Wstęp do programowania

Listy

Do czego stosujemy listy?

- Listy stosuje się wszędzie tam, gdzie występuje duży rozrzut w możliwym rozmiarze danych, np. w reprezentacji grafów – jeśli węzłów jest dużo (n), a krawędzi niewiele, to każdy węzeł ma potencjalnie n-1 krawędzi, ale w praktyce może okazać się, że liczba krawędzi jest liniowa (np. dla grafów planarnych nigdy nie przekracza 6n-3).
- Listy umożliwiają nam łatwe wstawianie i usuwanie elementów, ale dostęp do nich nie jest bezpośredni.

Co to są listy?

- Zbiór list L(A) nad pewnym zbiorem A definiujemy, jako najmniejszy zbiór spełniający następujące warunki:
  - nil $\in$ L(A) jest wyróżnioną pustą listą
  - Dla a $\in$ A, l $\in$ L(A), para (a,l) jest listą.
- Przyjmujemy, że w parze (a,l) a jest pierwszym elementem listy, zwanym jej głową, a lista l ogonem, czyli pozostałą częścią listy (być może pustą).

Typ listowy

```
type lista=^elemlisty;
elemlisty = record
              w : typ;
           nast : lista;
            end;
```

- Zakładamy, że `typ` jest typem elementu zadeklarowanym wcześniej. Na tym wykładzie `typ=Integer`.

Wyszukiwanie w liście

```
function jest_r(l:lista;x:typ):Boolean;
{przyjmuje wartość true wtedy i tylko
  wtedy, gdy w liście l znajduje się
  wartość x}
begin
if l=nil then jest_r :=false
else if  l^.w = x then jest_r := true
     else jest_r := jest_r(l^.nast,x)
end;
```

Wady i zalety rekurencji w listach

- Zalety:
  - poprawność funkcji wynika wprost z definicji.
  - prosty kod.
- Wady:
  - liniowa złożoność pamięciowa.
  - czasowo również tracimy na obsłudze rekurencji.

Wyszukiwanie iteracyjne

```
function jest(l:lista; x:typA):Boolean;
{przyjmuje wartość true wtedy i tylko wtedy, gdy w
   liście l znajduje się wartość x}
begin
 if l=nil then jest :=false else begin
  while  (l^.nast<>nil) and (l^.w <> x) do  {od początku
    listy aż do l wyłącznie nie ma x; l≠nil}
          l := l^.nast;
  jest  :=  l^.w = x
 end
end;
```

Wyszukiwanie iteracyjne leniwe

```
function jest_leniwie (l:lista;
  x:typA):Boolean;
{Zakładamy leniwy tryb wyliczania
  warunków logicznych}
begin
   while  (l<>nil) and (l^.w <> x) do
  l := l^.nast; {od początku listy aż do
  l wyłącznie nie ma x}
   jest_leniwie:= l <> nil {wartość
  funkcji ustala inny warunek}
end;
```

Leniwa logika

- Przy wyliczaniu leniwym działamy w nieklasycznej trójwartościowej logice. Przerywamy bowiem obliczenia gdy tylko wiemy, jaki będzie wynik. Oznaczmy wartość niewyliczonego wyrażenia przez „?”. Mamy wtedy:
  - $\neg\ ?\ =\ ?,$
  - $? \wedge 1\ =\ ?,\ 1 \wedge ?\ =\ ?$
  - $0 \wedge ?\ =\ 0,\ ?\ \wedge 0\ =\ ?$
  - $1 \vee ?\ =\ 1,\ ?\ \vee 1\ =\ ?,$
  - $0 \vee ?\ =\ ?,\ ?\ \vee 0\ =\ ?$
- Alternatywa i koniunkcja nie są przemienne!

Wstawianie do listy

- Zacznijmy od prostej procedury wstawiającej nową wartość x **za** istniejącym elementem listy wskazywanym przez wsk:

```
procedure wstawZA(wsk:lista; x:typ);
var pom:lista;
begin
 new(pom);
 pom^.nast:=wsk^.nast;
 pom^.w := x;
 wsk^.nast:=pom
end;
```

Wstawianie do posortowanej listy

```
procedure wstaw_sort(var l:lista_A;x:typ);
{do posortowanej listy l wstawia x}
var akt,next: lista; {akt - aktualny, next
  - następnik akt}
begin {leniwie}
if (l=nil) or (x<=l^.w) then begin
 akt:=l; new(l); l^.nast:=akt; l^.w:=x
end {wstawiliśmy x na początek listy}
else {...}
```

Wstawianie do posortowanej listy

```
 begin
  akt := l;
  next := akt^.nast;
  while (next<>nil) and (next^.w<x) do
 {między l a akt włącznie są wartości
  mniejsze od x; akt<>nil next=akt^.nast}
  begin akt:=next; next:=akt^.nast end;

 wstawZA(akt,x)
 end
end;
```

Atrapa

- Czasem wygodnie jest używać atrapy jako elementu zaczynającego listę, ale do niej nienależącego.
- `type listaA=lista`
- Tworzenie pustej listy:
  - `new(l);`
  - `l^.nast:=nil`

Wstawianie do posortowanej listy z atrapą

```
procedure wstaw_sort_z_atrapa (l:lista_A;
  x:typ);{l może być wołana przez wartość}
var next: lista; {rolę zmiennej akt może
  pełnić parametr l}
begin next:=l^.nast;
if next=nil then {l pusta} wstawza(l,x)
else begin
 while (next^.w<x) and (next^.nast<>nil)
  do begin l:=next; next:=l^.nast end;
  if x>next^.w then wstawza(next,x)
  else wstawZA(l,x)
 end
end;
```

Listy ze strażnikiem

- Czasem wygodnie jest mieć dowiązanie nie tylko do poczzątku, ale i do ostatniego elementu listy
- Niekiedy taki ostatni element listy jest atrapą; nazywamy go wtedy strażnikiem.
- Taka implementacja jest nieco droższa pamięciowo, ale tylko o stałą, a za to umożliwia wstawianie na końcu listy, które jest bardzo przydatne, na przykład przy implementacji kolejek.

Listy ze strażnikiem

```
Type listaK=record
        pocz,kon:lista
     end;
```

- Zdefiniowaliśmy zatem listę z dowiązaniem do poczatku i do końca
- Teraz, w zależności od tego, czy chcemy mieć po prostu dowiązanie do końca listy, czy chcemy mieć dodatkowy rekord, nie bedący elementem listy (strażnika) bedziemy inaczej interpretowali wartości listy.

Listy puste ze strażnikiem

- Tworzenie listy pustej ze strażnikiem:

```
var l:listaK;
begin
 new(l.pocz);
 l.kon:=l.pocz
end.
```

- List ze strażnikiem najczęściej nie kończymy nilem. Rekord l.kon ma pole nast nieokreślone, a to, że wskaźnik p wyskoczył poza listę sprawdzamy za pomocą warunku `p=l.kon`.

Wstawianie do posortowanej listy ze strażnikiem

```
procedure wstaw_sort_ze_strażnikiem
                        (var l:listaK; x:typ);
var akt: lista;
begin l.kon^.w:=x;
if x<=l.pocz^.w then begin
   new(akt); akt^.w:=x; akt^.nast:=l.pocz;
   l.pocz:=akt end
else begin  akt:=l.pocz;
 while (akt^.nast^.w<x) do akt:=akt^.nast;
 wstawZA(akt,x);
end
end;
```

Lista pusta z atrapą i strażnikiem

```
Type listaAS=listaK;
```

- Tworrzymy listę pustą składającą się z pary atrap: początkowej i końcowej (strażnika).

```
var l_as:listaAS;
begin
 new(l_as.pocz);
 new(l_as.pocz^.nast);
 l_as.kon:=l_as.pocz^.nast
end;
```

Wstawianie do posortowanej listy zatrapą i strażnikiem

```
procedure wstaw_sort_ze_strażnikiem
                              (l:listaAS; x:typ);
{tym razem znowu wołamy l przez wartość}
var akt: lista;
begin
 l.kon^.w:=x;
 akt:=l.pocz;
 while (akt^.nast^.w<x) do akt:=akt^.nast;
 wstawZA(akt,x);
end;
```

Atrapy

- Nie należy przesadzać z atrapami, choć bywają one bardzo wygodne
- Czasami opłaca się wręcz stworzyć sobie sztucznie atrapę, wykonać algorytm, a na końcu usunąć ją, przywracając pierwotny charakter listy.
- Strażnika w łatwy sposób nie utworzymy, chyba, że mamy dowiązanie do końca listy. Trzeba wtedy czujnie odtworzyć stan sprzed zamieszania.

Listy dwukierunkowe

- Często wygodnie jest chodzić po liście w obu kierunkach.
- Typowe zastosowania
  - Drzewa
  - Rekurencja (pamiętamy na liście jakieś atrybuty znajdujące się na stosie)

Listy dwukierunkowe

```
type lista2=^record
                w: typ;
            poprz,nast: lista2
        end;

        lista_dwukierunkowa = record
              pocz,kon:lista2
        end;
```

- W liście dwukierunkowej pole poprz pokazuje na poprzedni, a pole nast na następny element listy.

Listy z cyklem

- Lista niekoniecznie musi kończyć sie nilem. Może się zdarzyć, że ostatni element listy pokazuje na któryś ze swoich poprzedników.
- Llistę nazwiemy cykliczną, jeśli ostatni element listy pokazuje na pierwszy.
- Ciekawe zadanie: rozpoznaj, czy lista ma cykl, czy też kończy się nilem.