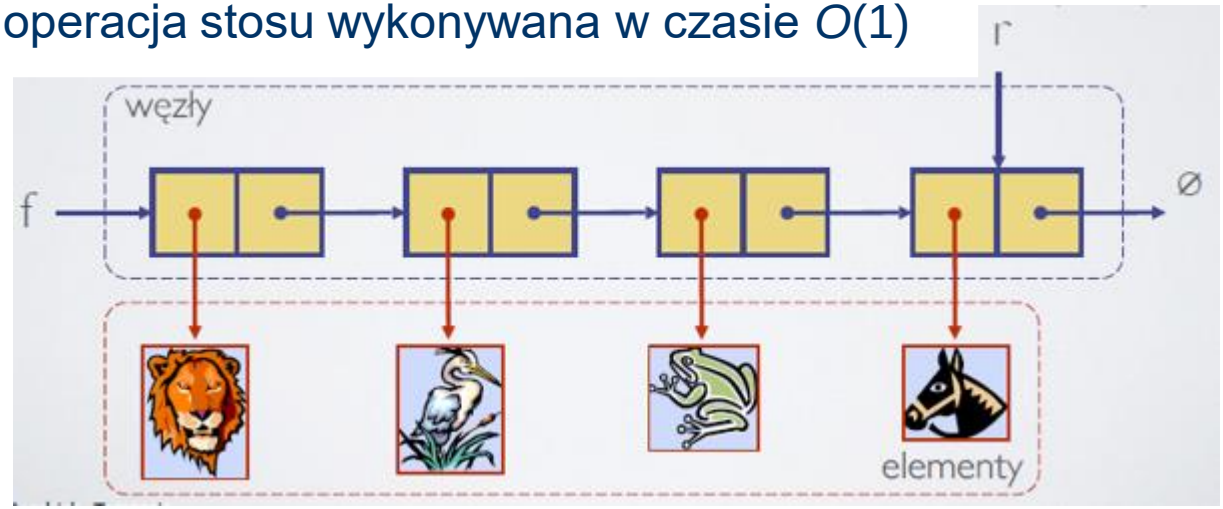
rysunek: Łukasz Jeleń

lista jednokierunkowa – jako kolejka**:**

- możemy zaimplementować kolejkę
  za pomocą listy jednokierunkowej

- element początkowy przechowywany jest w pierwszym węźle listy
- ostatni element przechowywany jest w ostatnim węźle

- wykorzystanie miejsca: $O(n)$
- każda operacja stosu wykonywana w czasie $O(1)$

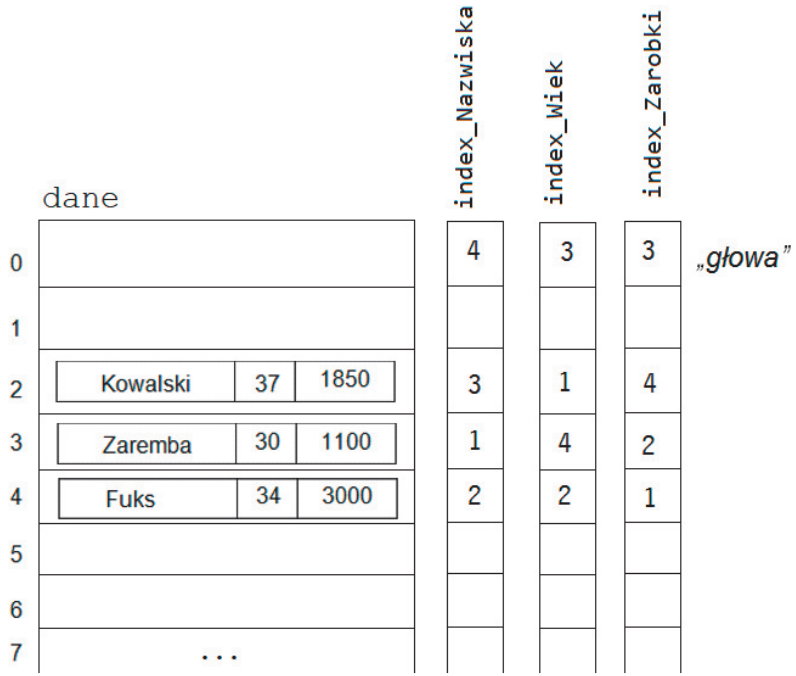


rysunek: Łukasz Jeleń

Spójrzmy obiektywnie na listy jednokierunkowe pod kątem ich wad i zalet:

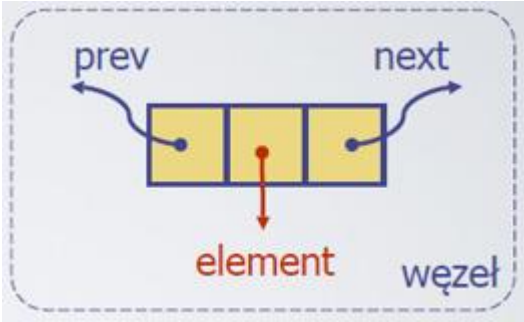
- *Wady*: nienaturalny dostęp do elementów, niełatwe sortowanie, utrudniona analiza zawartości i ocena wielkości listy.
- *Zalety*: efektywne zużycie pamięci, elastyczność.

struktury danych – lista jednokierunkowa

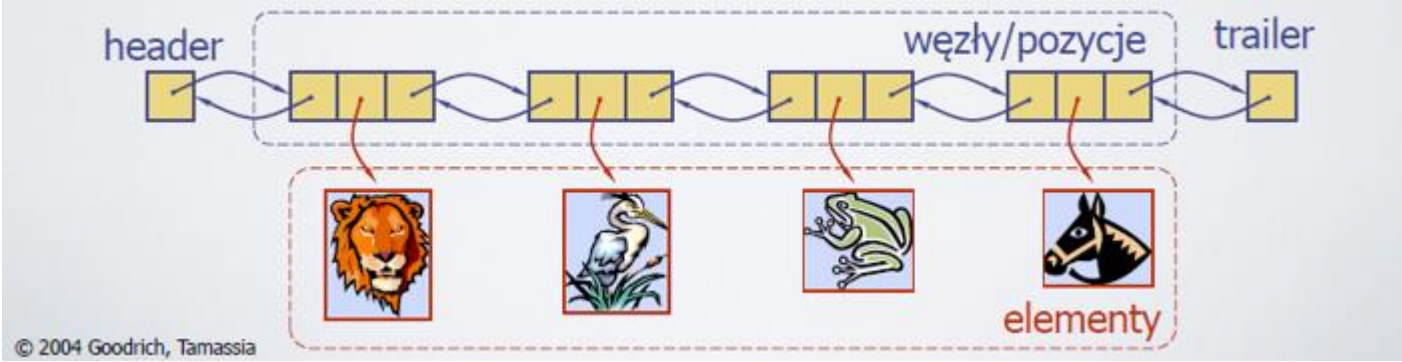


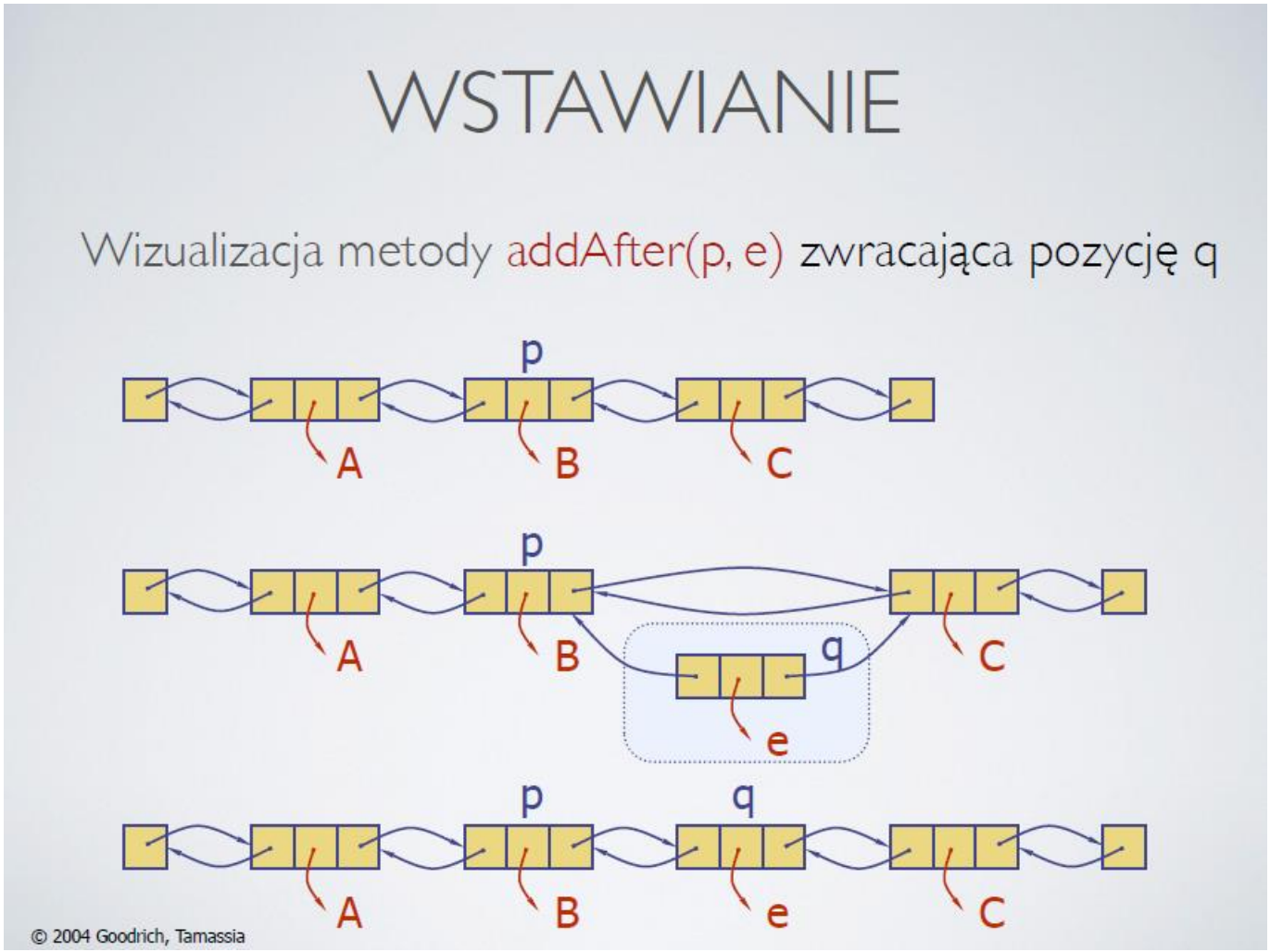
ADT **lista dwukierunkowa:**

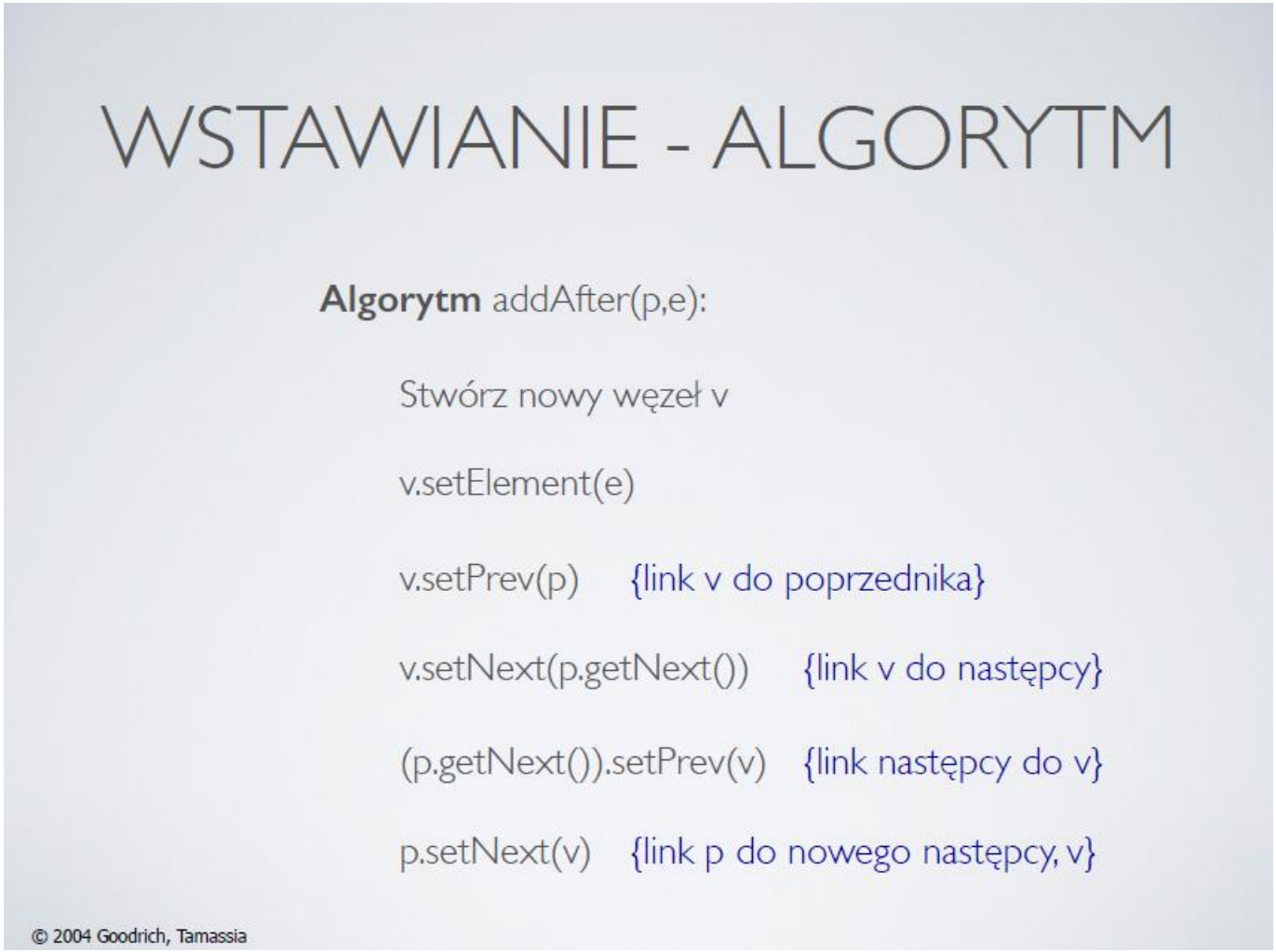
każdy węzeł przechowuje:
- element
- link do następnego węzła
- link do poprzedniego węzła

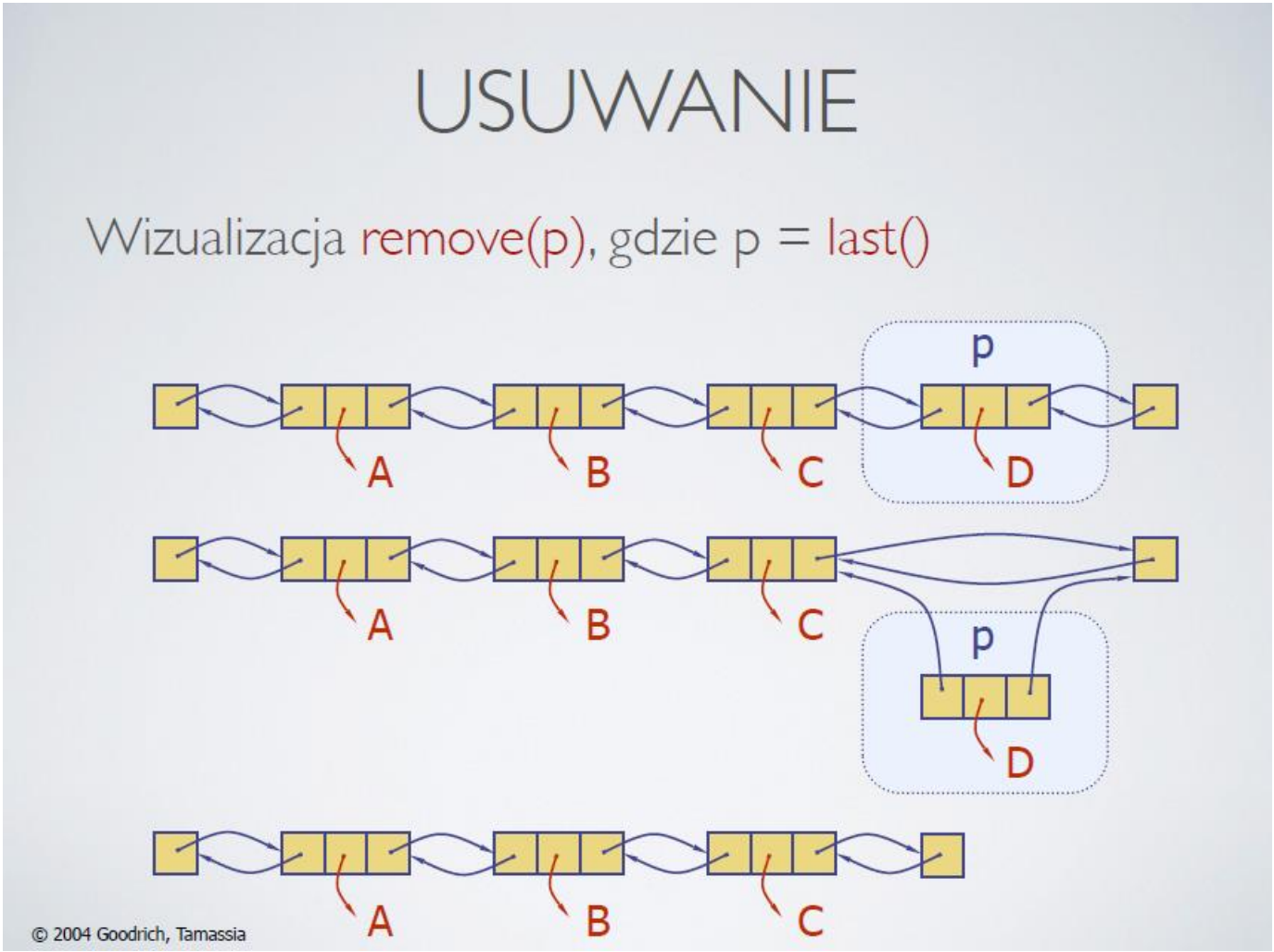


zawiera dodatkowe węzły:
- *header*
- *trailer*




WSTAWIANIE
Wizualizacja metody addAfter(p, e) zwracająca pozycję q
p
A
B
C
p
A
B
q
C
e
p
q
A
B
e
C
© 2004 Goodrich, Tamassia


WSTAWIANIE - ALGORYTM
Algorytm addAfter(p,e):
Stwórz nowy węzeł v
v.setElement(e)
v.setPrev(p) {link v do poprzednika}
v.setNext(p.getNext()) {link v do następcy}
(p.getNext()).setPrev(v) {link następcy do v}
p.setNext(v) {link p do nowego następcy, v}
© 2004 Goodrich, Tamassia

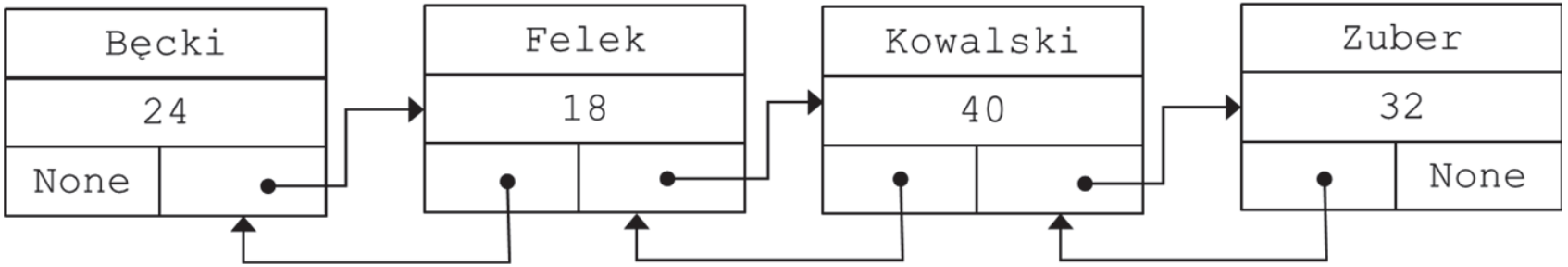

USUWANIE
Wizualizacja remove(p), gdzie p = last()
p
A
B
C
D
A
B
C
p
D
A
B
C
© 2004 Goodrich, Tamassia

USUWANIE - ALGORYTM

Algorytm remove(p):

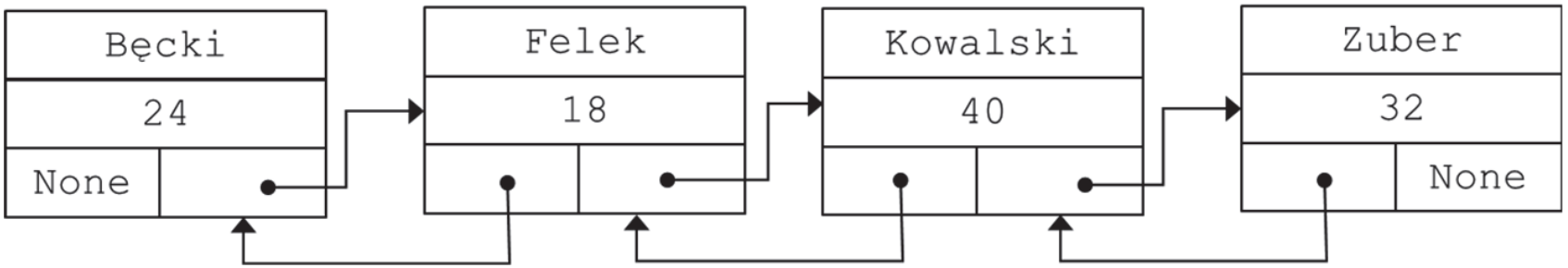
```
t ← p.element    {tymczasowa zmienna do przechowywania
                                    zwracanej wartości}
(p.getPrev()).setNext(p.getNext()) {usuwanie linków do  p}
(p.getNext()).setPrev(p.getPrev())
p.setPrev(null)    {usuwanie linków z p}
p.setNext(null)
return t
```

struktury danych – lista dwukierunkowa



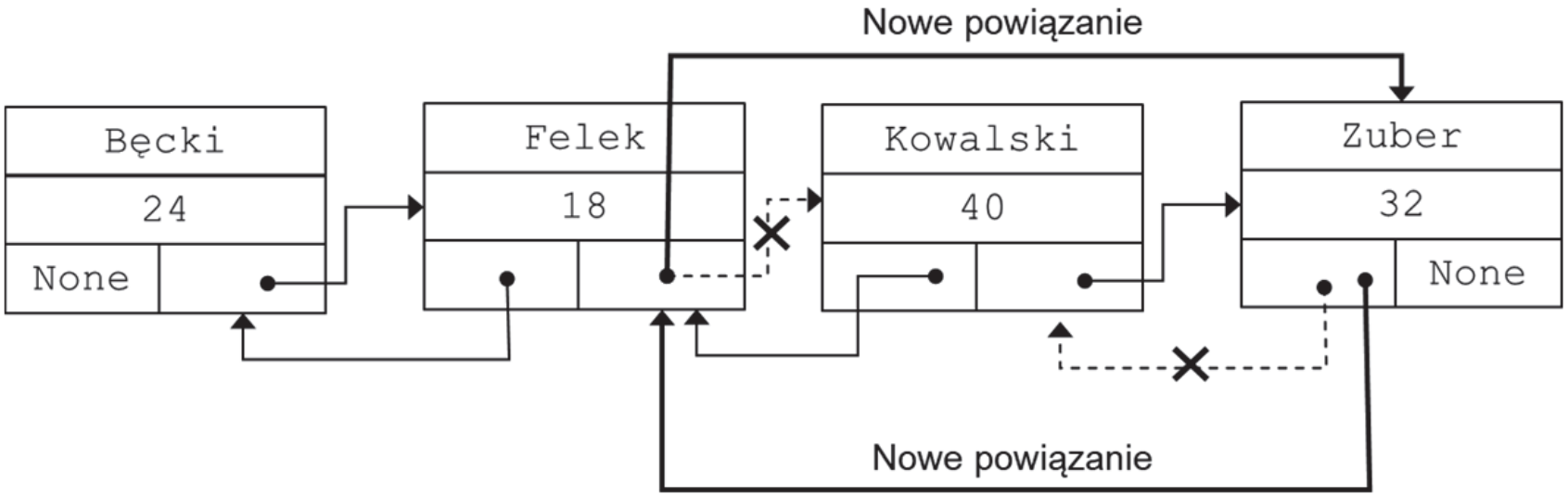

```
class Element:       # Rekord danych
    def __init__(self, pNazwisko="Doe", pWiek=0):
        self.nazwisko = pNazwisko
        self.wiek = pWiek
        self.nastepny = None
        self.poprzedni = None

class Lista2Kier:    # Właściwa lista dwukierunkowa
    def __init__(self):
       self.glowa = None
       self.ogon = None
```

struktury danych – lista dwukierunkowa




```
def wstaw(self, pNazwisko, pWiek):   # Proste wstawianie na koniec listy
    nowy = Element(pNazwisko, pWiek)
    if self.glowa !=None:
         self.ogon.nastepny = nowy
         nowy.poprzedni = self.ogon
         self.ogon=nowy
    else:
         self.glowa = nowy
         self.ogon =  nowy                    }
```

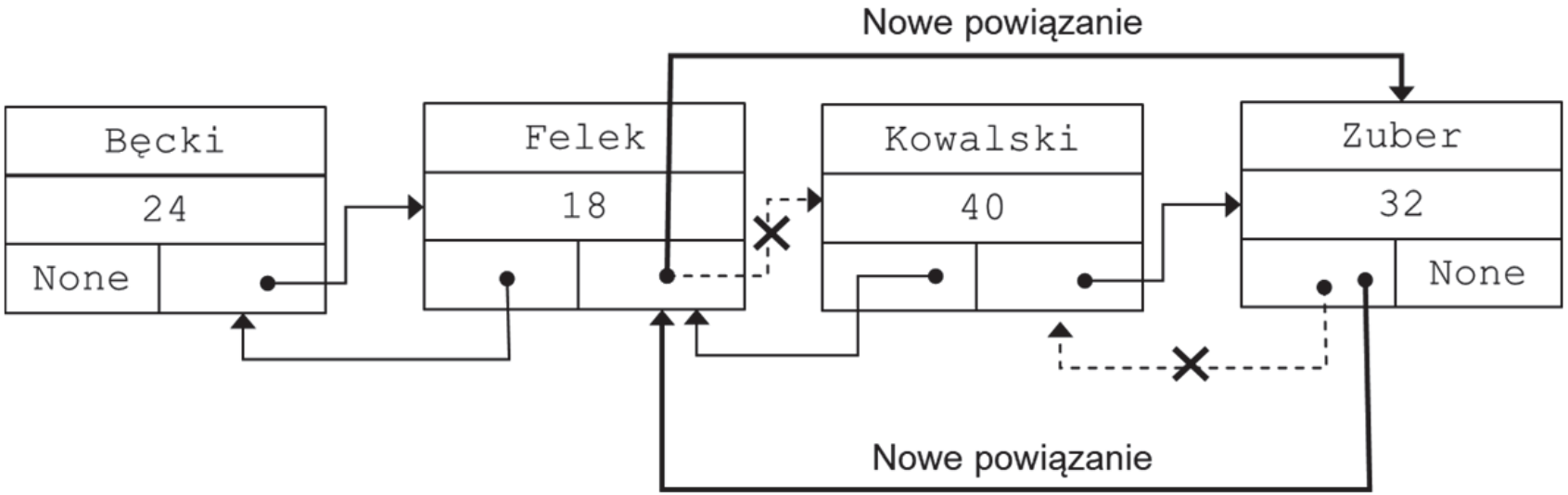
struktury danych – lista dwukierunkowa




```
def usun(self, pNazw):          # Odszukaj i usuń rekord pasujący do kryterium wyszukiwania
    res, znaleziono = self.szukaj(pNazw)
    if znaleziono==False:
        print(f"Brak [{pNazw}] na liście")
        return
    print(f"Usuwam [{pNazw}] z listy")

    if (res.poprzedni) != None and res.nastepny!=None:    # Środek
        (res.poprzedni).nastepny = res.nastepny
        (res.nastepny).poprzedni = (res.poprzedni).nastepny
        return

    if res == self.glowa:                                 # Usuwam z przodu
        self.glowa=res.nastepny
         (res.nastepny).poprzedni=None
    else:                                                 # Usuwam z tyłu
         (res.poprzedni).nastepny = None
          self.ogon = res.poprzedni
```

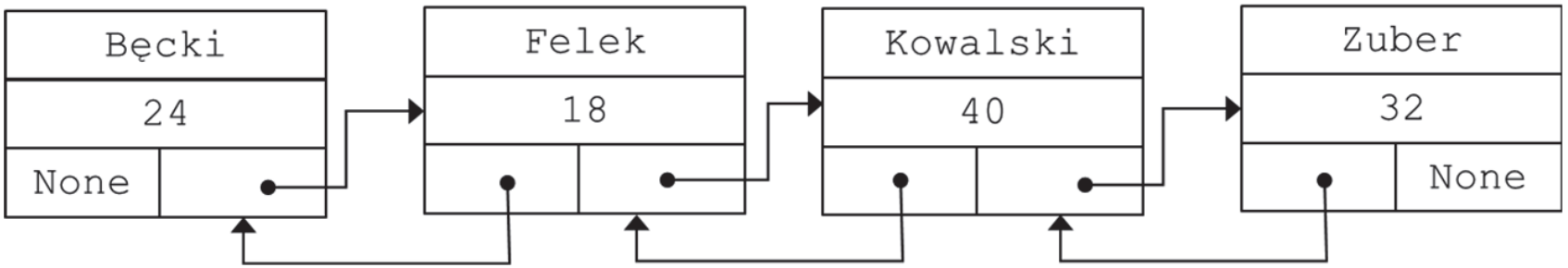
rysunek: Piotr Wróblewski. Algorytmy w Pythonie

struktury danych – lista dwukierunkowa




```
def szukaj(self, pNazw): # Odszukaj rekord 'pNazw' na liście
    tmp = self.glowa     # Zmienna zapamiętująca status przeszukiwania listy
    znaleziono=False
    while tmp != None:
        if tmp.nazwisko==pNazw:
           znaleziono=True
           break                # Wychodzimy z pętli
        else:
            tmp=tmp.nastepny    # Idź dalej
    if znaleziono==True:
        return tmp, True        # Zwróć wynik poszukiwań (referencja do rekordu)
    else:
        return None, False      # Nic nie znaleziono!
```

struktury danych – lista dwukierunkowa




```
def wypiszWprzod(self, s):
    print(s)
    tmp=self.glowa
    while (tmp !=None):
        print(f"[{tmp.nazwisko}, {tmp.wiek}]", end=" ")
        tmp = tmp.nastepny
    print("")

def wypiszWstecz(self,s ):
    print(s)
    tmp=self.ogon
    while (tmp != None):
        print(f"[{tmp.nazwisko}, {tmp.wiek}]", end=" ")
        tmp = tmp.poprzedni
    print("")
```

ADT **sekwencja:**

- struktura danych rozszerzająca definicję listy
- zawiera funkcje pozwalające na dostęp do elementu przez podanie jego indeksu – jak dla wektora
- interfejs – jak dla listy plus:
  - atIndex(i) – zwraca element na pozycji *i*
  - indexOf(p) – szuka elementu *p* i zwraca jego pozycję

```
class NodeSequence :  public NodeList {
 public:
   Iterator atIndex( int i)  const;
                                // zwraca element na pozycji i

   int indexOf( const Iterator& p)  const;
                                // zwraca indeks elementu p
};
```

struktury danych – lista z iteratorem

```
class Element:
    def __init__(self, pDane, pNastepny=None):
        self.dane = pDane
        self.nastepny = pNastepny
# Dalej metody klasy 'Element':
    def wypiszElementy(self):
    ...
# ------------------------------------------------------------------------
class Lista:
    def __init__(self):
        self.glowa = None
        self.ogon = None
        self.dlugosc = 0
# Dalej metody klasy 'Element':
    def wstawNaKoniec(self, pDane):
    <...>
    def wypisz(self):
    <...>
    def szukaj(self, x)
    <...>
    def szukajRef(self,
    <...>
    def usunWybrany(sel
    <...>

    def __iter__(self):
        return MojIterator(self)
```

```
class MojIterator:
    def __init__(self, pLista):
        self._kursor = pLista.glowa

    def __next__(self):          # Zwraca kolejny element z kolekcji obsługiwanej przez iterator
        if self._kursor !=None:  # (*)
            res= "["+ str(self._kursor.dane)+"]"
            self._kursor=self._kursor.nastepny
            return res
        else:
            raise StopIteration  # Kończymy iterowanie
```

struktury danych – lista z iteratorem

```
class MojIterator:
    def __init__(self, pLista):
        self._kursor = pLista.glowa

    def __next__(self):              # Zwraca kolejny element z kolekcji obsługiwanej przez iterator
        if self._kursor !=None:      # (*)
            res= "["+ str(self._kursor.dane)+"]"
            self._kursor=self._kursor.nastepny
            return res
        else:
            raise StopIteration      # Kończymy iterowanie
```

struktury danych – lista z iteratorem

```
lista = Lista()   # Tworzymy pustą listę
for x in [1, 3, 5, 6, 12, 9]:
    lista.wstawNaKoniec(x)
print("Lista lista=", end=" ")
lista.wypisz()
print("Wywołujemy iterator poprzez użycie pętli 'for'")
for x in lista:
    print(x, end = " ")
```

Wyniki:

```
Lista lista= 1 3 5 6 12 9
Wywołujemy iterator poprzez użycie pętli 'for'
[1] [3] [5] [6] [12] [9]
```

ADT **kolejka priorytetowa:**

- przechowuje kolekcję wpisów
- każdy element jest parą (klucz, wartość)
- wartość – odpowiada ważności danego elementu

- podstawowe operacje:
  - insert(k,x) – dodaje element o kluczu *k* i wartości *x*
  - removeMin() – usuwa i zwraca element o najmniejszym kluczu

- ewentualnie / alternatywnie:
  - removeMax() – usuwa i zwraca element o największym kluczu

- dodatkowe metody
  - min() – zwraca, ael nie usuwa, element o najmniejszym kluczu
  - max() – alternatywnie
  - size() – wiadomo
  - isEmpty() – wiadomo

struktury danych – kolejka priorytetowa

operation	argument	return value	size	contents (unordered)	contents (ordered)
insert	P		1	P	P
insert	Q		2	P Q	P Q
insert	E		3	P Q E	E P Q
remove max		Q	2	P E	E P
insert	X		3	P E X	E P X
insert	A		4	P E X A	A E P X
insert	M		5	P E X A M	A E M P X
remove max		X	4	P E M A	A E M P
insert	P		5	P E M A P	A E M P P
insert	L		6	P E M A P L	A E L M P
insert	E		7	P E M A P L E	A E E L M
remove max		P	6	E E M A P L	A E E L M

A sequence of operations on a priority queue

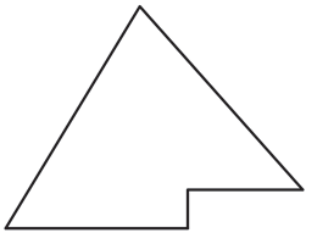
ADT **kolejka priorytetowa:**

- do implementacji kolejki priorytetowej można wykorzystać
  ang. *heap* – pol. kopiec / stertę

- sterta to:

  - nadgryziona od prawej strony piramidka
  - albo wypełniane od lewej drzewo

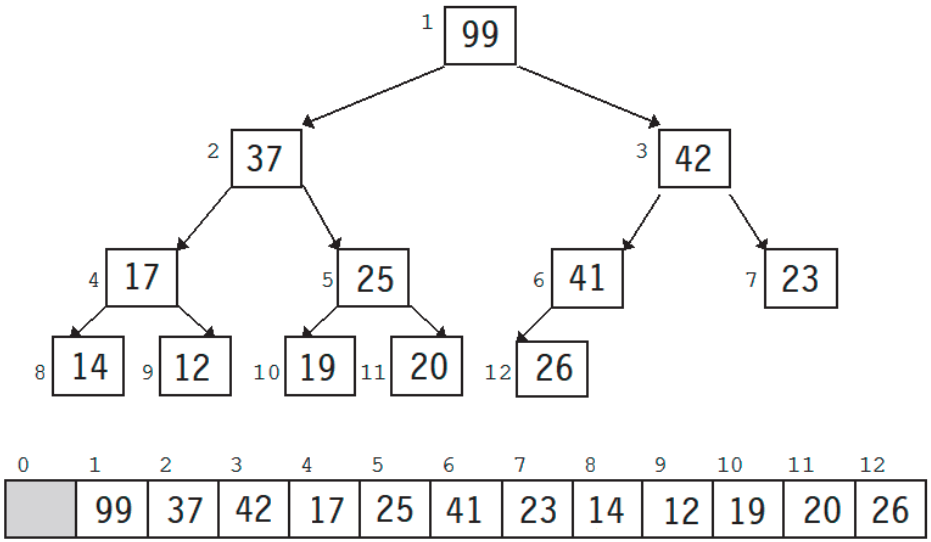
  - szczególne drzewo binarne
  - wartość każdego węzła jest większa od wartości węzłów potomnych – o ile istnieją



  - pod istniejące węzły podwieszamy max 2 elementy – od lewej
  - nowe elementy dodajemy na końcu
  - po czym przywracamy własność sterty/kopca

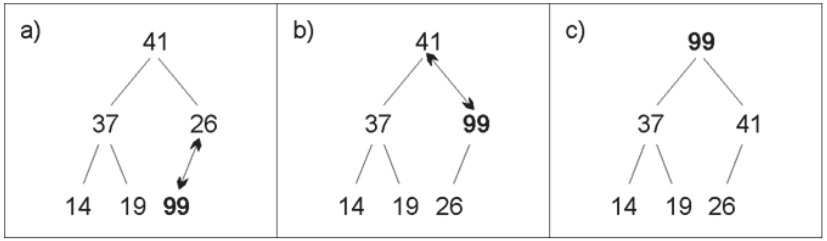
ADT **kolejka priorytetowa:**

- sterta w tablicy:
  - wierzchołek – indeks 1
  - lewy potomek *i*-tego węzła – indeks 2*i*
  - prawy potomek *i*-tego węzła – indeks 2*i*+1



ADT **kolejka priorytetowa:**

- dodawanie nowego elementu do sterty
  - z przywróceniem porządku



ADT **kolejka priorytetowa:**

- przykładowa implementacja sterty:

```
class Sterta:
    def __init__(self, nMax):
        self._Licznik = 0                          # Liczba elementów
        self._sterta=[0 for x in range(nMax+1)]    # Tworzymy statyczną tablicę
                                                   # o rozmiarze nMax+1

    def wstaw(self, x):
        self._Licznik=self._Licznik+1
        self._sterta[self._Licznik] = x
        self.DoGory()

    def obsluz(self):
        x=self._sterta[1]
        self._sterta[1]=self._sterta[self._Licznik]
        self._Licznik=self._Licznik-1
        self.NaDol()
        return x
```

ADT **kolejka priorytetowa:**

- przykładowa implementacja sterty:

```
def DoGory(self):
        tmp = self._sterta[self._Licznik]
        n = self._Licznik
        while ((n != 1) and (self._sterta[n // 2] <= tmp)): # Operator // realizuje
                                                            # dzielenie całkowite
            self._sterta[n] = self._sterta[n // 2]
            n = n // 2
        self._sterta[n] = tmp
```

ADT **kolejka priorytetowa:**

- przykładowa implementacja sterty:

```
def NaDol(self):
        i=1
        while(True):
            p=2*i                          # Lewy potomek węzła 'i' to (p), prawy to (p+1)
            if p>self._Licznik:
                break
            if p+1 <= self._Licznik:       # Prawy potomek niekoniecznie musi istnieć!
                if self._sterta[p] < self._sterta[p+1]:
                    p=p+1                  # Przesuwamy się do następnego
            if(self._sterta[i]>=self._sterta[p]):
                break
            tmp=self._sterta[p]            # Zamiana
            self._sterta[p]=self._sterta[i]
            self._sterta[i]=tmp
            i=p
```

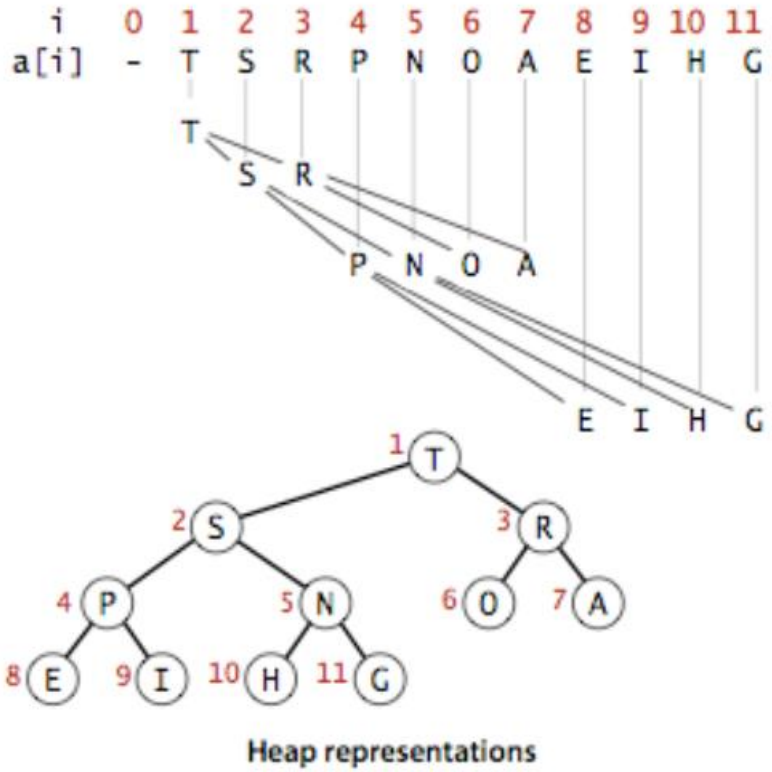
ADT **kolejka priorytetowa:**

- przykładowa implementacja sterty:

```
def wypisz(self, s):
    print(s)
    for i in range(1, (self._Licznik // 2) +1):
        print(" Wierzchołek: "+ str(self._sterta[i]), end=" ")
        print(" lewy potomek: " + str(self._sterta[2*i]), end=" ")
        # Prawy potomek niekoniecznie musi istnieć!
        if 2 * i +1 <= self._Licznik:
            print(" prawy potomek: " + str(self._sterta[2*i+1]) , end=" ")
        print()  # Przejście do nowej linii po serii instrukcji print() zawierających end=" "
```

ADT **kolejka priorytetowa - sterta:**

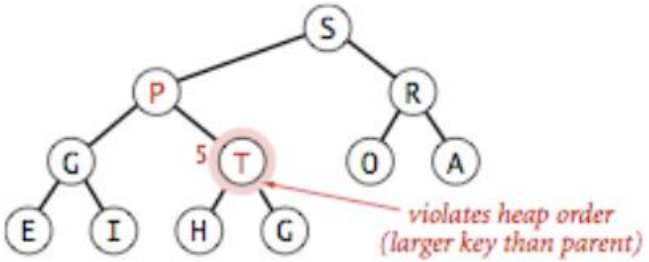
- sterta w tablicy:

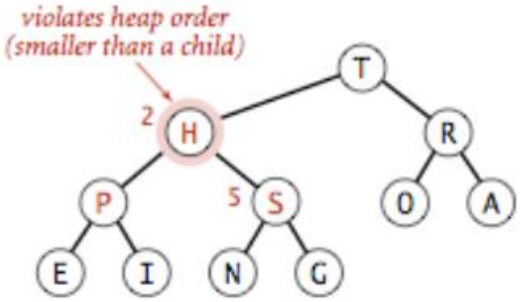


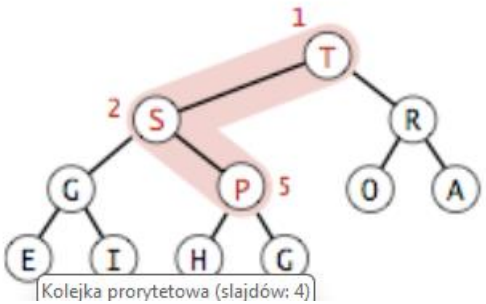
Heap representations

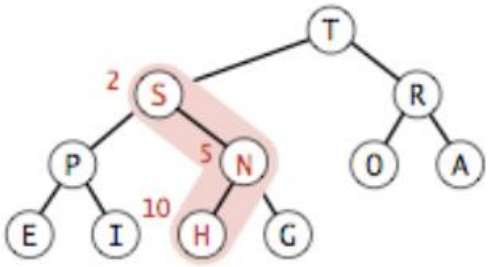
ADT **kolejka priorytetowa - sterta:**

- przywracanie porządku:
  - po lewej – doGóry
  - po prawej – naDół









rysunek: https://algs4.cs.princeton.edu/24pq/