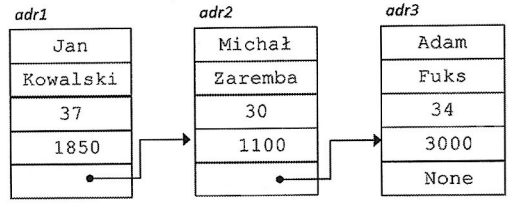
Tak obiektów tworzących listę danych będziemy dysponować listami referencji do elementów listy danych, ale te listy referencji będą tworzone na bieżąco od nowa w wersji posortowanej. List posortowanych referencji będzie tyle, ile sobie zażyczymy kryteriów sortowania. Dzięki takiemu podejściu oszczędzimy miejsce w pamięci, gdyż pierwotne dane będą zapisywane tylko raz, natomiast obok będą przechowywane struktury pomocnicze, które pozwolą nam na łatwe zarządzanie zapisanymi informacjami. Na zasadzie analogii do baz danych można przyjąć, że będziemy posiadali tabelę danych i tabele indeksów do danych.

Jak nietrudno się domyślić, jeśli nie zamierzamy sortować listy danych (a jednocześnie chcemy mieć dostęp do danych posortowanych!), to podczas wstawiania nowego adresu do którejś z list referencji musimy dokonać jej przesortowania. Zadanie jest zbliżone do tego, które wykonywała funkcja `wstawSort()`, z tą tylko różnicą, że dostęp do danych nie odbywa się w sposób bezpośredni.

Co ważne, **podczas sortowania list referencji pierwotne dane nie są w ogóle przesuwane** — przemieszczaniu w listach będą ulegały wyłącznie same referencje! Na tym etapie ma prawo to wszystko brzmieć dość enigmatycznie, pora zatem na jakiś konkretny przykład.

Popatrzmy w tym celu na rysunek 7.9 pokazujący listę trzech elementów — rekordów danych prostej bazy danych o pracownikach pewnego przedsiębiorstwa.



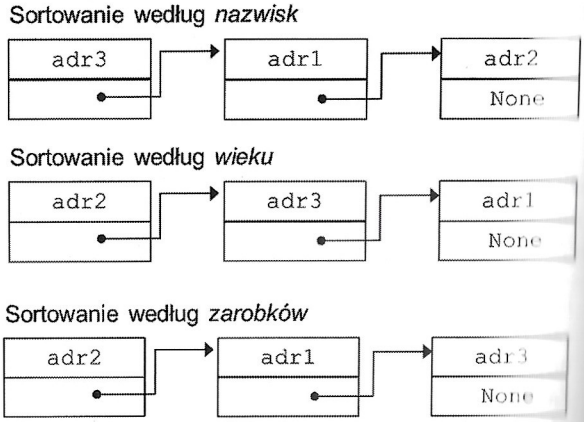
RYSUNEK 7.9. Prosta baza danych o pracownikach

Nasz przykład pokazuje listę zbudowaną z kilku rekordów, które stanowią zaczątek miniaturowej bazy danych o pracownikach pewnego przedsiębiorstwa. Przyjmijmy dla uproszczenia, że jedyne istotne informacje, które chcemy zapamiętać, to: imię, nazwisko, wiek i oczywiście zarobki. Na rysunku są zaznaczone symbolicznie adresy rekordów: `adr1`, `adr2` i `adr3` (są to rzeczywiście adresy symboliczne, a nie fizyczne adresy obiektów w pamięci). Można sobie teraz wyobrazić nieco inne listy, które będą przechowywały wyłącznie referencje do tej pierwotnej bazy danych, kolejno: sortowanie według nazwiska, wieku i zarobków (rysunek 7.10).

W prawdziwych programach zysk związany z rozdzieleniem „surowych" informacji od list indeksów będzie ewidentny, gdy obiekty informacyjne będą zawierały wiele pól i dużo danych (np. zdjęcia, załączniki binarne), a same listy indeksów pozostaną „lekkie", utrzymując wyłącznie listy adresów.

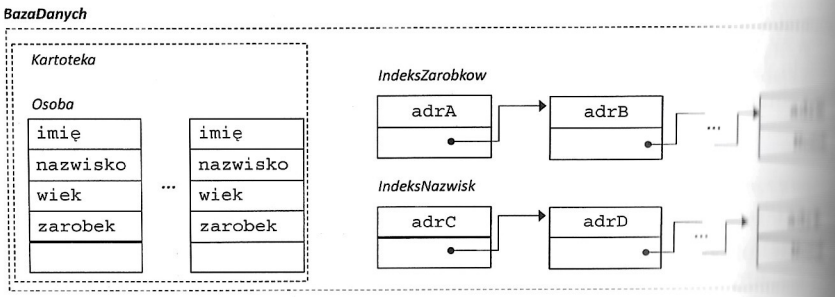
Poniżej jest przedstawiona nowa wersja naszej klasy listowej, uwzględniająca już wcześniej przedstawione propozycje. Ponieważ naszą intencją jest posortowanie bazy danych osobowych, odpowiednio przemianowałem też nazwy klas, wprowadzając następujące:

RYSUNEK 7.10. Listy referencji realizujące sortowanie według wybranych kryteriów



- `BazaDanych` — główna baza danych zawierająca obiekt `Kartoteka` (dane nie posortowane) i klasy pomocnicze `IndeksZarobkow` i `IndeksNazwisk`, służące do przeglądania list w wersji posortowanej według konkretnych kryteriów
- Klasa `Kartoteka` — zawiera listę nieposortowanych obiektów typu Osoba

Całość tworzy propozycję (zapewne jedną z wielu możliwych) realizacji prostej bazy danych (rysunek 7.11).



RYSUNEK 7.11. Model realizacji obiektowej bazy danych

Aby umożliwić sensowną prezentację w postaci przykładowego programu temu uproszczeniu uległa struktura danych zawierająca informacje o pracownikach ograniczymy się tylko do nazwiska i zarobków. (Rozbudowa tych struktur danych nie wniosłaby koncepcyjnie nic nowego, natomiast zagmatwałaby i tak pokaźny objętościowo listing).

Podobnie jak poprzednio, struktury danych potrzebne do realizacji naszego projektu zapiszemy w folderze *MojeTypy* i oddzielimy od głównego pliku sterującego.

- Katalog podrzędny *MojeTypy* — pliki: *BazaDanych.py*, *Kartoteka.py*, ...
- Katalog bieżący (zawierający katalog *MojeTypy*) — plik: *BazaDanychM...*