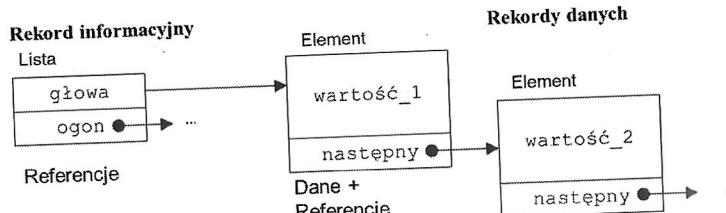


Do budowy listy jednokierunkowej używamy zazwyczaj dwóch rodzajów komórek pamięci:<sup>1</sup>

- rekordów informacyjnych zawierających właściwe dane,
- pewnej struktury informacyjnej pomagającej poruszać się pomiędzy rekordami danych (patrz rysunek 7.1).



RYSUNEK 7.1. Typy rekordów używanych podczas programowania list (1)

Rekord natury informacyjnej (na rysunku oznaczenie Lista) zawiera dwie referencje: do początku listy i do końca listy i stanowi zalążek klasy służącej do zarządzania listą.

Rekord o charakterze roboczym (oznaczenie Element) zawiera także jeden lub więcej atrybutów definiujących pożądaną zawartość informacyjną, np. liczbę, ciąg znaków, zbiór atrybutów różnych typów — także obiektów i referencję do następnego elementu listy.

Kosztem kilku bajtów pamięci zarezerwowanych dla rekordu informacyjnego uzyskujemy ciągły dostęp do bardzo istotnych odnośników ułatwiających zarządzanie w obrębie kolekcji. Ponadto ogromnie ułatwiamy operację podstawiania dołączenia nowego elementu na koniec listy (jeśli nie wstawiamy nowego elementu na koniec listy, to zawsze możemy przyłączyć go na jej początek, ale zawsze wówczas informację o kolejności przybywania danych!).

Pola głowa, ogon i następny są oczywiście referencjami (fakt wskazywania na wartości symbolizowany dalej przez strzałki), natomiast wartość często jest zbiorem atrybutów, danymi, którymi chcemy zarządzać. W przykładach znajdujących się w książce często operuję wartościami typu całkowitego, co nie umniejsza ogólnego charakteru wyniku. Ewentualne przeróbki tak uproszczonych algorytmów są raczej konieczne, nie są to zmiany o charakterze zasadniczym.

## Modelujemy listę jednokierunkową w Pythonie

Nasz pomysł na realizację listy w Pythonie jest następujący: jeżeli lista jest pusta, struktura informacyjna zawiera dwie wartości None. Pierwszy element listy jest złożony z grupy atrybutów (informacji do przechowania) oraz z referencji do drugiego elementu listy. Drugi element zawiera własne pola informacyjne i, oczywiście, referencję do trzeciego elementu listy itd. Miejsce zakończenia listy zaznaczamy także przy użyciu wartości specjalnej None. W celu ułatwienia pewnych czynności wbudujemy też do definicji klasy pole długość (a nóż się nam przyda... przekonasz się niedługo!).

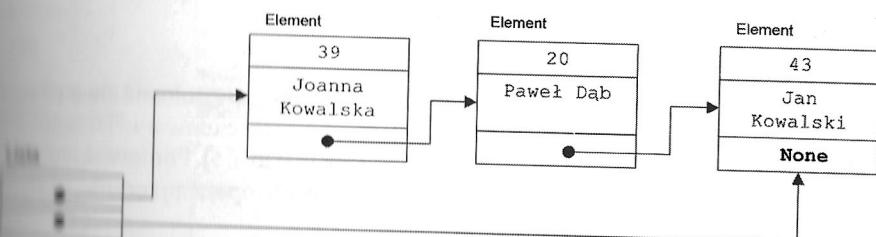
Realizacja pokazanej powyżej koncepcji może w Pythonie mieć taką postać (fragment listingu lista1.py, który będę systematycznie rozwijał — na FTP znajdziesz go oczywiście w wersji kompletnej):

```

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.głowa = None
        self.ogon = None
        self.długość=0
  
```

Przyjrzyjmy się teraz rysunku 7.2, na którym przedstawiono pewną listę jednokierunkową z trzech elementów: {39, "Joanna Kowalska"}, {20, "Paweł Dąb"}, {43, "Jan Kowalski"}.



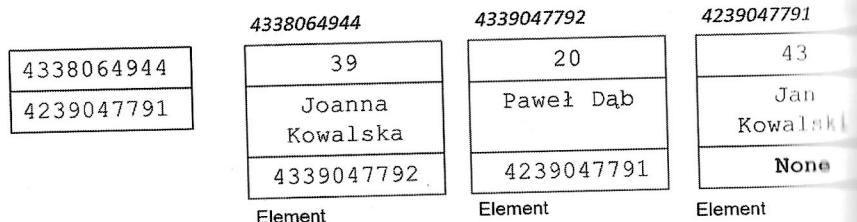
RYSUNEK 7.2. Typy rekordów używanych podczas programowania list (2)

Jeżeli komputer zapisuje w pamięci takie i podobne relacje? Można to wyjaśnić za pomocą poglądowego rysunku 7.3, który jest dokładnym odbiciem swojego poprzednika, z tą różnicą, że — w miejsce strzałek symbolizujących wskazywanie — znajdują się konkretne wartości liczbowe adresów komórek pamięci.

Linia umieszczona nad rekordem jest przykładowym *adresem logicznym* zarządzanym przez środowisko uruchomieniowe Pythona, przydzielonym podczas tworzenia obiektów w pamięci (wartości dla None nie podałem celowo, bo jest to pozbacone sensu). Pamiętaj, że adres ten nie jest adresem fizycznym.

<sup>1</sup> Większość akademickich opisów struktur listowych nie wyodrębnia rekordu informacyjnego (nie jest on bezpośrednio elementem struktury danych) — jest to oczywiście błędny opis.

<sup>2</sup> Wielkość informacji niezbędnej do realizacji potrzeb adresowych jest ukryta przed oczyma czytelnika, można powiedzieć, że kilka bajtów wystarczy do tego, by



RYSUNEK 7.3. Realizacja fizyczna listy jednokierunkowej

komputera i nie można go użyć bezpośrednio w programie, rysunek jest tylko ilustracją pewnej hipotetycznej realizacji na fizycznym komputerze.

Wróćmy jeszcze do analizy rekordów składających się na listę. Pole głowa struktury informacyjnej wskazuje na pierwszy element listy, zawierający dane {"Joanna Kowalska"} oraz referencję do następnego elementu listy. Pole ogon struktury informacyjnej wskazuje na ostatni element listy, zawierający dane {43, "Jan Kowalski"}. W tej komórce referencja na ostatni element jest pusta i wiemy, że jest to ostatni element listy. Pola głowa i ogon posłużą nam do wygodnego przeglądania elementów listy i dołączania nowych.

Jak można ręcznie dołożyć element np. na koniec listy? (Ręcznie, gdyż docelowo powinniśmy dorobić się eleganckich metod realizujących takie zadanie). Na przykład w następujący sposób:

```

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

Posiadając już pewne dane zapisane na liście, możemy pokusić się o przetestowanie procedury przeglądającej elementy listy, np. w poszukiwaniu jakiegoś elementu (w przykładzie będzie to rekord zawierający wartość 5). Ponieważ jesteśmy nieco zaawansowanymi programistami, to końcowy kod, to w dalszym ciągu zrealizujemy tę operację ręcznie, używając tej samej pętli while:

```

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")
    
```

W książce nie zawsze będę proponował gotowy kod Pythona w celu wyjaśnienia zasady działania algorytmu — wykład może być przeplatany także „pseudokodem”; kryterium wyboru będzie czytelność procedur. Oczywiście, jeśli nawet prezentacja algorytmu zostanie wykonana w pseudokodzie, omawiany przykładowy program na FTP będzie kompletny i gotowy do uruchomienia!

Off, nie było łatwo, ale przekonaliśmy się, wykonując ćwiczenie, że ten kod naprawdę działa — przetestuj go np. w edytorze PyCharm lub z linii poleceń!

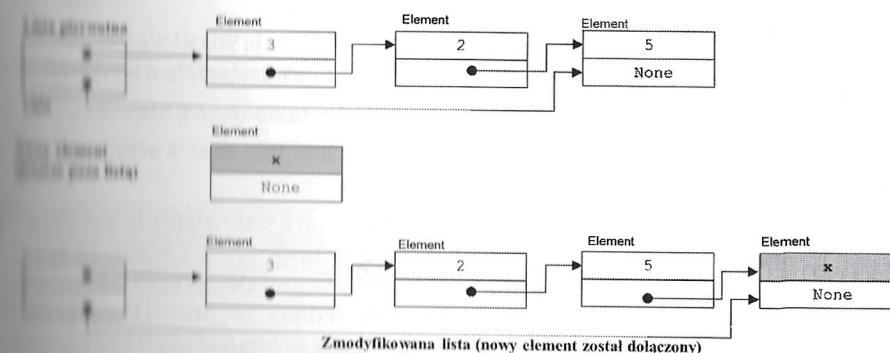
### Dokładamy pierwsze metody użytkowe

Pokażana wcześniej realizacja *struktury danych* listy jednokierunkowej w Pythonie powinna zostać wzbogacona o metody użytkowe, które ułatwią wszystkie typowe operacje, jakie są potrzebne w praktyce: wyświetlanie zawartości, dodawanie i usuwanie elementów, przeszukiwanie...

Chwilowo widzieliśmy tylko, że klasa Lista zawiera dwie zmienne referencyjne, służące do zapamiętania początku i końca listy. Teraz czas na metody!

Zacznijmy od prostej metody *dokładającej nowy element na koniec listy*. W tym celu przejdź do listingu o nazwie lista2.py, który jest kontynuacją przykładu z pliku lista1.py.

Popatrzmy, jak proste jest dokładanie nowego elementu do listy na jej koniec (rysunek 7.4).



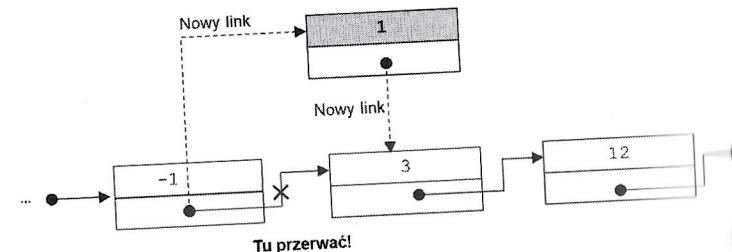
RYSUNEK 7.4. Dokładanie nowego elementu listy na jej koniec

„Technicznie” dołożyć element na koniec listy? Cóż, należy go odszukać i odpowiednio podpiąć referencję ostatniego elementu listy do nowo utworzonego elementu. Ale zaraz, po co tu szukać końca listy, jeśli tak naprawdę dysponujemy odniesieniem do niego w polu ogon naszej listy! Teraz wszystko staje się proste, bo metoda wstawNaKoniec() zajmuje naprawdę kilka linijek (zwróć uwagę, że kontynuacja definicji klasy Lista i na listingu wciecie kodu jest większe, niż widoczne na rysunku części jej definicji!):

we właściwym, ustalonym przez nas porządku — np. sortowane od razu w kierunku wartości niemalejących.

Dokładanie elementów na koniec listy pozwala zapamiętywać kolejność wadzania danych. (Oczywiście alternatywnie możliwe jest dokładanie rekordu przed pierwszy element listy).

O wiele bardziej złożona jest funkcja dołączająca nowy element w takie miejsce, aby sekwencja danych była widziana jako **lista posortowana** (tutaj: w kolejności wartości niemalejących). Ideę przedstawia rysunek 7.5, gdzie możemy zobaczyć sposób dołączania liczb 1 do już istniejącej listy złożonej z elementów 1, 2, 3, 4, 5.



#### **RYSUNEK 7.5.** Dotaczanie elementu listy z sortowaniem

Uogólniając wywód, można powiedzieć, że

- Każdy nowy element może zostać wstawiony na początek (a), koniec (b) lub pośrodku (c) listy, jak również gdzieś w jej środku (c).
  - W każdym z tych przypadków w istniejącej liście trzeba znaleźć miejsce do wstawienia, tzn. zapamiętać dwa wskazniki: element, przed którym ma być wstawiona nowa komórka, i element, za którym mamy to zrobić. Do zapamiętania tych istotnych informacji niech nam posłużą zmienne nazwane przez nas (swego rodzaju współzewnętrne).

Następnie, gdy dowiemy się, gdzie jesteśmy, możemy wstawić nowy element. Sposób, w jaki to zrobimy, zależy oczywiście od miejsca wstawienia i czy lista przypadkiem nie jest jeszcze pusta. Krótko mówiąc: realizacja jest dość złożona. Pewne skomplikowanie metody `wstawSort()` wynika z tego, że w niej poszukiwania miejsca wstawienia z dołączeniem elementu nie dobrze można było rozbić na osobne funkcje, jednak zrobione w obecnej wersji.

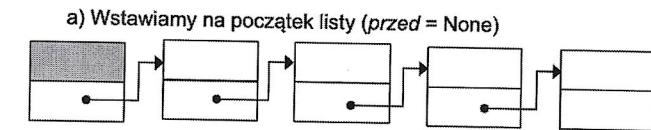
Te trzy przypadki współrzędnych nowego elementu w liście zostały przedstawione na rysunku 7.6 (zakładamy, że lista już coś zawiera z kilku elementów).

W każdym z tych przypadków zmieni się sposób dołączenia elementów. Przeanalizuj jedną z możliwych реализациj metodę `wstawSort()`, która

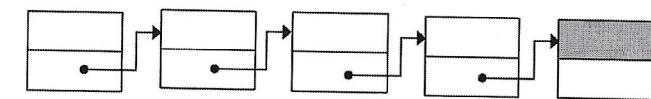
## **RYSUNEK 7.6.**

### Witawianie nowego elementu do listy

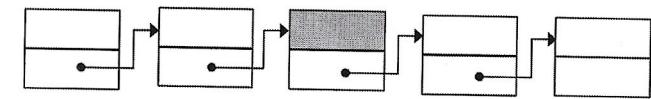
- analiza
- przypadków



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



c) Wstawiamy w środek listy ( $przed \neq \text{None}$ ,  $po \neq \text{None}$ )



```

def wstawSort(self, x):
    nowy=Element(x)
    self.dlugosc = self.dlugosc + 1

# Poszukiwanie właściwej pozycji na wstawienie elementu:
if (self.glowa==None): # Lista pusta
    self.glowa=nowy
    ogon=nowy
    return # Pozwala na szybkie opuszczenie funkcji

# Poszukiwanie miejsca na wstawienie:
szukamy=True # Stan poszukiwania miejsca na wstawienie
przed=None # 'przed' i 'po' określają miejsce wstawiania nowego elementu
self.glowa
while ( szukamy and (po!=None) ):
    if ( po.dane >= x ): # Kryterium sortowania (*)
        szukamy=False # Znaleźliśmy właściwe miejsce!
    else:
        przed=po
        po=po.nastepny

# Analizując wartości zapamiętane w 'przed' i 'po'
if (przed == None): # Na początek listy
    self.glowa=nowy
    nowy.nastepny=po

if (po == None): # Na koniec listy
    przed.nastepny = nowy
    self.ogon=nowy # Nowy koniec listy!
else: # Wstawiamy gdzieś w środku, „rozpinając” łańcuszek danych
    przed.nastepny=nowy
    nowy.nastepny=po

```

Algorytm nie jest zbyt złożona, ale jeśli masz kłopot z oceną jego funkcjonalności uruchom skrypt *listaMain.py*, zmodyfikuj ewentualnie jego treść oraz sprawdź wynikom jego działania (wybrane komunikaty są opisane w linii za znakiem #):

# Nie znaleziono poszukiwanego elementu 2  
sortowaniem  
poszukana, wstawiam kolejno: 0 1 3 5 6 7 8 )

```

m1.wstawSort(3) # 3
m1.wstawSort(5) # 3 5
m1.wstawSort(7) # 3 5 7
m1.wstawSort(0) # 0 3 5 7
m1.wstawSort(1) # 0 1 3 5 7
m1.wstawSort(6) # 0 1 3 5 6 7
m1.wstawSort(7) # 0 1 3 5 6 7 7
print("Lista m1=", end=" ")
m1.wypisz()      # Lista q= 0 1 3 5 6 7 7

```

Możesz także zmodyfikować kod klasy `Lista`, dokładając tu i ówdzie instrukcje `print()` z komunikatami pseudodebugowania (oczywiście możesz włączyć przydzielony tryb debugowania w PyCharmie!).

## Usuwanie danych z listy

Omawianie listy jednokierunkowej będziemy kontynuowali na przykładzie klasy `Lista`, którą rozbudujemy o możliwość usuwania elementów. Ponieważ język Python posiada wbudowane możliwości zarządzania obiektami w pamięci, w przykładowym kodzie nie znajdziesz komendy `delete`, znanej z języków C++ i C#. Jest to spore ułatwienie dla programistów!

Skończmy jednak z dygresjami na temat różnic występujących pomiędzy językami programowania, gdyż są one nieuniknione i należy je zwyczajnie poznać, aby móc skutecznie pisać programy komputerowe. Zrealizujmy dalej dwa warianty operacji usuwania elementów z listy:

- usunięcie elementu z przodu (głowy) listy — metoda `UsunPierwszy()`
- usunięcie elementu `x` ze środka listy — metoda `UsunWybrany(x)`.

Pierwsza metoda jest wręcz trywialna, gdyż wystarczy przestawić adres szanowany w zmiennej `glowa` o jeden do przodu; potraktuj ją zatem jako rozgrzewkę.

```

def UsunPierwszy(self):          # Usuń pierwszy element z listy
    if self.glowa != None:
        self.glowa = self.glowa.nastepny
        self.dlugosc = self.dlugosc - 1 # Skracamy parametr opisujący długość listy o 1

```

## Ćwiczenie 7.2

Zmień nieco logikę klasy `Lista` i napisz metodę realizującą usuwanie elementu z tytułu (ogona) listy — metoda `UsunOstatni()` z użyciem zwykłego przeszukiwania rekordów, bez używania zawartości pola `ogon`.

### Usunięcie wybranego lub ostatniego elementu z listy jest już bardziej złożone

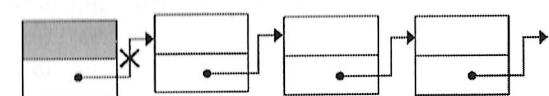
Musimy w tym celu uwzględnić kilka przypadków elementarnych: lista pusta, zawierająca jeden element, lista, w której jest więcej niż jeden element. Zanim przystąpimy do analizy gotowego przykładowego kodu, spróbujmy przeanalizować wizualnie, jakie mogą wystąpić elementarne przypadki usuwania określonego elementu listy. W tym celu popatrz na rysunek 7.7.

**RYSUNEK 7.7.**  
Analiza elementu usuwania z listy

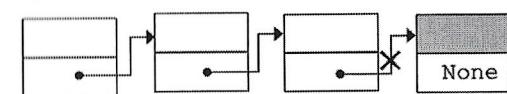
a) Lista ma długość 1



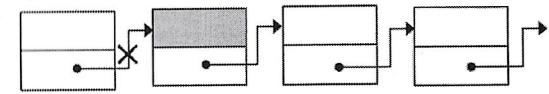
b) Lista ma długość >1, usuwamy z przodu



c) Lista ma długość >1, usuwamy z końca



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



Fay robieli to przypomina...? Tak, podobnie jak w przypadku *wstawiania połączonego i posortowania*, także tutaj w celu wykonania operacji usuwania (technicznie: przerwania linków oznaczonych symbolem `X`) należy odszukać *przedostatni* element listy, aby móc odpowiednio zmodyfikować wskaźnik *ogon* struktury informacyjnej. Znajomość przedostatniego elementu listy umożliwi nam łatwe usunięcie jej ostatniego elementu.

Próbowajmy go zatem odnaleźć za pomocą nowej metody klasy `Lista`:

```

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

```

Zauważ, że w pojedynczej instrukcji `return` są zwracane aż trzy wartości naraz! Jak już wspomniałem w rozdziale 5., taka wartość jest tzw. *tuplą*. Jej elementy (tutaj: trzy) należy „rozpakować” do odrębnych zmiennych, np. w taki sposób:

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

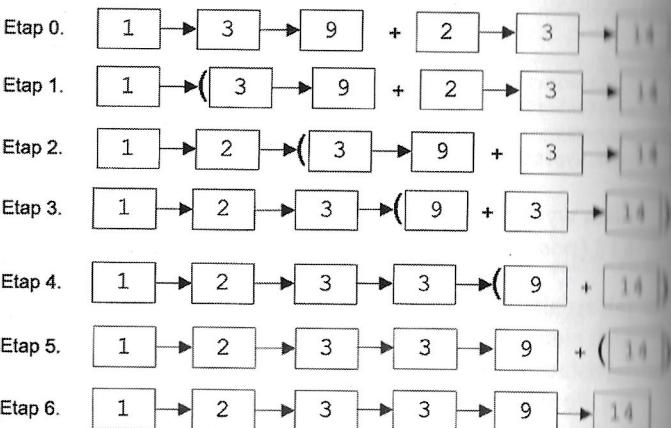
Zwróc uwagę, że w podanych przykładach dość swobodnie definiujemy w miarę potrzeb obiekty lokalne (np. w pętlach `while`, iteracjach, instrukcjach warunkowych). Mogą to być zarówno typy podstawowe, jak i referencje lub nowe obiekty dość złożonych klas. Czy nie jest to jednak marnotrawstwo pamięci? Odpowiem nieco przewrotnie: marnotrawstwo pamięci na pewno nigdy nie wychodzi na dobre i jeden język nie zapewnia tutaj wystarczających mechanizmów obronnych, które

Czy jest to najlepsza metoda? Na pewno nie, chociażby z uwagi na niepotrzebne dublowanie danych (tworzymy przecież ponownie listę i powielamy operacje wyszukania i sortowania w celu utworzenia kolekcji wynikowej). Co z tego? Python zadba o usunięcie zbędnych obiektów, jeśli nasz program będzie działać koszmarnie wolno, na dodatek na chwilę zapchamy pamięć, a możliwe, że program „zamrozi” nam na moment system operacyjny podczas porządkowania naszych śmieciowych obiektów (to może się zdarzyć przy przetwarzaniu dużych list). To nie żart — Python nie jest demonem szybkości w zakresie przetwarzania dużej liczby danych obiektowych, to środowisko nie funkcjonuje tak jak C++, gdzie wiele bazowych operacji jest wykonywanych na blokach pamięci.

Ideałem byłoby posiadanie metody, która, wykorzystując to, że listy są już porównywane<sup>4</sup>, dokonałaby ich zespolenia ze sobą (tzw. fuzji), używając wyłącznie istniejących komórek pamięci, bez tworzenia nowych. Inaczej mówiąc, będziemy zmniejszać do manipulowania wyłącznie adresami (referencjami), i to jest jedyne narzędzie, jakie dostaniemy do ręki!

Na rysunku 7.8 możemy przykładowo prześledzić, jak powinna być wykonywana fuzja pewnych dwóch list:  $x=(1, 3, 9)$  i  $y=(2, 3, 14)$ , tak aby w jednej z nich znalazły się wszystkie elementy  $x$  i  $y$  — oczywiście posortowane (w naszym przykładzie niemalejąco).

**RYSUNEK 7.8.**  
Fuzja list  
na przykładzie



Najmniejszym z dwóch pierwszych elementów list jest 1 i on też będzie stanowić początek nowej listy. Następnikiem tego elementu będzie fuzja dwóch list  $x' = (3, 9)$  i  $y' = (2, 3, 14)$ . Jak dokonać fuzji list  $x'$  i  $y'$ ? Dokładnie tak samo: bierzemy element 2, który jest najmniejszy z dwóch pierwszych elementów list  $x'$  i  $y'$ ... Można tak rekurencyjnie kontynuować aż do momentu, gdy natrafimy na przypadek ekstremalny: jeśli jedna z list jest pusta, to fuzją ich obu będzie oczywiście ta druga lista. Na tej zasadzie jest skonstruowana procedura fuzja, która zwróci w liście ob1 sumę elementów list ob1 i ob2.

<sup>4</sup> Zakładamy tym samym użycie metody `wstawSort()` podczas ich tworzenia.

Fuzję list wykonamy w dwóch etapach.

Pierwszym krokiem jest przygotowanie prostej funkcji `sortuj()`, która — otrzymując dwie posortowane listy wskazywane przez ich elementy początkowe a i b — zwróci jako wynik listę, której będzie ich fuzją. Rekurencyjny zapis tego procesu jest bardzo prosty i dobrze nadaje się do rozwiązywania problemów listowych w takich językach jak Lisp lub Prolog.

`fuzja(list): #funkcja (nie metoda klasy) operująca na liście obiektów klasy 'Element'`  
 `#funkcja zwracająca z obiektów klasy 'Element'`

```
def sortuj(a, b):
    if a == None:
        return b
    if b == None:
        return a
    if a.dane <= b.dane:      # Porównywanie wartości – tutaj zmień ew. kryterium lub wbuduj
                            # funkcję porównującą
        a.nastepny = sortuj(a.nastepny, b)
        return a
    else:
        b.nastepny = sortuj(b.nastepny, a)
        return b
```

Wykorzystując już funkcję pomocniczą `sortuj()`, możemy zastosować ją do właściwej funkcji `fuzja(x1, y1)`, która zrealizuje następujący algorytm:

- Dokonaj złączenia list  $x_1$  i  $y_1$  (sekwencje wskazywane przez pola głowa tych list).
- Zwróć nowy obiekt klasy `Lista`, którego pole głowa będzie wskazywało na złączoną kolekcję elementów, i dokonaj odpowiedniej aktualizacji wartości `dlugosc` (suma dwóch długości) i referencji `ogon`.

Może to zbyt skomplikowane zadanie, ale warto rozpisać precyzyjnie przypadki, aby nie pomylić się w ustaleniu pola `ogon`:

- jeśli listy wejściowe są puste, to głowa i ogon zostaną ustawione przez konstruktor na `None`.
- Ponieważ sortujemy *wstępującco* (kryterium porównywania jest  $\leq$ ), to pole `ogon` powinno wskazywać na prawy skrajny element sekwencji. Z uwagi na występowanie dwóch „ogonów” w listach pierwotnych, należy uwzględnić tylko ten faktycznie ostatni, za którym już nie występują żadne rekordy danych.

Popatrzmy, jak można zrealizować taką funkcję:

```
def fuzja(x1, y1): #Zakładamy, że x1 i y1 są obiektami klasy 'Lista'
    nowalista = Lista()
    nowalista.dlugosc = x1.dlugosc + y1.dlugosc
    nowalista.glowa = sortuj(x1.glowa, y1.glowa) #Sekwencja złączonych elementów
                                                    #danych
                                                    #(obiekty klasy 'Element')
    if x1.glowa == None:
        nowalista.ogon = y1.ogon
```