Teraz przystępujemy do rozważania drzew – najważniejszej nieliniowej struktury, która pojawia się w algorytmice.

Donald E. Knuth

1 Drzewa binarne

Zdefiniujemy najpierw drzewo binarne formalnie.

Można spotkać się z definicjami, gdzie S nie jest singletonem, a po prostu korzeniem.

Definicja 1. *Drzewo binarne* to krotka (L, S, R), gdzie L i R to drzewa binarne lub zbiory puste, zaś S to singleton zawierający korzeń.

Uwaga 1. Powyższa definicja dopuszcza nieskończone drzewa, jednak my nie będziemy się nimi zajmować.

Zatem każde drzewo binarne ma korzeń $s \in S$, lewe poddrzewo L oraz prawe poddrzewo R. Ponadto korzeń ma lewego syna – korzeń poddrzewa L oraz prawego syna – korzeń poddrzewa R.

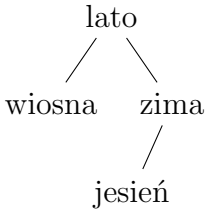
Przykład 1. Niech

$$T = ((\emptyset, \{\texttt{wiosna}\}, \emptyset), \{\texttt{lato}\}, ((\emptyset, \{\texttt{jesień}\}, \emptyset), \{\texttt{zima}\}, \emptyset)).$$

Krotka T to drzewo binarne. ♢

Jak widzimy powyższa definicja, chociaż jest bardzo ważna, to niezbyt praktyczna. Często drzewa binarne będziemy rysować.

Przykład 2. Poniższy rysunek przedstawia drzewo T z poprzedniego przykładu.



Odnotujmy, że istotne jest, czy węzeł jest lewym synem, czy prawym. ♢

Jak będziemy przechowywać drzewa binarne w pamięci komputera? W C++ moglibyśmy utworzyć taką strukturę:

```
struct Tree {
    Tree *L, *R;
    T s;
};
```

gdzie `L` i `R` to wskaźniki na lewe i prawe poddrzewo, zaś `s` – korzeń, z odpowiednim typem.

Zdefiniujmy jeszcze konstruktor:

```
Tree(Tree *L0, T s0, Tree *R0) {
    L = L0, s = s0, R = R0;
};
```

Przykład 3. Drzewo T z poprzedniego przykładu możemy zapisać w C++ w następujący sposób:

```
Tree *wiosna = new Tree(NULL, "wiosna", NULL);
Tree *jesien = new Tree(NULL, "jesień", NULL);
Tree *zima = new Tree(jesien, "zima", NULL);
Tree *lato = new Tree(wiosna, lato, zima);
```

$\Diamond$

Zdefiniujmy jeszcze dwie szczególne klasy drzew binarnych.

Definicja 2. *Regularnym drzewem binarnym* nazwiemy takie drzewo binarne, gdzie każdy węzeł ma zero lub dwóch synów, formalniej $L = R = \emptyset$ lub $L \neq \emptyset$ oraz $R \neq \emptyset$.

Przed zdefiniowaniem drugiej szczególnej klasy, zdefiniujemy wysokość, o której często będziemy mówić.

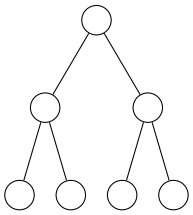
Definicja 3. *Wysokość drzewa binarnego* $T = (L, S, R)$ to $h(T) := 1 + \max\{h(L), h(R)\}$. Przyjmujemy $h(\emptyset) = -1$.

Czasami przyjmuje się $h(\emptyset) = 0$.

Przykład 4. Wysokość drzewa T z poprzedniego przykładu $h(T) = 2$. $\Diamond$

Definicja 4. Powiemy, że T jest *pełnym drzewem binarnym* iff ma $2^{h(T)+1} - 1$ węzłów.

Przykład 5. Poniżej przedstawiono 7-wierzchołkowe pełne drzewo binarne.



$\Diamond$

Możemy zaproponować czytelnikowi kilka ćwiczeń matematycznych.

Ćwiczenie 1. Ile jest różnych drzew binarnych o n wierzchołkach?

Ćwiczenie 2. Jaką najmniejszą wysokość może mieć n-wierzchołkowe drzewo binarne? A jaką największą?

Ćwiczenie 3. Dane jest regularne drzewo binarne, które ma l **liści** (węzłów, które nie mają synów, formalniej $L = R = \emptyset$). Ile ma ono wierzchołków niebędących liśćmi?

Ćwiczenie 4. Ile co najwyżej liści może mieć n-wierzchołkowe drzewo binarne? A ile najmniej?

2 Spacer po drzewie

Rozważmy następujący problem. W każdym węźle drzewa $T = (L, S, R)$ jest ciasteczko. Trzeba zjeść wszystkie ciasteczka. Sprawa jest trywialna, jednak nie dla komputera. Trzeba dokładnie sprecyzować w jakiej kolejności zjadać ciasteczka. Podamy teraz kilka takich kolejności.

Pierwszą, najprostszą jest kolejność *preorder*. Najpierw zjadamy ciasteczko z S, następnie wszystkie ciasteczka z L i na końcu wszystkie z R w tej samej kolejności. Widząc rekurencyjną strukturę możemy zaimplementować nasz algorytm.

```
void preorder(Tree *T)
{
    if (T == NULL)
        return;
    zjedz_ciasteczko(T->s);
    preorder(T->L);
    preorder(T->R);
}
```

Moglibyśmy najpierw zjeść ciasteczka z poddrzew, a dopiero potem z korzenia. Taką kolejność nazywamy *postorder*.

```
void postorder(Tree *T)
{
    if (T == NULL)
        return;
    postorder(T->L);
    postorder(T->R);
    zjedz_ciasteczko(T->s);
}
```

Jeszcze jedną, klasyczną i najważniejszą dla nas kolejnością jest kolejność *inorder*, zwana też *symmetric order*, co po polsku można przetłumaczyć jako *porządek symetryczny*. Najpierw zjadamy ciasteczka z L, potem S, a na końcu z R.

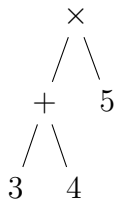
```
void inorder(Tree *T)
{
    if (T == NULL)
        return;
    inorder(T->L);
    zjedz_ciasteczko(T->s);
    inorder(T->R);
}
```

Przykład 6. Wyrażenia matematyczne możemy zapisywać w jednej z trzech notacji: notacji polskiej (od polskiego logika i matematyka Jana Łukasiewicza), gdzie operator występuje przed operandami, odwrotnej notacji polskiej,

gdzie oprator występuje po operandach oraz notacji wrostkowej (infiksowej, najpopoularniejszej) gdzie operator występuje pomiędzy operandami. Przykładowo

$$\texttt{NP}: \quad \times + 3\,4\,5$$
$$\texttt{ONP}: \quad 3\,4\,+\,5\,\times$$
$$\texttt{IN}: \quad (3+4)\times 5$$

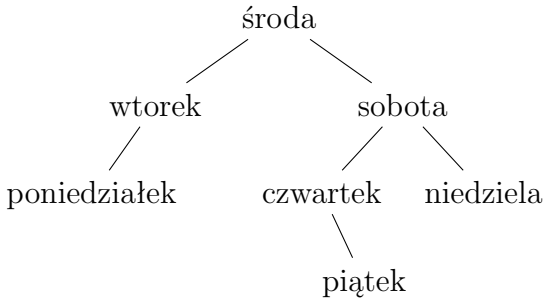
Zauważmy, że w dwóch pierwszych nie potrzeba używać nawiasów. Wyrażenie możemy przedstawić jako drzewo binarne. Przykładowo



Odczytane w preorderze da zapis w NP, w postorderze – w ONP, zaś w inorderze – w IN. ♢

Polecamy rozwiązać teraz kilka ćwiczeń.

Ćwiczenie 5. Wypisz węzły drzewa



w preorder, postorder i inorder.

Ćwiczenie 6. Przejście pewnego drzewa w preorder dało kolejność (A, B, D, E, C, F), zaś w inorder – (D, B, E, A, C, F). Znajdź to drzewo. Podaj algorytm dla ogólnego problemu. Czy drzewo jest w ten sposób zdefiniowane jednoznacznie? Co gdyby dany był postorder i inorder. A preorder i postorder?

Ćwiczenie 7. Z ćwiczenia 1. wiemy, że jest około 4^n różnych drzew o n wierzchołkach. Skoro $\log_2 4^n = 2n$, to potrzebujemy co najmniej $2n + o(1)$ bitów, żeby strukturę takiego drzewa przechować w pamięci. Rozważmy następującą procedurę.

```
void zakoduj(Tree *T)
{
    if (T == NULL)
    {
        output(0);
        return;
    }
```

```
        output(1);
        preorder(T->L);
        preorder(T->R);
    }
```

Użyliśmy tylko $2n + 1$ bitów. Pokazać, że z takiego zapisu możemy jednoznacznie odzyskać strukturę drzewa.

Ćwiczenie 8. Zaimplementować preorder, postorder i inorder iteracyjnie.

3 Binarne drzewa poszukiwań

Binarne drzewa poszukiwań, z angielskiego *binary search tree*, w skrócie BST, to bardzo użyteczna struktura danych.

Teraz będziemy często mówić o węzłach. Powiedzmy formalniej, że węzeł u w drzewie T jest korzeniem pewnego poddrzewa drzewa T. Ma on lewego i prawego syna – $l(u)$ oraz $r(u)$. Czasem zamiast poddrzewo lewego syna węzła u będziemy mówić po prostu lewe poddrzewo u i pisać $L(u)$. Analogicznie z prawym poddrzewem oznaczanym $R(u)$.

Powiedzmy, że węzły drzewa mają dodatkowe pole – klucz, który będziemy oznaczać przez $\texttt{key}(u)$.

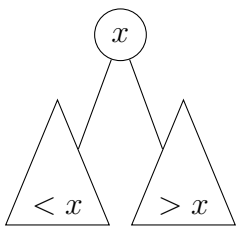
Trzeba jeszcze zmodyfikować definicję struktury w C++

```
struct Node {
    Node *l, *r;
    T key;
};
```

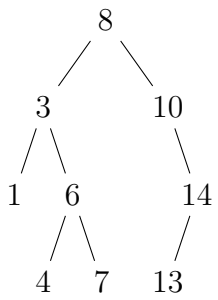
Od teraz będziemy zakładać, że na kluczach przechowywanych w węzłach zdefiniowana jest pewna relacja większości.

Zdefiniujmy najpierw następującą własność BST. Będziemy zwyczajnie mówić drzewo BST na drzewo, które ma własność BST.

Definicja 5. Powiemy, że drzewo T ma *własność BST* iff dla każdego węzła u drzewa T klucze wszystkich węzłów z $L(u)$ są mniejsze niż $\texttt{key}(u)$, zaś klucze wszystkich węzłów z $R(u)$ są większe niż $\texttt{key}(u)$.



Przykład 7. Drzewo



ma własność BST. ◊

Poniższe twierdzenie jest prostym, ale jednocześnie bardzo istotnym faktem.

Twierdzenie 1. *Niech drzewo T ma własność BST. Wtedy jego wierzchołki zapisane w porządku inorder są w kolejności rosnącej.*

Umiemy więc wypisać klucze BST w kolejności rosnącej. Teraz pokażemy jak znaleźć węzeł o najmniejszym kluczu. Wystarczy schodzić ciągle do lewego syna, który zawiera mniejsze wartości.

```
Node* minimum(Node *node)
{
    while (node->l != NULL)
        node = node->l;
    return node;
}
```

Analogicznie znajdujemy klucz największy.

Teraz pokażemy jak znaleźć rządany klucz w BST. Wystarczy użyć rekurencji i własności BST. Poniższy kod to realizuje.

```
Node* search(Node *node, T key)
{
    if (node == NULL)
        return NULL;
    if (node->key == key)
        return node;
    else if (key < node->key)
        return search(node->l, key);
    else if (node->key < key)
        return search(node->r, key);
}
```

Można jeszcze zastanowić się, jak dla zadanego węzła znaleźć jego następnik w kolejności inorder, tj. węzeł o następnym większym kluczu.

Jeśli węzeł ten, powiedzmy u, ma prawego syna, wystarczy znaleźć węzeł o minimalnym kluczu w prawym poddrzewie. W.p.p. następnikiem jest najniższy przodek u, którego lewy syn także jest przodkiem u.

Poniższy kod realizuje to, jednak zakłada on, że w węzłach trzymamy dowiązania do ojców. Odnotujmy, że nie wykonuje on żadnych porównań.

```
Node* successor(Node *node)
{
    if (node->right != NULL)
        return minimum(node->right);
    Node *p = node->parent;
    while (p != NULL && node == p->right)
    {
        node = p;
        p = node->parent;
    }
    return p;
}
```

Możemy uniknąć trzymania dowiązań schodząc po drzewie z góry na dół. Wtedy musimy znać jednak korzeń drzewa.

```
Node* successor(Node *root, Node *node)
{
    if (node->right != NULL)
        return minimum(node->right);
    Node *curr = root, *res = NULL;
    while (curr != NULL)
    {
        if (node->key < curr->key)
        {
            res = curr;
            curr = curr->left;
        }
        else if (node->key > curr->key)
        {
            curr = curr->right;
        }
        else
        {
            break;
        }
    }
    return res;
}
```

Analogicznie możemy znaleźć poprzednika.

Umiemy już wykonywać operacje statyczne. Następnie pokażemy jak aktualizować BST, tj. dodawać i usuwać klucze.

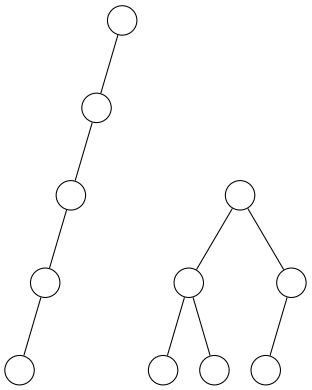
Odpowiedzi

1 Niech C_n będzie to liczba n-wierzchołkowych drzew. Wprost z definicji mamy

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}.$$

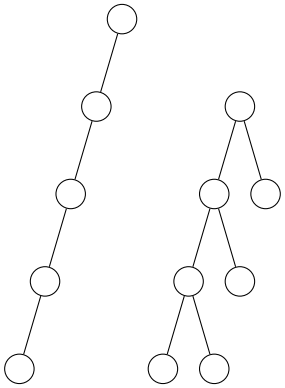
Przyjmujemy $C_0 = 1$. Znanym faktem jest, że powyższa rekurencja opisuje n-tą liczbę Catalana. Odnotujmy jeszcze, że $C_n = O(4^n / \log n)$.

2 Drzewo binarne o wysokości h może mieć co najwyżej $2^{h+1}-1$ wierzchołków. Zatem $n \leqslant 2^{h+1}-1$, czyli $h \geqslant \log_2(n+1)-1$. Ponadto $h \leqslant n-1$. Odnotujmy, że te ograniczenia są osiągalne.



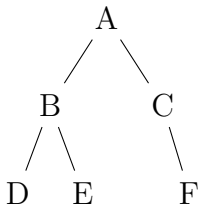
3 Zliczając krawędzie na dwa sposoby, wnioskujemy, że ma ono $l-1$ wierzchołków niebędących liśćmi.

4 Co najmniej 1, a co najwyżej $\lceil n/2 \rceil$.



5 Prerder: środa, wtorek, poniedziałek, sobota, czwartek, piątek, niedziela. Postorder: poniedziałe, wtorek, piątek, czwartek, niedziela, sobota, środa. Inorder: poniedziałek, wtorek, środa, czwartek, piątek, sobota, niedziela.

6 Było to następujące drzewo.



Ogólnie, należy znaleźć korzeń – pierwszy wierzchołek w preorder i podzielić rekurencyjnie inorder na dwie częsci w miejscu korzenia. Preorder lub postorder wraz z inorder definiują drzewo jednoznacznie. Preorder z postorder już nie.

7 Jeśli czytamy 1, to rekurencyjnie odczytujemy lewe i prawe poddrzewo. Jeśli przeczytaliśmy 0, wiemy, że należy przerwać to wywołanie.

8 Należy użyć stosu. Preorder i inorder są dosyć nietrudne. Aby zaimplementować postorder trzeba przetrzymywać dodatkowe informacje.