Zmienne dynamiczne

- **pamięć dynamiczna**: obszar pamięci, który program może wykorzystywać w sposób jawny, umieszczając tam dane różnych typów
  $\Rightarrow$ **przydzielanie** (alokacja) pamięci
  $\Rightarrow$ **zwalnianie** (dealokacja) pamięci

- zmienne umieszczane w pamięci dynamicznej nazywamy **zmiennymi dynamicznymi**; różnią się one od znanych nam zmiennych statycznych:
  - są **anonimowe**, tzn. nie mają nazwy; odwołania do nich są możliwe za pomocą wskaźników
  - muszą być **tworzone jawnie** procedurą `NEW`
  - zmienne dynamiczne mają **nieograniczony zakres**, tzn. można odwoływać się do nich z dowolnego miejsca programu; są w pewnym sensie globalne (nie jest to zupełnie ścisłe ponieważ globalne lub lokalne mogą być nazwy, a te zmienne nie mają nazw)
  - zmienne dynamiczne **istnieją dowolnie długo**, tzn. do chwili jawnego ich skasowania procedurą `DISPOSE`
  - jednak tak samo jak dla zmiennych statycznych obowiązuje ścisłe przestrzeganie **zgodności typów**

Wskaźniki

- **wskaźniki** są specjalnym typem danych, służącym wyłącznie do posługiwania się zmiennymi dynamicznymi:
  - operacje możliwe na wskaźnikach: przypisanie `:=`, porównanie `=`, i dostęp do zmiennej dynamicznej `^`
  - mogą być parametrami i wartością funkcji
  - specjalna stała wskaźnikowa NIL
  - zmienne wskaźnikowe są statyczne

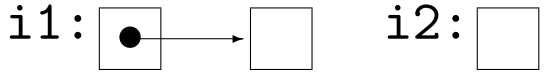
Przykład

```
TYPE WskaznikInt = ^INTEGER;
VAR i1, i2: WskaznikInt;
{...}
NEW(i1);                                    {tworzymy zmienna dynamiczna }
i1^ := 5;                                   {nadajemy jej wartosc }
WRITELN('Mamy zapamietana wartosc: ',i1^);
NEW(i2);                                    {to samo dla drugiej zmiennej}
i2^ := 5;
WRITELN('Mamy zapamietana wartosc: ',i2^);
IF i1 = i2
  THEN WRITELN('Wskazniki sa takie same.')
  ELSE WRITELN('Wskazniki sa rozne.');
i2 := i1;                                   {utozsamiamy wskazniki }
IF i1 = i2
  THEN WRITELN('Wskazniki sa takie same.')
  ELSE WRITELN('Wskazniki sa rozne.');
i2^ := 7;                                   {przypis.wart.jednej zm.dynam}
IF i1^ = i2^
  THEN WRITELN('Wartosci pozostaja takie same.')
  ELSE WRITELN('Wartosci sa rozne.')
```

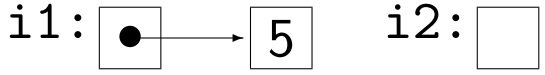
Graficzna ilustracja wskaźników i zmiennych dynamicznych

```
VAR i1,i2:  ^INTEGER;
```

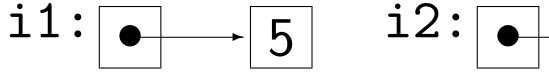

```
NEW(i1);
```



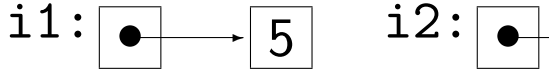

```
i1^ := 5;
```



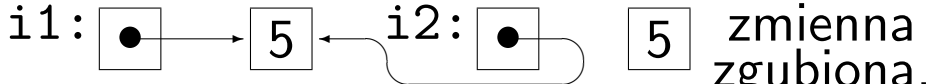

```
NEW(i2);
```



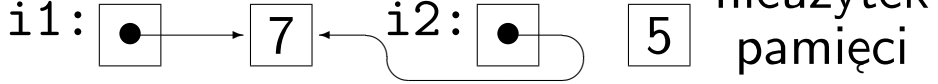

```
i2^ := 5;
```




```
i2 := i1;
```

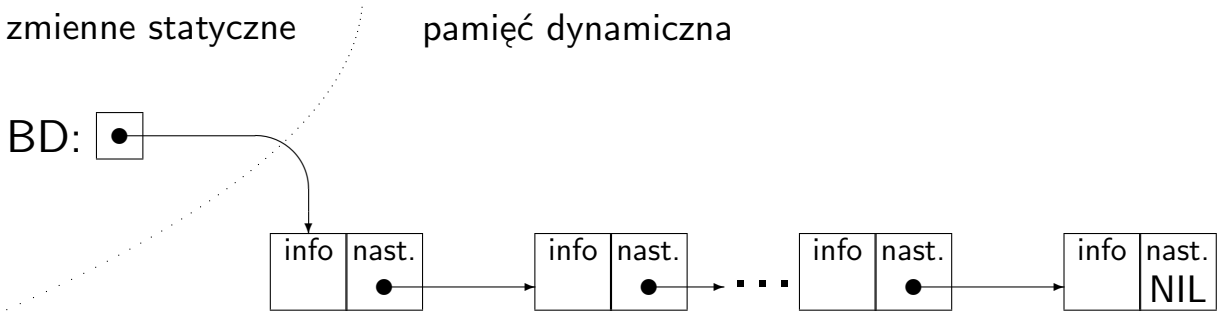



```
i2^ := 7;
```



Lista — regularna struktura dynamiczna

```
TYPE T_Lista = ^T_Element;
     T_Element = RECORD
                   info: T_info;
                   nastepny: T_Lista;
                 END;
```



- dynamiczna struktura danych jest regularną strukturą łączącą elementy treści z elementami konstrukcyjnymi (wskaźnikami)
- niezbędna jest co najmniej jedna zmienna statyczna zapewniająca dostęp do struktury dynamicznej
- specjalna wartość wskaźnikowa NIL jest przydatna do oznakowania pustych zakończeń struktury dynamicznej

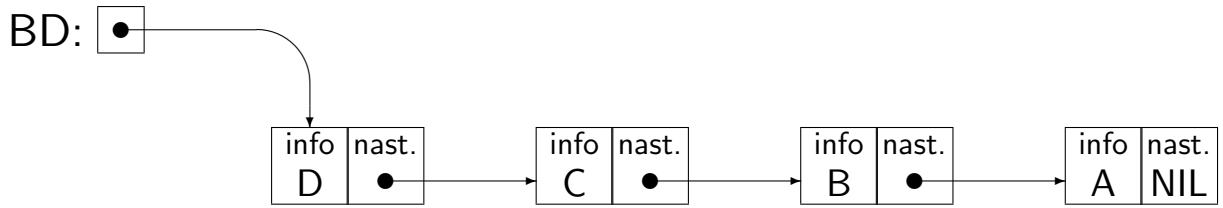
Listy — tworzenie

```
VAR BD, pom: T_Lista;

BD := NIL;                              {zainicjuj pusta liste}
WHILE NOT EOF(plik) DO                  {czytaj plik do konca }
  BEGIN
    NEW(pom);                           {utworz elem.w pamieci}
    WczytajElement(plik,pomˆ);          {wczytaj jego dane    }
    pomˆ.nastepny := NIL;               {to na wszelki wypadek}
    DodajElement(BD,pom);               {wlacz elem.do listy  }
  END;
```

- trzy etapy w procesie konstrukcji listy:
  - stworzenie nowej zmiennej dynamicznej
  - wpisanie do niej właściwej treści
  - utworzenie właściwych wskaźników konstrukcyjnych dla włączenia nowego elementu do reszty listy
- należy pamiętać o wartości znacznikowej NIL

Listy — tworzenie (2)

```
PROCEDURE DodajElement1(VAR Lista: T_Lista;
                        Element  : T_Lista);
{wersja 1: na poczatek listy}
BEGIN
  Element^.nastepny := Lista;
  Lista := Element;
END; {DodajElement1}


BD := NIL;
NEW(pom);   pom^.info := 'A';   DodajElement1(BD, pom);
NEW(pom);   pom^.info := 'B';   DodajElement1(BD, pom);
NEW(pom);   pom^.info := 'C';   DodajElement1(BD, pom);
NEW(pom);   pom^.info := 'D';   DodajElement1(BD, pom);
```



Listy — tworzenie (3)

```
PROCEDURE DodajElement2(VAR Lista: T_Lista;
                        Element  : T_Lista);
{wersja 2: na koniec listy, realizacja rekurencyjna}
BEGIN
  IF Lista = NIL
    THEN Lista := Element
    ELSE DodajElement2(Lista^.nastepny, Element);
  Element^.nastepny := NIL;   {na pewno poprawne,choc moze niekonieczne}
END; {DodajElement2}
```

- zauważmy, że sprawdzenie czy 'Lista = NIL' dotyczy nie tyle przypadku pustej listy (początkowo), ale ogólnie przypadku końca listy; a więc każdy nowy element będzie wpisywany w tę pustą wartość NIL

- zauważmy również, że ta niewinnie wyglądająca procedura wykonuje przeszukiwanie całej istniejącej listy, w dodatku szeregiem tylu wywołań rekurencyjnych, ile elementów ma lista

Listy — tworzenie (4)

```
PROCEDURE DodajElement3(VAR Lista: T_Lista;
                        Element  : T_Lista);
{wersja 3: na koniec listy, realizacja iteracyjna}
VAR ostatni: T_Lista;
BEGIN
  IF Lista = NIL
  THEN Lista := Element
  ELSE
  BEGIN
    ostatni := Lista;          {wiemy ze Lista<>NIL}
    WHILE ostatni^.nastepny <> NIL DO
      ostatni := ostatni^.nastepny
    ostatni^.nastepny := Element;
  END;
  Element^.nastepny := NIL;   {na pewno poprawne,choc moze niekonieczne}
END; {DodajElement3}
```

- pętla WHILE znajduje ostatni element listy, dlaczego nie można:
  `WHILE ostatni <> NIL DO ostatni := ostatni^.nastepny`

Listy — tworzenie (5)

- dodawanie do listy elementu za wskazanym elementem:

```
element^.nastepny := poprzedni^.nastepny;
poprzedni^.nastepny := element;
```

- stąd pomysł kolejnej wersji procedury dodawania elementu:

```
PROCEDURE DodajElement4(VAR PoczatekListy,
                            KoniecListy  : T_Lista;
                        Element          : T_Lista);
{wersja 4: na koniec listy ze wskazn.ostat.elemen.}
BEGIN
  IF PoczatekListy = NIL
  THEN BEGIN
         PoczatekListy := Element;
         KoniecListy := Element
       END
  ELSE BEGIN
         KoniecListy^.nastepny := Element;
         KoniecListy := Element;
       END;
END; {DodajElement4}
```

- posługiwanie się listą ze wskaźnikiem ostatniego elementu:

```
VAR BD_pocz, BD_konc: T_Lista; { dwa wskazniki listy }
{...}
BD_pocz := NIL;
BD_konc := NIL;                { niekonieczne }
```

Listy — wyszukiwanie elementu

```
FUNCTION PorownajElement(e1, e2: T_Element): CHAR; {...}
{niech zwraca jeden ze znakow: '<', '=', '>' }

{wersja 1 - zla; moze nie zatrzymac sie na koncu}
pom := BD;
WHILE PorownajElement(pom^, elem_wzorcowy) <> '='
  DO pom := pom^.nastepny;

{wersja 2 - tez zla; obie strony AND sa obliczane}
pom := BD;
WHILE (pom<>NIL) AND
      PorownajElement(pom^, elem_wzorcowy) <> '='
  DO pom := pom^.nastepny;

{wersja 3 - dobra}
pom := BD;
znalazl := FALSE;
WHILE (pom<>NIL) AND (NOT znalazl) DO
  IF PorownajElement(pom^, elem_wzorcowy) = '='
    THEN znalazl := TRUE
    ELSE pom := pom^.nastepny;
```

```
{wersja 4 - dla list uporzadkowanych}
pom := BD;
znalazl := FALSE;
minal := FALSE;
WHILE (pom<>NIL) AND (NOT znalazl) AND (NOT minal) DO
  CASE PorownajElement(pom^, elem_wzorcowy) OF
    '>': minal := TRUE;
    '=': znalazl := TRUE;
    '<': pom := pom^.nastepny;
  END; {CASE}
```

Listy — tworzenie uporządkowane

```
PROCEDURE DodajElement5(VAR Lista: T_Lista;
                        Element  : T_Lista;
                        FUNCTION PorownajElement(e1,e2:T_Element):CHAR);
{wersja 5: w porzadku zgodnym z funkcja PorownajElement}
BEGIN    { realizacja rekurencyjna}
  IF Lista = NIL
  THEN BEGIN (* przypadek gdy lista jest pusta *)
         Element^.nastepny := NIL;
         Lista := Element;
       END
  ELSE
  CASE PorownajElement(Element^, Lista^) OF
    '<': BEGIN (* przypadek gdy element pasuje na poczatku *)
           Element^.nastepny := Lista;
           Lista := Element;
         END;
    '=': ; (* przypadek gdy element jest juz na liscie *)
    '>': (* przypadek ogolny, wstawiamy dalej na liste *)
         DodajElement5(Lista^.nastepny, Element, PorownajElement);
  END; {CASE}
END; {DodajElement5}
```

```
{realizacja iteracyjna przypadku ogolnego}
poprzedni := Lista;
dalej := TRUE;
WHILE (poprzedni^.nastepny <> NIL) AND dalej DO
  IF PorownajElement(Element^, poprzedni^.nastepny^) = '<'
    THEN dalej := FALSE
    ELSE poprzedni := poprzedni^.nastepny;
Element^.nastepny := poprzedni^.nastepny;
poprzedni^.nastepny := Element;
```

Listy — usuwanie elementu

• Musimy odróżnić czynność wyłączania elementu ze struktury danych od trwałego unicestwiania elementu, i odzyskiwania zajmowanej pamięci (służy do tego procedura DISPOSE).

• Najprostszy jest przypadek wyłączenia z listy pierwszego elementu, w pewnym sensie analogiczny do wstawiania elementu na początek listy:

```
PROCEDURE UsunElement1(VAR Lista: T_Lista;
                       VAR Element: T_Lista);
{wylacza pierwszy element z listy, i zwraca go w parametrze VAR}
BEGIN
  Element := Lista;
  Lista := Lista^.nastepny;
END; {UsunElement1}
```

⇒ zauważmy, że procedura nie sprawdza, czy lista nie jest przypadkiem pusta (co spowodowałoby błąd wykonania)

• wyłączanie z listy elementu, gdy mamy wskaźnik elementu poprzedniego

```
poprzedni^.nastepny := poprzedni^.nastepny^.nastepny;
```

Listy — usuwanie elementu (2)

• Usunięcie z listy konkretnego elementu, do którego mamy wskaźnik, wymaga przeszukiwania:

```
PROCEDURE UsunElement2(VAR Lista: T_Lista;
                           Element: T_Lista);
{usuwa wskazany element z listy nie niszczac go - wersja rekurencyjna}
BEGIN
  IF Lista <> NIL THEN {lista pusta lub koniec wywolania rekurencyjnego}
    IF Lista=Element
    THEN Lista := Lista^.nastepny {tu nastepuje rzeczywiste wylaczanie}
    ELSE UsunElement2(Lista^.nastepny, Element);
END; {UsunElement2}
```

• Wywołanie rekurencyjne w ostatniej procedurze można zastąpić pętlą:

```
pom := Lista;
WHILE (pom^.nastepny <> NIL) AND
      (pom^.nastepny <> Element) DO
  pom := pom^.nastepny;
IF pom^.nastepny = Element THEN
  pom^.nastepny := pom^.nastepny^.nastepny;
```

Listy — usuwanie elementu (3)

• Nie zawsze mamy jawny wskaźnik elementu, który chcemy usunąć — często wiemy tylko **jaki** element chcemy usunąć; możemy wtedy porównywać kolejno wszystkie elementy z elementem „wzorcowym".

• W ogólnym przypadku mamy schemat:

```
el := ZnajdzElement(Lista, element_wzorcowy, PorownajElement);
IF (el <> NIL) THEN UsunElement2(Lista, el);
```

⇒ Zauważmy, że lista jest przeszukiwana dwa razy.

• Schemat ten możemy powtarzać aby usunąć z listy kolejne (wszystkie) elementy pasujące do elementu wzorcowego.

• ostateczne kasowanie elementu ze zwalnianiem pamięci dynamicznej:

```
DISPOSE(el);
el := NIL;
```

⇒ Przypisanie wskaźnikowi wartości `NIL` ułatwia wykrywanie późniejszych błędnych odwołań do skasowanego elementu pamięci, ale im nie zapobiega!

Listy — usuwanie elementów (4)

```
PROCEDURE UsunElementy1(VAR Lista       : T_Lista;
                        Element_wzorcowy: T_Element;
                        FUNCTION PorownajElement(e1,e2:T_Element):CHAR);
{usuwa z listy i niszczy elementy rownowazne wzorcowemu}
{UWAGA: moze usuwac od konca do poczatku, lub na odwrot}
VAR pom: T_Lista;
BEGIN
  IF (Lista <> NIL) THEN
  BEGIN
    UsunElementy1(Lista^.nastepny, Element_wzorcowy, PorownajElement);
    IF PorownajElement(Lista^, Element_wzorcowy) = '=' THEN
    BEGIN
      pom := Lista;
      Lista := Lista^.nastepny;              {tu usuwamy 1 element }
      DISPOSE(pom);                          {tu odzyskujemy pamiec}
    END;
  END;
END; {UsunElementy1}
```

Listy — usuwanie elementów (5)

```
PROCEDURE UsunElementy2(VAR Lista       : T_Lista;
                        Element_wzorcowy: T_Element;
                        FUNCTION PorownajElement(e1,e2:T_Element):CHAR);
{realizacja iteracyjna, dziala tylko od poczatku do konca}
VAR pom, dalej: T_Lista;
BEGIN
  dalej := Lista;
  WHILE (dalej <> NIL) DO
    IF PorownajElement(dalej^, Element_wzorcowy) = '=' THEN
    BEGIN
      pom := dalej;
      dalej := dalej^.nastepny;
      IF Lista=pom THEN Lista:=dalej;        {tu usuwamy 1 element }
      DISPOSE(pom);                          {tu odzyskujemy pamiec}
    END
    ELSE
      dalej := dalej^.nastepny;
END; {UsunElementy2}
```

Operacje na zmiennych dynamicznych

• Zmienne dynamiczne tworzone w procedurach nie znikają w momencie zakończenia procedury lecz istnieją do chwili jawnego ich skasowania.

• Używając w procedurach wskaźników jako parametrów nie możemy korzystać z wiedzy, czy są to parametry VAR, czy zwykłe, aby zorientować się, czy procedura może zmienić stan zmiennych globalnych — zmienne dynamiczne są zawsze globalne.

• W procedurach operujących na dynamicznych strukturach danych trzeba szczególnie uważnie sprawdzać przypadki graniczne: koniec listy, pusta lista, pierwszy element, itp.

• Procedury i funkcje rekurencyjne są szczególnie przydatne w operacjach na dynamicznych strukturach danych — pozwalają na prosty zapis skomplikowanych operacji i szybkie dopracowanie poprawnie funkcjonujących programów — jednak ich użycie jest znacznie bardziej kosztowne niż odpowiednich procedur iteracyjnych.

$\Rightarrow$ Czasem stosuje się podejście polegające na użyciu procedur rekurencyjnych w wersji prototypowej programu, i przepisaniu wybranych procedur i funkcji na realizacje iteracyjne w wersji docelowej.

Problemy ze wskaźnikami

• Typowym błędem zdarzającym się w programach ze wskaźnikami jest „wypadnięcie" przez koniec listy, co jest trochę podobne do przekroczenia zakresu indeksów tablicy.

• Wskaźniki umożliwiają jednak popełnianie znacznie więcej rodzajów błędów, np. używanie niezainicjowanych wskaźników, bądź wskaźników do skasowanych zmiennych dynamicznych.

$\Rightarrow$ Takie błędy możemy łatwiej wykrywać systematycznie ustawiając nieużywane wskaźniki na NIL.

• Innym rodzajem błędów jest gubienie wskaźników do zmiennych dynamicznych (tworzenie nieużytków pamięci), bądź niezwalnianie niepotrzebnej pamięci dynamicznej, nawet jeśli nadal mamy do niej wskaźnik.

$\Rightarrow$ Aby temu zaradzić dobrze jest systematycznie zwalniać każdy element pamięci dynamicznej w chwili, gdy stał się niepotrzebny.

• Jeszcze innym rodzajem błędów, nie dającym się kontrolować w Pascalu (na szczęście ?!), jest przypadek wyczerpania pamięci dynamicznej.

Listy — przeglądanie

```
PROCEDURE PrzegladajListe(l: T_Lista;
                          PROCEDURE PrzegladajElement(e:T_Element));
(* wersja iteracyjna *)
BEGIN
  WHILE (l <> NIL) DO BEGIN
                        PrzegladajElement(l^);
                        l := l^.nastepny;
                      END;
END; {PrzegladajListe}


PROCEDURE PrzegladajListeR(l: T_Lista;
                           PROCEDURE PrzegladajElement(e:T_Element));
(* wersja rekurencyjna *)
BEGIN
  IF (l <> NIL) THEN BEGIN
                        PrzegladajElement(l^);
                        PrzegladajListeR(l,PrzegladajElement);
                      END;
END; {PrzegladajListeR}
```

Listy — przeglądanie (cd.)

• wywołanie procedury przeglądania listy wymaga podania każdorazowo jako parametru aktualnego nazwy procedury o specyfikacji zgodnej z deklaracją parametru formalnego

```
VAR BD: T_Lista;


PROCEDURE WyswietlElement(el: T_Element);
BEGIN
  WRITELN('Nazwa elementu: ', el.nazwa, ', numer id: ', el.id);
END; {WyswietlElement}


{...}


PrzegladajListe(BD, WyswietlElement);
```

Listy — odwracanie

```
PROCEDURE OdwrocListe(VAR lista:
                      T_Lista);
{przenoszenie elementow}

VAR elem, lista2: T_Lista;
BEGIN
  lista2 := NIL;
  WHILE (lista <> NIL) DO
  BEGIN

    {usuw.el.z pocz.1-ej listy}
    elem := lista;
    lista := lista^.nastepny;

    {dodaj.na poczatek drugiej}
    elem^.nastepny := lista2;
    lista2 := elem;
  END;

  lista := lista2;
END; {OdwrocListe}
```

```
PROCEDURE OdwrocListe2(VAR lista:
                       T_Lista);
{odwracanie wskaznikow w elementach}

VAR nast, lista2: T_Lista;
BEGIN
  lista2 := NIL;
  WHILE (lista <> NIL) DO
  BEGIN
    {zapamietujemy nast.element l-1}
    nast := lista^.nastepny;
    {odwrac.wskaznik w 1-ym el. 1-1}
    lista^.nastepny := lista2;
    {i wlaczamy go do listy drugiej}
    lista2 := lista;
    {przesuwamy wskaznik listy 1-ej}
    lista := nast;
  END;

  lista := lista2;
END; {OdwrocListe2}
```

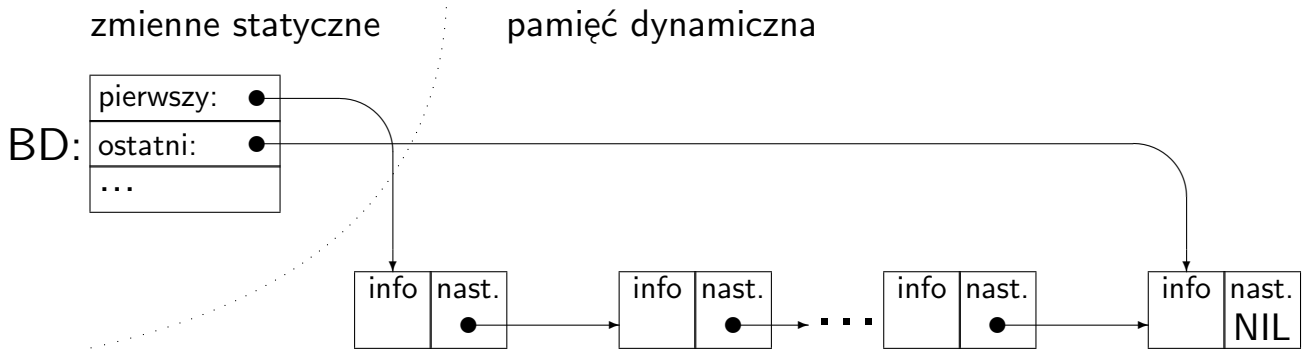
Listy — odwracanie, wersja rekurencyjna

rekurencyjna realizacja odwracania listy jest skomplikowana; teoretycznie możliwe jest zapisanie rekurencyjnie pierwszej koncepcji odwracania, tzn. przenoszenia kolejnych elementów z jednej listy na drugą, ale wymaga to innej specyfikacji nagłówka procedury, na przykład poprzez zewnętrzną procedurę „owijającą" (ang. *wrapper*), i wewnętrzną procedurę właściwą:

```
PROCEDURE OdwrocListeR(VAR lista: T_Lista);
VAR lista2: T_Lista;
                PROCEDURE OdwrocListeR_R(VAR l, l2: T_Lista);
                VAR e: T_Lista;
                BEGIN
                  IF l <> NIL THEN BEGIN e := l;
                                         OdwrocListeR_R(l^.nastepny,l2);
                                         e^.nastepny := l2; l2 := e;
                                    END;
                END; {OdwrocListeR_R}
BEGIN
  lista2 := NIL;
  OdwrocListeR_R(lista, lista2);
  lista := lista2;
END; {OdwrocListeR}
```

Listy z głową

```
TYPE T_WskElementu = ^T_Element;
     T_Element = RECORD
                   info: T_info;
                   nastepny: T_WskElementu;
                 END;
     T_ListaZG = RECORD
                   pierwszy: T_WskElementu;
                   {inne informacje o liscie, np.: ostatni, dlugosc}
                 END;

VAR BD: T_ListaZG;    {powinna byc statyczna}

BD.pierwszy := NIL;   {inicjalizacja listy}
BD.dlugosc  := 0;     {itd.}
```



Listy z głową (2)

- zastosowania głowy:

  - przechowywanie dwóch wskaźników listy, pierwszego i ostatniego elementu, w celu np.:
    - szybkiego dodawania węzłów na koniec listy
    - szybkiego włączania całej listy w środek innej listy
  - pojemnik na informacje: liczba elementów, liczba referencji, itp.
  - umożliwienie usuwania pierwszego elementu listy, gdy wskaźnik listy jest skopiowany w kilku miejscach, np. w strukturach danych
  - znacznik na listach kołowych

Listy z głową (3)

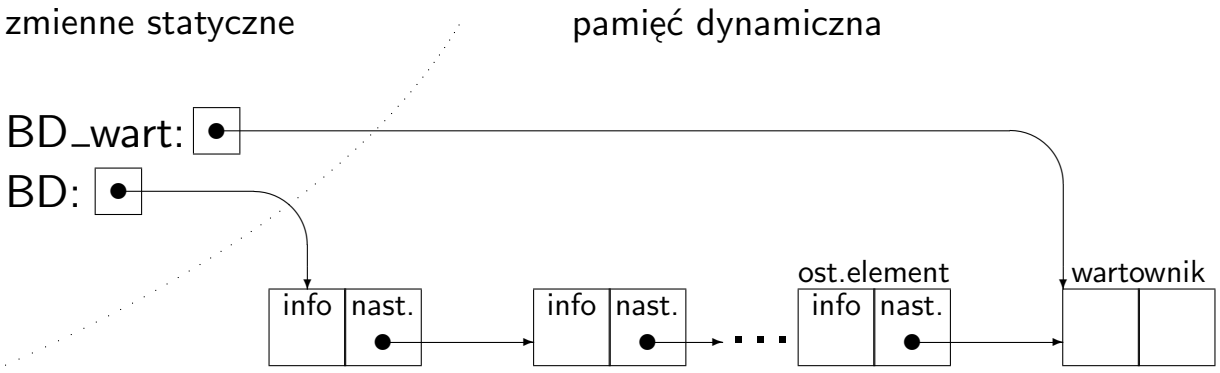
- użycie głowy komplikuje procedury rekurencyjne:

```
PROCEDURE DodajElement2(VAR Lista: T_ListaZG;
                        Element  : T_WskElementu);
{na koniec listy: wersja rekurencyjna dla listy z glowa}

  PROCEDURE DodajElement2_R(VAR l: T_WskElementu;
                            e    : T_WskElementu);
  BEGIN
    IF l = NIL
      THEN l := e
      ELSE DodajElement2_R(l^.nastepny, e);
  END; {DodajElement2_R}

BEGIN
  DodajElement2_R(Lista.pierwszy, Element);
  Element^.nastepny := NIL;
END; {DodajElement2}
```

Listy z wartownikiem

• posługiwanie się listą ma pewne niespójności ponieważ operacje na wskaźniku listy czasem muszą być różne dla listy pustej i niepustej

• możemy to zmienić stosując zwykły schemat listy z dodatkiem **wartownika** (ang. *guard*), czyli dodatkowego elementu na końcu listy, którego treść jest nieistotna i nieużywana



Listy z wartownikiem (2)

- posługiwanie się listą z wartownikiem:

```
VAR BD, BD_wartownik: T_Lista;

{inicjalizacja}
NEW(BD_wartownik);
BD := BD_wartownik;
```

- usprawnienia w kodzie, np. przeszukiwania, na jakie pozwala wartownik

```
{atrakcyjny ale niepoprawny schemat przeszukiwania}
pom := BD;
WHILE (pom<>NIL) AND PorownajElement(pom^, elem_wzorcowy) <> '='
  DO pom := pom^.nastepny;
IF (pom<>NIL) THEN WRITELN('Znaleziony!', pom^...);

{schemat jest poprawny dla listy z wartownikiem}
pom := BD;
WHILE (pom<>BD_wartownik) AND PorownajElement(pom^,elem_wzorcowy)<>'='
  DO pom := pom^.nastepny;
IF (pom<>BD_wartownik) THEN WRITELN('Znaleziony!', pom^...);
```

Listy dwustronnie połączone

```
TYPE T_ListaDP = ^T_Element;
     T_Element = RECORD
                   info: T_info;
                   poprzedni: T_ListaDP;
                   nastepny: T_ListaDP;
                 END;

PROCEDURE DodajElementDP(VAR Lista: T_ListaDP;
                             Element  : T_ListaDP);
BEGIN {dodawanie elementu przed dowolnym elementem listy}
  Element^.nastepny := Lista;
  Element^.poprzedni := NIL;
  IF (Lista <> NIL) THEN
  BEGIN
    Element^.poprzedni := Lista^.poprzedni;
    IF (Lista^.poprzedni <> NIL) THEN
      Lista.^poprzedni.^nastepny := Element;
    Lista.^poprzedni := Element;
  END
  ELSE Lista := Element; {konieczne bezwarunkowo gdyby zmienna Lista}
END; {DodajElementDP}    {miala wskazywac pierwszy element listy}
```

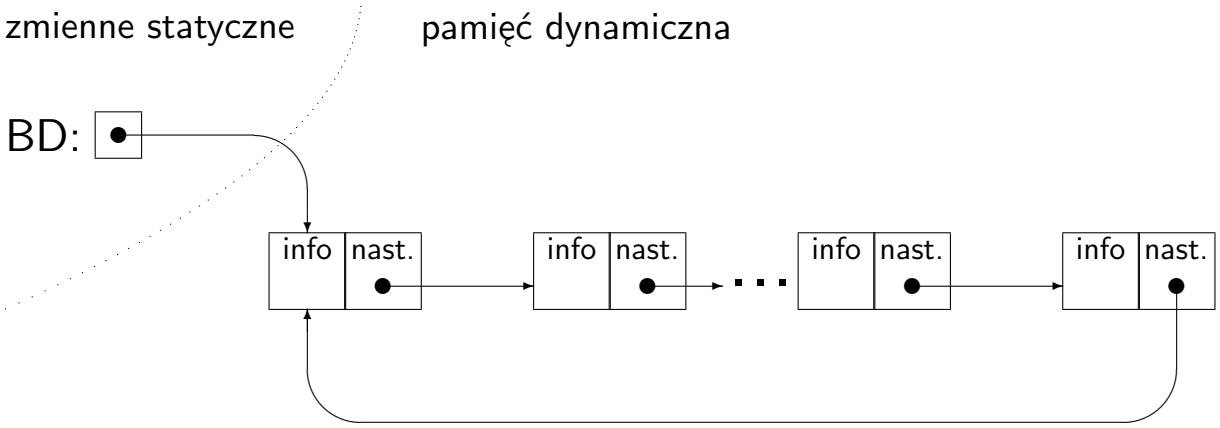
• listy dwustronnie połączone mają wiele cech odmiennych od „zwykłych” list, np. można się na nich „cofnąć”, łatwo włączać i usuwać elementy, dostęp do listy zapewnia wskaźnik dowolnego elementu

• jednak zasadnicze cechy list pozostają te same: sekwencyjny dostęp do elementów, brak istotnych korzyści z porządkowania elementów

• posługiwanie się listami dwustronnie połączonymi jest bardziej złożone niż zwykłymi listami, np. trzeba uważać na koniec listy po obu stronach

```
PROCEDURE UsunElementDP(VAR Lista: T_ListaDP;
                        Element  : T_ListaDP);
BEGIN {wylaczanie dowolnego elementu z listy}
  IF (Element^.nastepny <> NIL)            {w el.nastepnym, zaktualizuj}
  THEN Element^.nastepny^.poprzedni := Element^.poprzedni;   {poprzedni}
  ELSE IF (Element^.poprzedni <> NIL)            {jesli istn.poprzedni,}
       THEN Element^.poprzedni^.nastepny := NIL; {wpisz,ze nie ma nast.}
  IF (Element^.poprzedni <> NIL)           {w el.poprzednim,zaktualizuj}
  THEN Element^.poprzedni^.nastepny := Element^.nastepny;     {nastepny}
  ELSE IF (Element^.nastepny <> NIL)             {jesli istn.nastepny, }
       THEN Element^.nastepny^.poprzedni := NIL  {wpisz,ze nie ma poprz}
       ELSE Lista := NIL;                 {w tym przyp.lista jest pusta}
  IF (Lista = Element) THEN {...};        {ten przyp.tez wymaga korekty}
END; {UsunElementDP}
```

Listy kołowe



• listy kołowe nie mają fizycznego końca (ani początku), nie mają pustych wskaźników NIL, i są całkowicie symetryczne

• nadają się np. do realizacji tzw. buforów kołowych, służących do przechowywania pewnej ilości danych historycznych, automatycznie kasowanych w miarę zapisywania nowych danych

Połączenia

• często przydatne jest łączenie różnych modyfikacji podstawowego schematu listy, np.:

- lista z wartownikiem i wskaźnikiem ostatniego elementu
- lista kołowa dwustronnie połączona
- nie ma sensu dodawanie wartownika ani wskaźnika ostatniego elementu do listy kołowej
- głowa przydaje się, i można ją dodać do każdego rodzaju listy

Abstrakcyjne typy danych (wstęp)

• przez ATD (ang. *ADT*) rozumiemy określenie typu danych poprzez zdefiniowanie zestawu operacji, jakie mają być dlań dostępne

• ATD wnoszą wyższy poziom abstrakcji niż typy danych definiowane w programach, i pozwalają rozdzielić proces tworzenia programu na implementację ATD, oraz pisanie właściwego programu przy ich użyciu

• przykłady ATD:

- stosy (*LIFO*)
  - dodaj na stos (*push*)
  - zdejmij ze stosu (*pop*)
  - sprawdź szczyt stosu (*top*)
  - ◇ sensowna realizacja przez zwykłe listy, a także tablice statyczne
- kolejki chronologiczne (*FIFO*)
  - dodaj na koniec
  - usuń z początku
  - podaj długość kolejki (?)
  - ◇ sensowna realizacja przez listy ze wskaźnikiem ostatniego elementu, a także tablice statyczne (z „zawijaniem")