Drzewa — podstawowe pojęcia

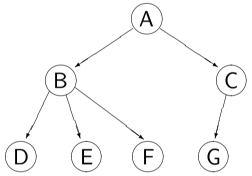
- **drzewo** — graf reprezentujący regularną strukturę wskaźnikową, gdzie każdy element zawiera dwa lub więcej wskaźników (ponumerowanych) do takich samych elementów; **węzły** (albo wierzchołki) grafu reprezentują elementy pamięciowe, a **łuki** reprezentują wskaźniki
- drzewo **binarne** — każdy element zawiera dokładnie dwa wskaźniki
- **korzeń** — element drzewa, od którego zaczynają się wskaźniki do pozostałych elementów
- **liście** — elementy drzewa, których oba wskaźniki są puste (NIL)
- **wewnętrzne** węzły drzewa — węzły, które nie są ani liściami ani korzeniem
- **ścieżka** — ciąg węzłów biegnący zgodnie ze wskaźnikami od korzenia do jakiegoś liścia
- **poddrzewo** — drzewo, które jest całkowicie zawarte w innym drzewie
- **rząd** drzewa — liczba wskaźników (≥ 2) w jednym węźle drzewa (niektóre z nich są NIL)
- **wysokość** — długość najdłuższej ścieżki drzewa

$$\lceil \log_k(N+1) \rceil \leq h_k(N) \leq N$$

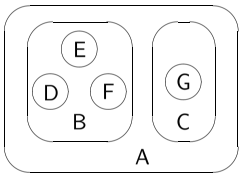
- **waga** — całkowita liczba węzłów w drzewie

Struktury drzewiaste

- grafy skierowane



- zbiory zagnieżdżone



- tekstowe listy symboli

```
(A,(B,(D),          (A (B (D)
      (E),                (E)
      (F)),               (F))
   (),                 ()
   (C,(G),             (C (G)
      (),                 ()
      ()))                ())))

(A,(B,(D),(E),(F)),(),(C,(G),(),()))
(A,(B,(D),(E),(F)),(),(C,(G)))
```

- typ danych drzew binarnych

```
TYPE
 T_Drzewo = ^T_Element;

 T_Element = RECORD
              info: T_info;
              lewe: T_Drzewo;
              prawe: T_Drzewo;
             END;
```


Dodawanie elementu do drzewa (1)

```
PROCEDURE DodajElement1(VAR drzewo: T_Drzewo;
                        element: T_Drzewo);
{ wersja minimalna }
BEGIN
   element^.lewe := drzewo;
   element^.prawe := NIL;
   drzewo := element;
END; {DodajElement1}
```

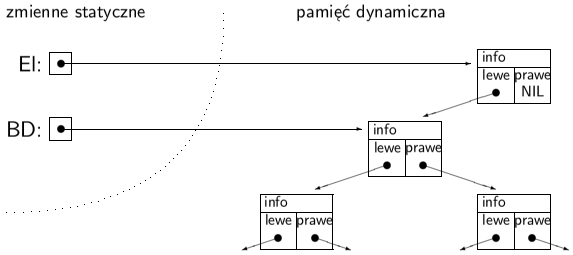
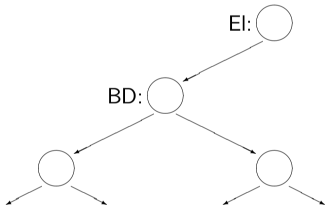
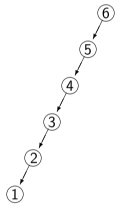


diagram uproszczony tworzonego drzewa binarnego:

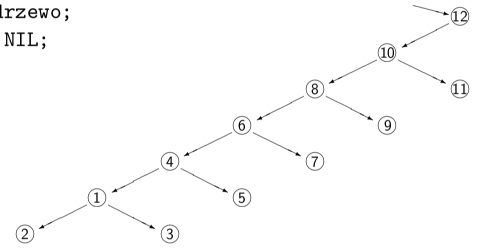


$\Rightarrow$ procedura `DodajElement1` tworzy drzewa zdegenerowane!



Dodawanie elementu do drzewa (2)

```
PROCEDURE DodajElement2(VAR drzewo: T_Drzewo;
                        element: T_Drzewo);
{ wersja ulepszona }
BEGIN
   IF drzewo = NIL THEN drzewo := element
   ELSE IF drzewo^.lewe = NIL THEN drzewo^.lewe = element
   ELSE IF drzewo^.prawe = NIL THEN drzewo^.prawe = element
   ELSE BEGIN
          element^.lewe := drzewo;
          element^.prawe := NIL;
          drzewo := element;
        END
END; {DodajElement2}
```



⇒ otrzymane drzewo nie jest wiele lepsze!

Dodawanie elementu do drzewa (3)

```
PROCEDURE DodajElement3(VAR drzewo: T_Drzewo;
                        element: T_Drzewo);
{ wersja "optymalna" }
BEGIN
   IF drzewo = NIL THEN drzewo := element
   ELSE IF Waga(drzewo^.lewe) < Waga(drzewo^.prawe)
        THEN DodajElement3(drzewo^.lewe, element)
        ELSE DodajElement3(drzewo^.prawe, element);
END; {DodajElement3}
```

⇒ procedura tworzy drzewa zrównoważone

• Definicja: drzewo binarne nazywamy **zrównoważonym**
albo: w pełni zrównoważonym,
albo: dokładnie zrównoważonym,
albo: zrównoważonym wagowo,

jeżeli dla każdego węzła liczby węzłów w jego lewym i prawym poddrzewie (czyli wagi tych poddrzew) różnią się co najwyżej o 1

Przeszukiwanie drzew binarnych

```
FUNCTION Wyszukaj(drzewo: T_Drzewo;
                  elem_wzorc: T_Element;
                  FUNCTION PorownajEl(e1,e2:T_Element):CHAR): T_Drzewo;
{znajd.na drz.el.rownowazny danemu;zwraca wsk.el.,lub NIL gdy go nie ma}
VAR test: T_Drzewo;
BEGIN
  IF drzewo=NIL THEN Wyszukaj := NIL
  ELSE
  IF PorownajEl(elem_wzorc, drzewo^) = '='
  THEN Wyszukaj := drzewo
  ELSE
  BEGIN
    test := Wyszukaj(drzewo^.lewe, elem_wzorc, PorownajEl);
    IF test <> NIL THEN Wyszukaj := test
    ELSE Wyszukaj := Wyszukaj(drzewo^.prawe, elem_wzorc, PorownajEl);
  END
END; {Wyszukaj}
```

Ćwiczenie: przepisać powyższą funkcję w wersji iteracyjnej!

Przeglądanie drzew binarnych

```
PROCEDURE Przegladaj(Drzewo: T_Drzewo;
                     PROCEDURE Op(e:T_Element));
BEGIN
  IF Drzewo<>NIL THEN
  BEGIN
    Op(Drzewo^);              (* porzadek "preorder" *)
    Przegladaj(Drzewo^.lewe);
    Przegladaj(Drzewo^.prawe);
  END;
END; {Przegladaj}
```

• przydatne porządki przeglądania drzewa binarnego:

- *preorder:* najpierw korzeń, potem całe lewe poddrzewo, potem prawe
- *inorder:* najpierw lewe poddrzewo, następnie korzeń, na końcu prawe
- *postorder:* najpierw lewe poddrzewo, potem prawe, na końcu korzeń

Drzewa uporządkowane

• Definicja: drzewo binarne jest uporządkowane gdy dla wszystkich węzłów drzewa spełniona jest własność

$$\begin{array}{c}\text{wszystkie elementy}\\ \text{w lewym poddrzewie}\end{array} < \text{korzeń} < \begin{array}{c}\text{wszystkie elementy}\\ \text{w prawym poddrzewie}\end{array}$$

• zyskujemy na przeszukiwaniu:

```
FUNCTION Wyszukaj(drzewo: T_Drzewo;
                  elem_wzorc: T_Element;
                  FUNCTION PorownajEl(e1,e2:T_Element):CHAR): T_Drzewo;
BEGIN
  IF drzewo=NIL THEN Wyszukaj := NIL
  ELSE CASE PorownajEl(elem_wzorc, drzewo^) OF
         '<': Wyszukaj := Wyszukaj(drzewo^.lewe, elem_wzorc, PorownajEl);
         '=': Wyszukaj := drzewo;
         '>': Wyszukaj := Wyszukaj(drzewo^.prawe, elem_wzorc, PorownajEl);
       END; {CASE}
END; {Wyszukaj}
```

• wysokość drzewa binarnego o N węzłach: $\lceil \log_2(N+1) \rceil \leq h_N \leq N$

Dodawanie elementu do drzewa (4)

• dodawanie uporządkowane jest bardzo podobne do wyszukiwania: praktycznie znajdujemy miejsce gdzie element powinien w drzewie się znaleźć, po czym go tam wklejamy

```
PROCEDURE DodajElement4(VAR drzewo: T_Drzewo;
                        element: T_Drzewo;
                        FUNCTION PorownajEl(e1,e2:T_Element):CHAR);
BEGIN
  IF drzewo=NIL
  THEN
    drzewo := Element
  ELSE
    CASE PorownajEl(element^, drzewo^) OF
      '<': DodajElement4(drzewo^.lewe, element, PorownajEl);
      '=': WRITELN('Element juz jest na drzewie!');
      '>': DodajElement4(drzewo^.prawe, element, PorownajEl);
    END;
END; {DodajElement4}
```


Budowa drzew uporządkowanych

• Od czego zależy wysokość drzewa utworzonego procedurą uporządkowanego dodawania elementów?

Rozważmy przypadki skrajne:

◦ ciąg elementów: A B C D E F ... X Y Z
⇒ wysokość drzewa będzie 26 ($= N$)

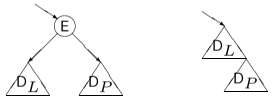
◦ ciąg elementów: M F T C I P W ... A L N Z
⇒ wysokość drzewa będzie 5 ($\approx \log_2(N)$)

• Zła wiadomość: wysokość tak tworzonych drzew uporządkowanych może wahać się od $\log_2(N)$ do N

• Dobra wiadomość: średnia oczekiwana wysokość drzewa tworzonego z przypadkowej sekwencji N elementów $\approx 1.4 \times \log_2(N)$

Usuwanie elementu z drzewa

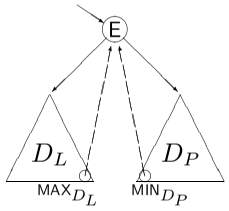
```
PROCEDURE UsunElement(VAR drzewo: T_Drzewo;
                      element: T_Drzewo);
                      FUNCTION PorownajEl(e1,e2:T_Element):CHAR);
{ wylacza z drzewa podany element }
BEGIN
  IF drzewo=NIL THEN WRITELN('Nie ma takiego elementu') ELSE
  CASE PorownajEl(element^, drzewo^) OF
    '<': UsunElement(drzewo^.lewe, element, PorownajEl);
    '>': UsunElement(drzewo^.prawe, element, PorownajEl);
    '=': IF drzewo^.lewe=NIL THEN drzewo := drzewo^.prawe ELSE
         IF drzewo^.prawe=NIL THEN drzewo := drzewo^.lewe ELSE
          BEGIN
            DodajElement4(drzewo^.lewe, drzewo^.prawe, Porownaj);
            drzewo := drzewo^.lewe;
          END;
  END; {CASE}
END; {UsunElement}
```



⇒ niestety, metoda pogarsza zrównoważenie

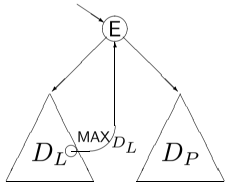
Usuwanie elementu z drzewa (2)

• ciekawy pomysł: aby usunąć z drzewa element E, przenosimy na jego miejsce maksymalny element jego lewego poddrzewa (albo minimalny element poddrzewa prawego); w ten sposób sprowadzamy usuwanie węzła do usuwania liścia



⇒ niestety, tej metody nie zawsze można użyć, bo nie zawsze MAX_{D_L} (albo MIN_{D_P}) będzie liściem w drzewie

• uzupełnienie pomysłu: jeśli MAX_{D_L} nie jest liściem to po przeniesieniu go na miejsce węzła E usuwamy go z kolei wywołaniem rekurencyjnym



Zastosowanie drzew uporządkowanych

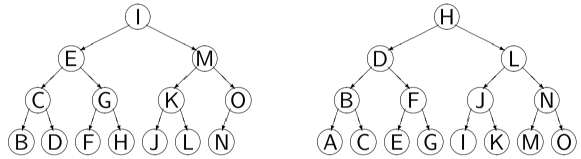
Podsumujmy: rozważamy struktury danych do przechowywania elementów np. jakiejś bazy danych. Dynamiczną strukturą danych — odpowiednikiem statycznej tablicy nieuporządkowanej — jest lista wskaźnikowa. Pomimo iż tablica, w odróżnieniu od listy, zapewnia bezpośredni dostęp do wszystkich elementów, bez sekwencyjnego przeszukiwania, to bez uporządkowania elementów każde przeszukiwanie i tak musi być sekwencyjne.

W takim razie dynamicznym odpowiednikiem tablic uporządkowanych są drzewa uporządkowane. W jednych i drugich możliwe jest niesekwencyjne wyszukanie konkretnego elementu. Jednak w przypadku drzew uporządkowanych liczba (albo czas) operacji niezbędnych do dotarcia do elementu zależy od zrównoważenia drzewa, nad którym nie umiemy zapanować (na razie).

Jednak zagadnienie równoważenia drzew uporządkowanych jest trudne — nie jest znana praktyczna metoda tworzenia drzew uporządkowanych i jednocześnie w pełni zrównoważonych. Podobnie nie istnieje żadna efektywna metoda równoważenia już zbudowanego drzewa. Jak się okaże, najlepszym rozwiązaniem są drzewa **częściowo zrównoważone**.

Zagadnienie równoważenia drzew

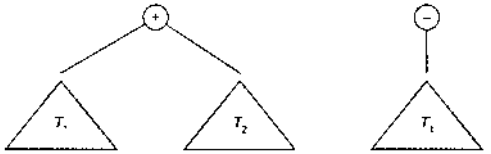
• rozważmy dodanie pojedynczego węzła do drzewa uporządkowanego z zachowaniem równowagi



⇒ struktura drzewa uległa całkowitej zmianie, żaden węzeł nie pozostał na swoim miejscu

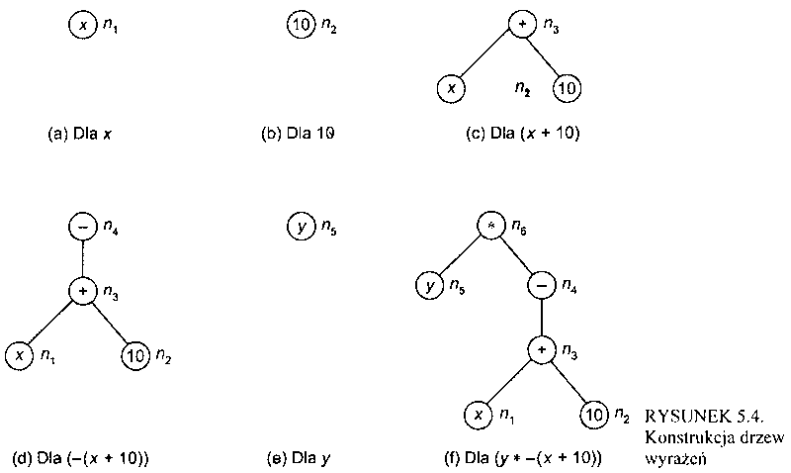
Inne zastosowania — drzewa wyrażeń

Drzewa binarne są przydatne w wielu zastosowaniach, nie tylko do przechowywania elementów jakiejś bazy danych. Jednym z nich są drzewa wyrażeń reprezentujące wyrażenia algebraiczne wyrażone w jakimś języku zawierającym operatory i operandy. Takie drzewa były użyte do reprezentacji wyrażeń Pascala — dowolne wyrażenie można przedstawić za pomocą drzewa, którego liście reprezentują zmienne i stałe, a węzły wewnętrzne operatory.



(a) $(E_1 + E_2)$ (b) $(-E_1)$

RYSUNEK 5.3. Drzewa wyrażeń dla (E_1+E_2) oraz $(-E_1)$



(a) Dla x (b) Dla 10 (c) Dla $(x + 10)$

(d) Dla $(-(x + 10))$ (e) Dla y (f) Dla $(y * -(x + 10))$

RYSUNEK 5.4. Konstrukcja drzew wyrażeń