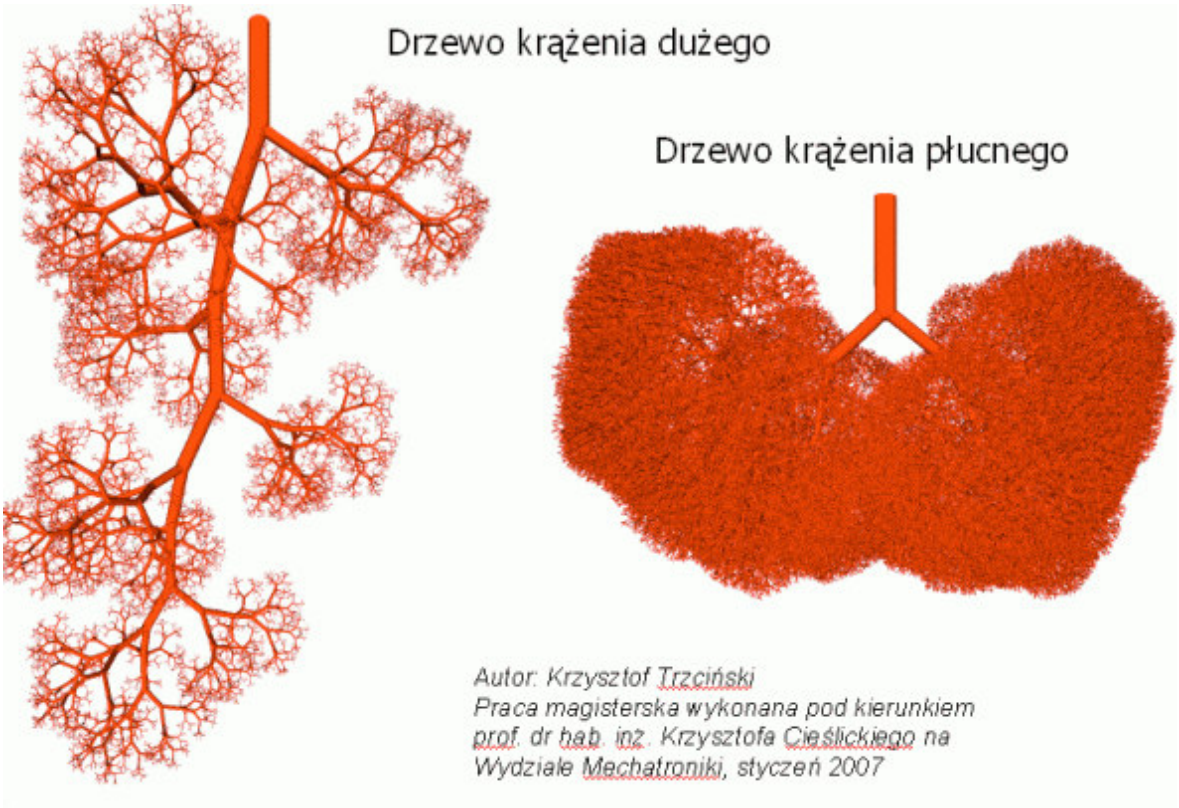
# Lekcja 5: Drzewa binarne

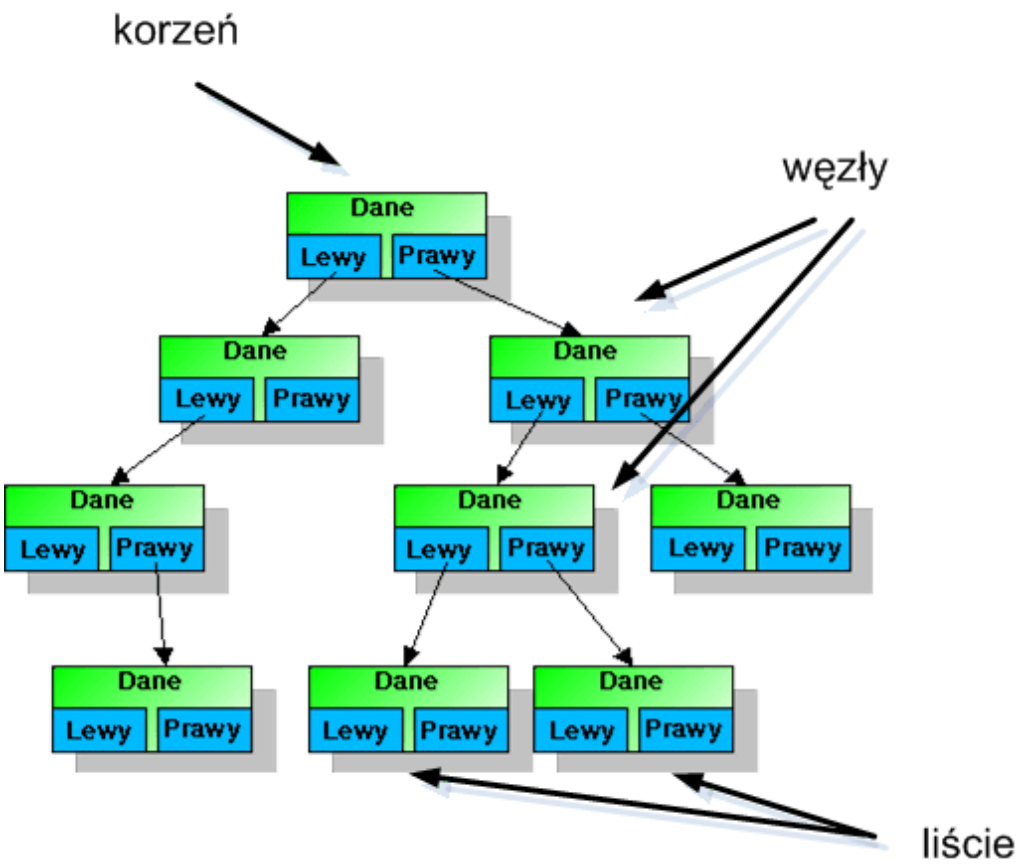
## Wstęp

W tym rozdziale zajmiemy się drzewami binarnymi, które są dynamicznymi strukturami rekurencyjnymi. Zamiast opisywać ich siłę i możliwości - prezentujemy interesujący przykład zastosowania. To dla zachęty, byście polubili tworzenie struktur drzewiastych.




### Drzewo binarne

W Waszych programach niejednokrotnie zajdzie potrzeba zastosowania bardziej wyrafinowanych struktur danych niż prosta lista. Listy są znacznie bardziej elastyczne niż tablice (poprzez dynamiczny sposób ich tworzenia), niestety są one tylko strukturami liniowymi i przedstawienie złożonych relacji między obiektami przy użyciu list jest mało wygodne. Strukturą, którą zajmiemy się w tym rozdziale, będzie **drzewo** (ang. *tree*). Tak jak lista jednokierunkowa jest najprostszą dynamiczną strukturą liniową (jednowymiarową), tak drzewo jest dynamiczną strukturą wielowymiarową. Drzewo jest regularną strukturą danych składającą się z węzłów (odpowiednik elementów na liście) posiadających co najmniej dwa wskaźniki do takich samych węzłów. Tak, jak prawdziwe drzewo w przyrodzie posiada liczne rozgałęzienia, tak samo i drzewo w informatyce posiada węzły (wierzchołki) – miejsca rozgałęzień. Każdy węzeł może wskazywać na kilka innych węzłów w drzewie. Jedynym węzłem, na którego nie wskazuje żaden inny węzeł drzewa, jest **korzeń**. Natomiast węzły, których wskaźniki są puste (czyli nie wskazują na żaden inny węzeł), nazywamy najczęściej **liśćmi**. Na ogół drzewa są przedstawiane w sposób graficzny z korzeniem na górze rysunku oraz z liśćmi na dole. Węzły wskazywane przez dany węzeł nazywamy jego **synami** (potomkami), natomiast on jest w stosunku dla nich **ojcem**. Najprostszym i zarazem najczęściej używanym rodzajem drzewa jest **drzewo binarne**. Jest to struktura dwuwymiarowa. W praktyce przejawia się to tym, że każdy element (węzeł) ma dokładnie dwie odnogi - lewą i prawą (oczywiście nie każda z nich musi być wypełniona). Rysunek poniżej pokazuje, jak takie drzewo może wyglądać.




Kolorem niebieskim oznaczyliśmy wskaźniki do innych węzłów drzewa. Żeby uciąć wątpliwości – wskaźniki równe NULL/nil nie są narysowane w postaci strzałek. Typowa definicja pojedynczego węzła będącego składnikiem drzewa binarnego jest następująca:

```
struct Twezel  {
   int Dane;
   Twezel *Lewy;
   Twezel *Prawy;
 };
```

Każdy węzeł zawiera pole do przechowywania informacji oraz dwa wskaźniki. Jak widzimy, wskaźniki w węźle zostały nazwane mianem lewego oraz prawego – na ogół stosuje się właśnie takie nazewnictwo, w którym węzeł może mieć lewego oraz prawego syna. Pamiętajmy, że każdy węzeł w drzewie binarnym nie musi mieć obu synów – może nie mieć żadnego, może mieć również jednego z synów (lewego lub prawego).

W tym miejscu należy wyjaśnić jedną niezmiernie istotną rzecz. Otóż, jak wiecie, lista dwukierunkowa składa się z elementów zawierających dwa wskaźniki. Różnica między taką listą a drzewem kryje się w sposobie połączenia elementów. Elementy listy dwukierunkowej pokazują na siebie wzajemnie, tzn. jeśli drugi element listy wskazuje na trzeci, to trzeci wskazuje na drugi. Z tego faktu wynika jeszcze jedno – każdy element listy z wyjątkiem pierwszego i ostatniego **musi** mieć dwóch sąsiadów – czyli oba wskaźniki w nim umieszczone są różne od NULL/nil.

Natomiast przy drzewie binarnym zależność ta nie jest spełniona. Po pierwsze: węzeł może mieć jednego, dwu, lub nie mieć wcale synów. Po drugie: jeśli A wskazuje na B, to B **nie może** wskazywać na A.

## Drzewa binarnego wyszukiwania

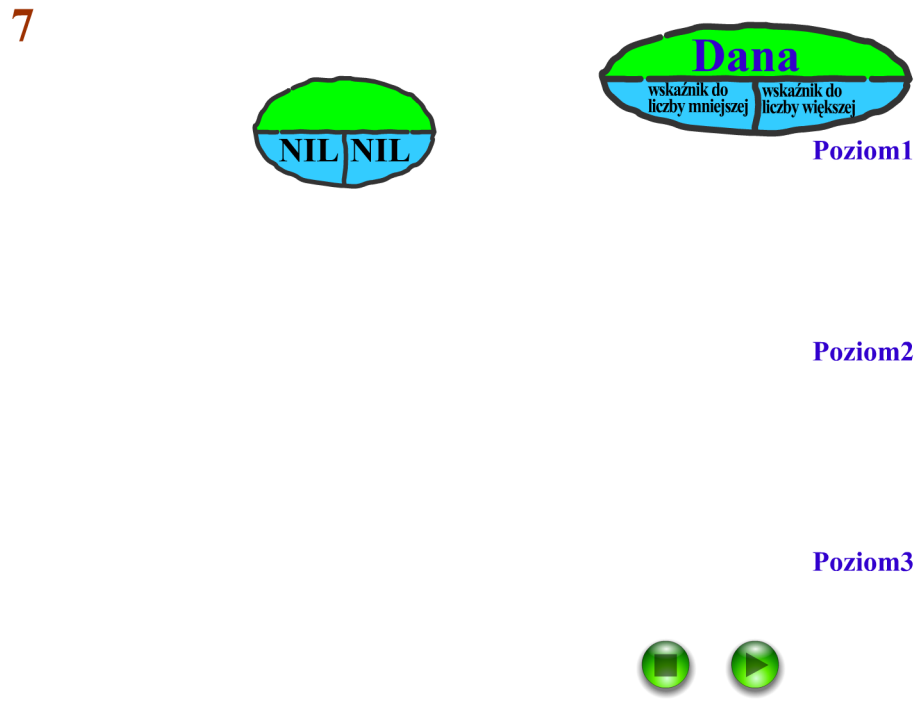
Drzewa binarne mogą być tworzone według różnych reguł. Jedną z grup drzew binarnych, na której są określone reguły zachowywania się elementów w drzewie są drzewa binarnego przeszukiwania (ang. **Binary Search Tree** - **BST**). Struktura BST jest stosowana przede wszystkim w celu szybkiego wyszukiwania danych (a także dodawania i usuwania informacji); często przy użyciu takich drzew realizowane są struktury słownikowe oraz kolejki priorytetowe, które dopiero poznacie. Główna zasada istniejąca w drzewie BST jest taka, że wartości w węzłach znajdujących się „na lewo" od danego węzła są mniejsze niż wartość zapisana w tym węźle, natomiast wszystkie wartości w węzłach znajdujących się „na prawo" od danego węzła są większe niż wartość w nim zapisana. Mówiąc ścisłej, wartość w węźle jest większa niż wartości w węzłach znajdujących się w lewym **poddrzewie** (czyli drzewie o korzeniu będącym jego lewym synem), natomiast wartość w konkretnym węźle jest mniejsza niż wartości w węzłach znajdujących się w prawym poddrzewie. Zatem:

**wszystkie wartości lewego poddrzewa < wartość w węźle < wszystkie wartości prawego poddrzewa**

Ta zależność obowiązuje na każdym poziomie drzewa BST, a więc zarówno dla jego korzenia, jak i dla wszystkich poddrzew dowolnego węzła. Możecie to sprawdzić na poniższej animacji i na wszystkich następnych rysunkach w tej lekcji.

Struktura BST wymaga, by każdy węzeł posiadał unikatową wartość danej - to bardzo ważne: elementy drzewa BST nie mogą się powtarzać!.

Przyjrzyjcie się, jak działa proces tworzenia drzewa BST:



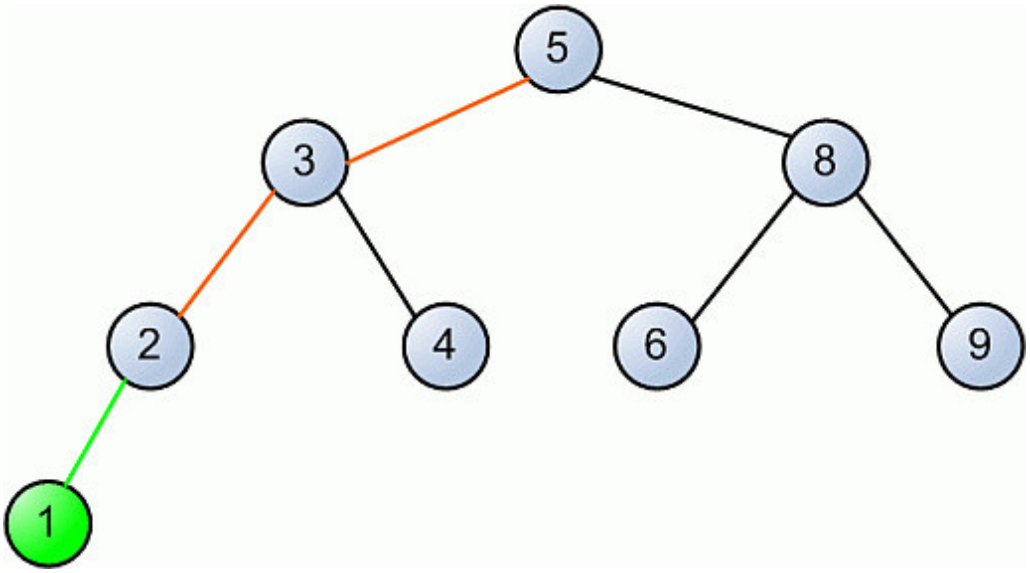

Skoro pokazaliśmy w sposób graficzny, jak dodawane są nowe elementy do drzewa BST, to również warto przedstawić ścisły kod budowania takiego drzewa. Kod realizujący dodawanie nowego węzła można przedstawić w postaci procedury rekurencyjnej (z przekazywaniem referencji do wskaźnika), wraz z przykładowym jej wywołaniem w programie:

```
void  DodajWezel (Twezel *&wezel, int wartosc) {
   if (wezel == NULL)  { // tworzymy i dołączamy element
       wezel = new Twezel;
       wezel -> Dane = wartosc;
       wezel -> Lewy = NULL;
       wezel -> Prawy = NULL;
   }
   else
     if (wartosc < wezel -> Dane) DodajWezel (wezel -> lewy, wartosc );    // rekurencyjne wyw
   else
     if (wartosc > wezel -> Dane) DodajWezel (wezel -> prawy, wartosc ); // rekurencyjne wywol
   // uwaga: liczby powtarzające się nie są dodawane do drzewa
}

int main( ) {
   int n, liczba;
   Twezel korzen;
   korzen=NULL;
   cin >> n;          // załóżmy, że n liczb ma być wczytanych
   for (int i=0; i<n; i++) {
    cin >> liczba;
    DodajWezel (korzen, liczba);
    // teraz korzen nie jest już NULL
    ...
   }
}
```

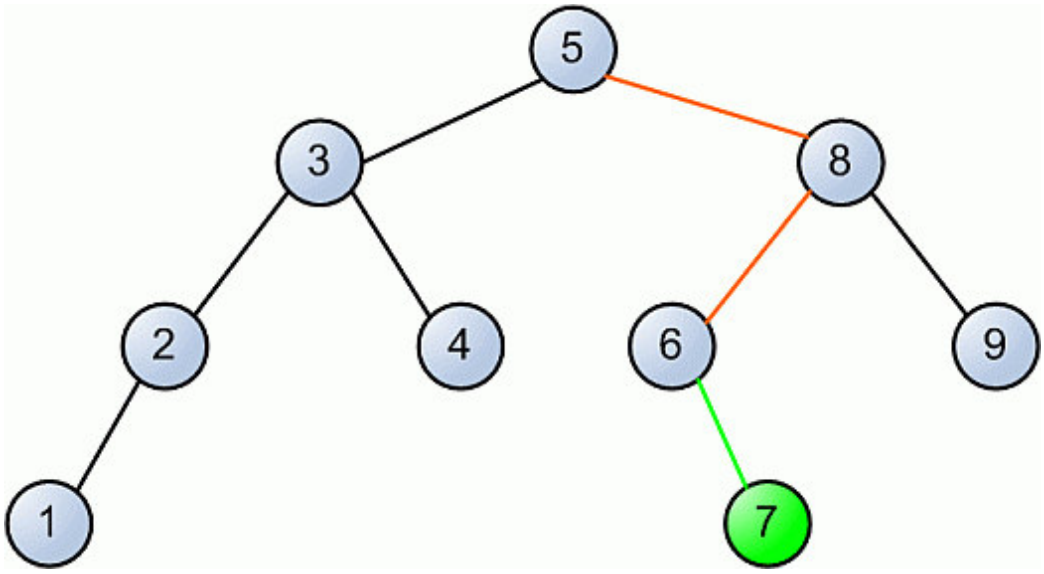
Algorytm dodawania węzła w sposób rekurencyjny wykonuje się się od korzenia w stronę liści, poruszając się w lewo bądź w prawo w zależności od wyniku porównania z napotkanymi węzłami. W momencie dotarcia do jednego z liści następuje ostatnie porównanie wartości, po czym nowy węzeł staje się lewym lub prawym synem węzła, który do tego czasu był liściem. Złożoność czasowa algorytmu *DodajWezel* jest rzędu **O(h)**, gdzie h jest **wysokością** drzewa. Dopiero teraz wprowadzamy pojęcie wysokości – oznacza ona odległość korzenia (w sensie liczby krawędzi) od najbardziej oddalonego liścia drzewa.

Warto w tym miejscu pokazać na rysunkach, w jaki sposób dodawany węzeł wędruje w od korzenia w stronę liści w poszukiwaniu odpowiedniego miejsca dla siebie.

Przykładowo, chcemy wczytać ciąg liczb 5, 3, 8, 2, 6, 4, 9, 1, 7, które są wartościami zapisywanymi w węzłach. Po utworzeniu drzewa BST z liczb 5, 3, 8, 2, 6, 4, 9 prezentujemy poniżej dodanie węzła o wartości 1:




Z kolei w następnym (ostatnim) kroku dodawany jest węzeł o wartości 7:




Skoro wiemy już, w jaki sposób tworzy się drzewo BST, warto by było poznać korzyści płynące z używania takiej struktury danych. Jedna z nich zawarta jest w procedurach wyświetlania drzewa BST. Otóż omawiane drzewo można wyświetlić na 3 główne sposoby:

- w kolejności **inorder**: najpierw lewe poddrzewo, potem węzeł, a następnie prawe poddrzewo
- w kolejności **preorder**: najpierw węzeł, potem lewe poddrzewo, a następnie prawe poddrzewo
- w kolejności **postorder**: najpierw lewe poddrzewo, potem prawe poddrzewo, a na samy końcu węzeł

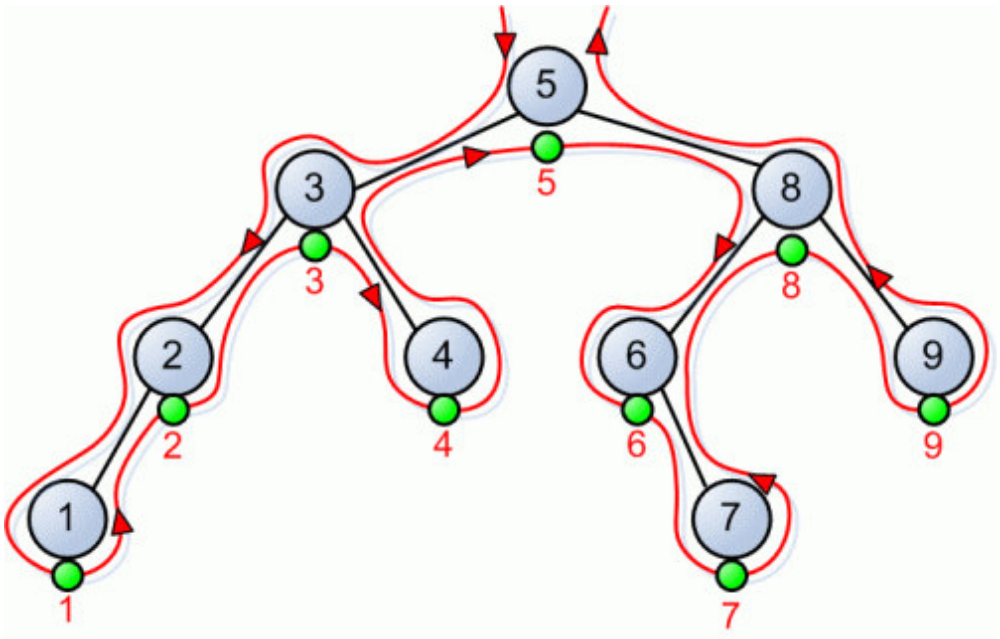
Powyższe sposoby tak naprawdę dotyczą metod przechodzenia przez drzewo i kolejności odwiedzania jego wierzchołków – przy okazji „odwiedzin" możemy wyświetlić zawartość tego węzła.

Co ciekawe, jeden z tych sposobów zwiedzania wierzchołków daje w rezultacie wyświetlanie zawartości węzłów w kolejności posortowanej (rosnącej). Chodzi konkretnie o metodę **inorder**.

```
void WyswietlDrzewo (Twezel *wezel)   {

  // porządek inorder
   if (wezel != NULL)  {
         WyswietlDrzewo (wezel->Lewy);              // lewe poddrzewo
         cout << wezel->Dane << " ";               // węzeł
         WyswietlDrzewo (wezel->Prawy);           // prawe poddrzewo
  }
}
```

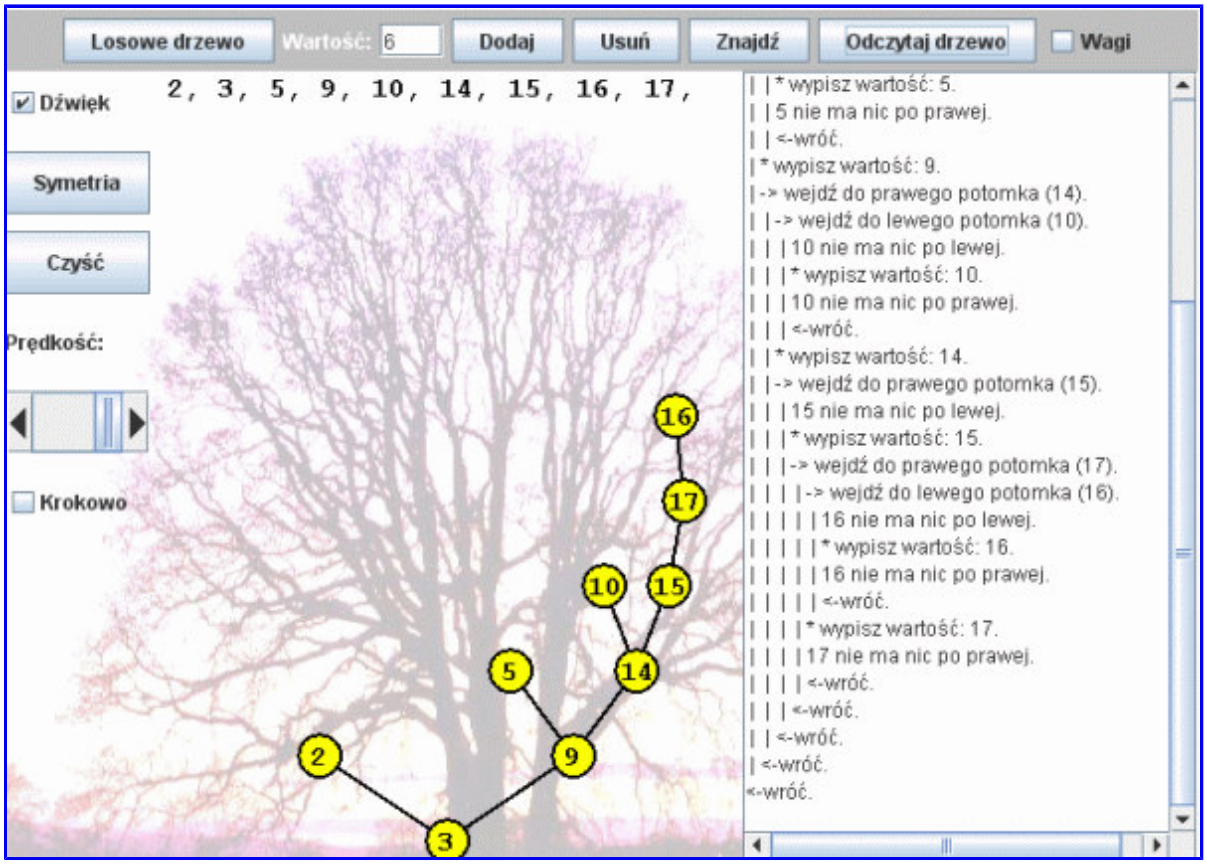
Odnosząc się do naszego przykładowego drzewa, liczby zapisane w węzłach zostaną wyświetlone w kolejności: **1, 2, 3, 4, 5, 6, 7, 8, 9.**

Odczytanie danych w kolejności **inorder** wymaga rekurencyjnego sięgania w pierwszej kolejności do skrajnie lewych węzłów drzewa, następnie w razie konieczności cofamy się o jeden poziom, odczytujemy wartość węzła, przechodzimy do jego prawego syna i ponownie próbujemy podążać skrajnie na lewo. Algorytm wykonuje się do momentu odwiedzenia wszystkich węzłów drzewa. Zauważcie, że wartość węzła jest odczytywana dopiero przy jego drugich "odwiedzinach", jak już nie możemy iść dalej w lewo. Ta kolejność odczytywania elementów wynika z kolejności rekurencyjnych wywołań w procedurze powyżej i możecie prześledzić ją na poniższym rysunku, gdzie czerwoną linią zaznaczona została droga poruszania się wzdłuż drzewa w trakcie tych wywołań. Zielone kółeczka oznaczają miejsca odczytu węzłów, czerwone numerki przy nich to numer kolejny odczytywanego węzła.




**Mamy więc prosty sposób sortowania danych:**
**wpisać je do drzewa BST, a następnie odczytać w kolejności inorder.**

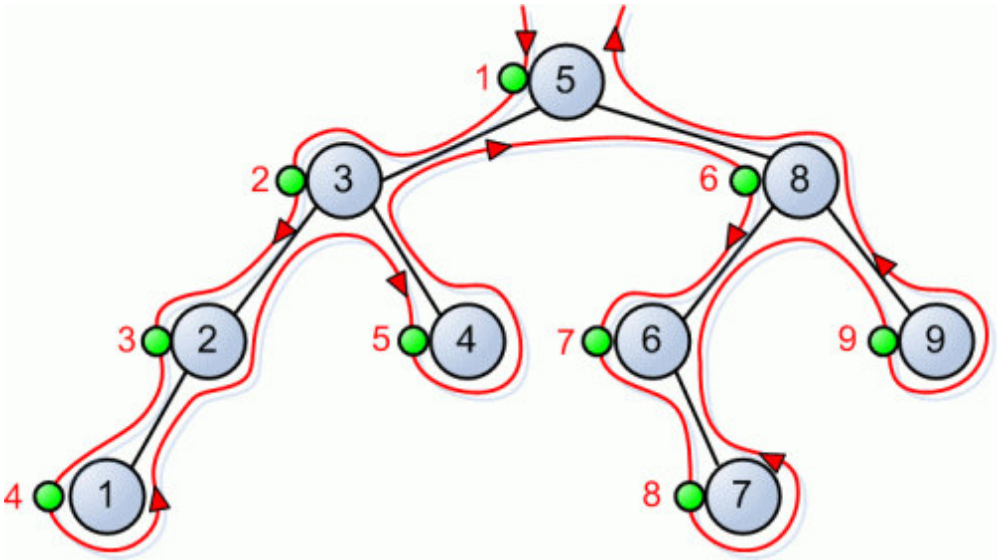
Możecie to teraz dokładnie prześledzić na różnych przykładach drzew, korzystając ze specjalnie napisanego w tym celu apletu - wejdziecie do niego klikając w poniższy obrazek. Autor tego ciekawego apletu przygotował go dla Was w postaci tradycyjnego drzewa, rosnącego od dołu do góry - ale zauważcie, że spełnia ono nadal wszystkie warunki drzewa BST, wartości w lewym poddrzewie są mniejsze niż w prawym itd. Tradycyjne drzewo binarne (rosnące do dołu) można z niego otrzymać przez odbicie lustrzane względem poziomej linii u podnóża drzewa, nie zaś przez obrót.




Dla porównania przedstawimy również kod programu oraz schemat rysunkowy dla metod preorder oraz postorder.

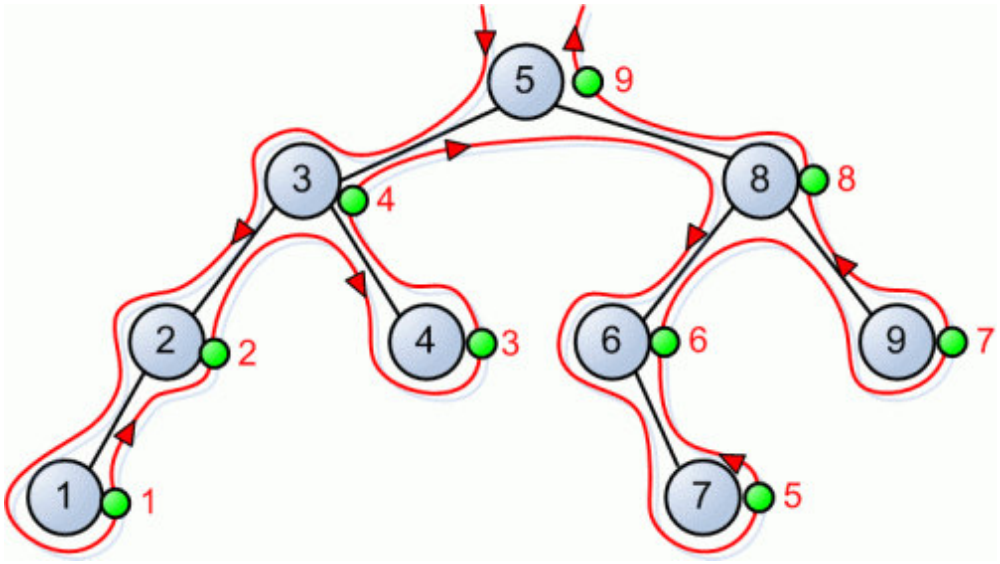
```
void WyswietlDrzewo (Twezel *wezel) {

  // porządek preorder
  if (wezel != NULL)  {
       cout << wezel->Dane << " ";
       WyswietlDrzewo (wezel->Lewy);
       WyswietlDrzewo (wezel->Prawy);
   }
}
```

Dla wariantu odwiedzania węzłów **preorder** liczby z wierzchołków dla naszego przykładowego drzewa będą wyświetlane w następującym porządku: **5, 3, 2, 1, 4, 8, 6, 7, 9.**




```
void WyswietlDrzewo (Twezel *wezel)  {

     // porządek postorder
     if (wezel != NULL)  {
        WyswietlDrzewo (wezel->Lewy);
        WyswietlDrzewo (wezel->Prawy);
        cout << wezel->Dane << " ";
   }
}
```

Z kolei dla porządku odwiedzania **postorder** węzły są odwiedzane (i wyświetlane) w następującej sekwencji: **1, 2, 4, 3, 7, 6, 9, 8, 5.**




## Wyszukiwanie w drzewie BST

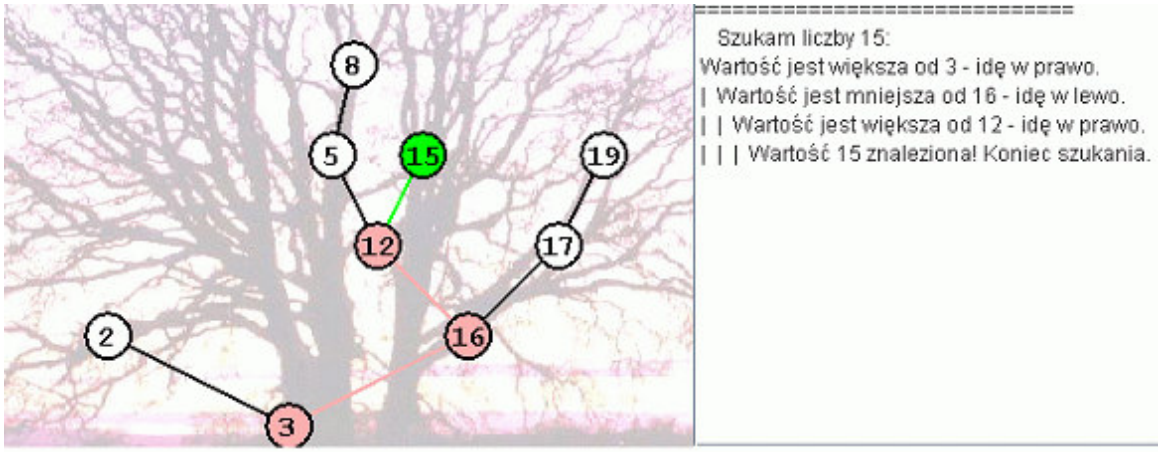
Wspomnieliśmy na początku rozważań o drzewach wyszukiwań binarnych BST o głównym celu, do realizacji którego ta ciekawa struktura jest najczęściej stosowana. Otóż w drzewie BST możliwe jest szybkie wyszukanie wierzchołka o danym kluczu (na podstawie którego określane są relacje większy/mniejszy między wierzchołkami).

Od funkcji wyszukującej wierzchołki w drzewie BST najczęściej wymaga się, by pokazała „miejsce” w drzewie, w którym się ów węzeł znajduje (chyba, że nie ma go w tym drzewie).

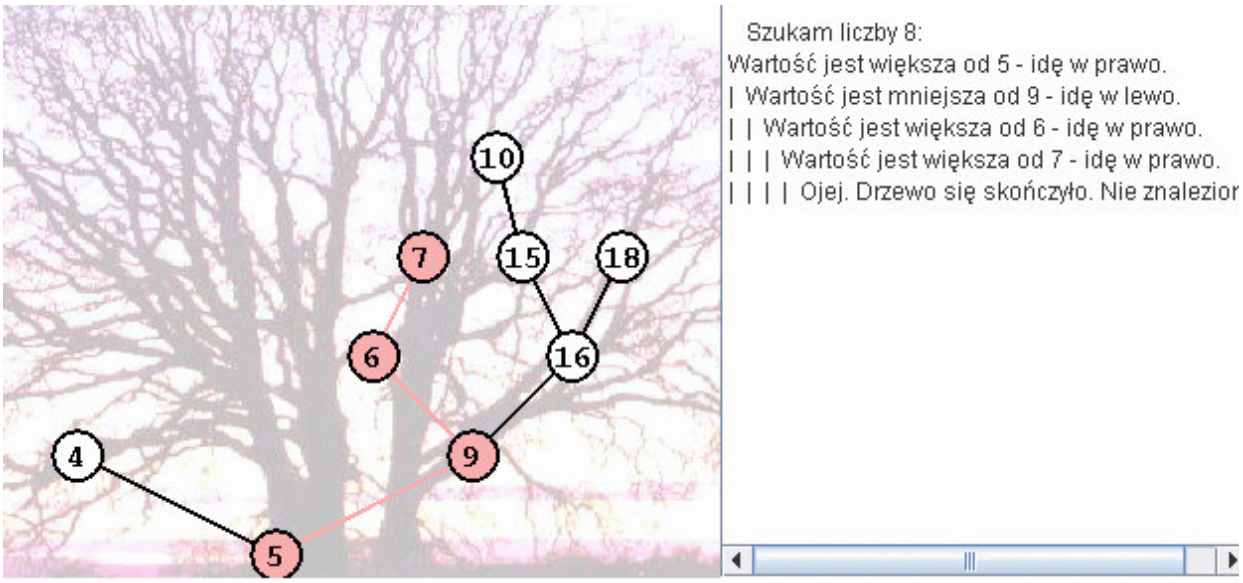
```
Twezel*  ZnajdzWezel (Twezel* wezel, int wartosc)  {
 // Funkcja zwraca wskaźnik do węzła zawierającego „wartosc”, jeśli taki węzeł znajduje się
 // w drzewie, w przeciwnym razie zwraca wartość NULL
 if (wezel == NULL || wartość == wezel->Dane)
          return wezel;

 if (wartosc < wezel->Dane)
          return ZnajdzWezel (wezel->Lewy, wartosc);
 else
          return ZnajdzWezel (wezel->Prawy, wartosc);
}
```

Poniżej prezentujemy dwa możliwe rezultaty działania funkcji ZnajdzWezel:

- gdy znaleziono poszukiwany węzeł (o wartości 15)




- gdy w drzewie nie ma poszukiwanego wierzchołka (o wartości 8)




Funkcja *ZnajdzWezel* rozpoczyna poszukiwanie od korzenia drzewa i porównuje z nim (z wartością zapisaną w korzeniu) poszukiwaną wartość. Jeśli są równe, to poszukiwanie się kończy, w przeciwnym razie wybieramy poddrzewo (lewe lub prawe) w zależności od tego, czy poszukiwana wartość była mniejsza, czy większa od wartości korzenia. Algorytm jest rekurencyjnie wywoływany dla kolejnych lewych bądź prawych poddrzew i przesuwa się w stronę liści. Jeśli okaże się, że dotarliśmy już do jednego z liści, który nie posiada szukanej wartości, to wówczas algorytm kończy działanie z negatywnym wynikiem wyszukiwania. Złożoność czasowa algorytmu *ZnajdzWezel* jest rzędu O(h), gdzie h jest wysokością drzewa.

Nie wspomnieliśmy jeszcze o sposobie usuwania węzła drzewa BST, jak i całego drzewa przeszukiwania binarnego. Zacznijmy może od tej drugiej operacji. Otóż podczas usuwania kolejnych węzłów drzewa musi być cały czas zachowana prawidłowa struktura drzewa wraz z poprawnie ustawionymi wskaźnikami w węzłach. Zatem usuwanie drzewa rozpoczynamy od „likwidowania” liści (w naszym kodzie programu usuwane są liście znajdujące się skrajnie w lewej części drzewa), a kończymy na usunięciu liścia, który jest zarazem korzeniem drzewa. W celu utrzymania struktury drzewa, tuż przed usunięciem konkretnego liścia następuje ustawienie wskaźnika jego ojca (wskazującego do tej pory na ten liść) na wartość NULL.

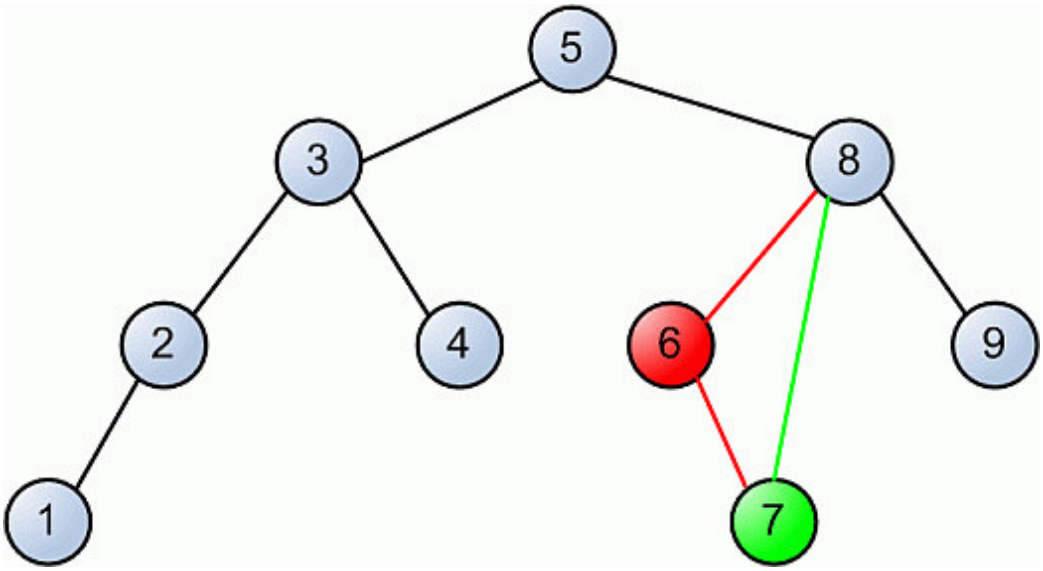
```
void UsunDrzewo (Twezel *&wezel) {
 Twezel *pom;
 if (wezel != NULL)  {
            UsunDrzewo (wezel -> Lewy);
            UsunDrzewo (wezel -> Prawy);
            pom=wezel;
        wezel=NULL;
        delete pom;
 }
}
```

Skoro już wiemy, jak poprawnie usuwa się całe drzewo, to koniecznie trzeba poznać, w jaki sposób usuwany jest pojedynczy węzeł. Problem ten możemy rozbić na kilka przypadków:

1. w drzewie BST nie ma węzła o żądanej wartości klucza (czyli pola struktury węzła)
2. usuwany węzeł ma co najwyżej jednego syna (obejmuje przypadek, gdy węzeł jest liściem)
3. usuwany węzeł ma dwóch synów

W pierwszym przypadku z wiadomych względów algorytm nie usuwa żadnego wierzchołka, struktura drzewa pozostaje identyczna.

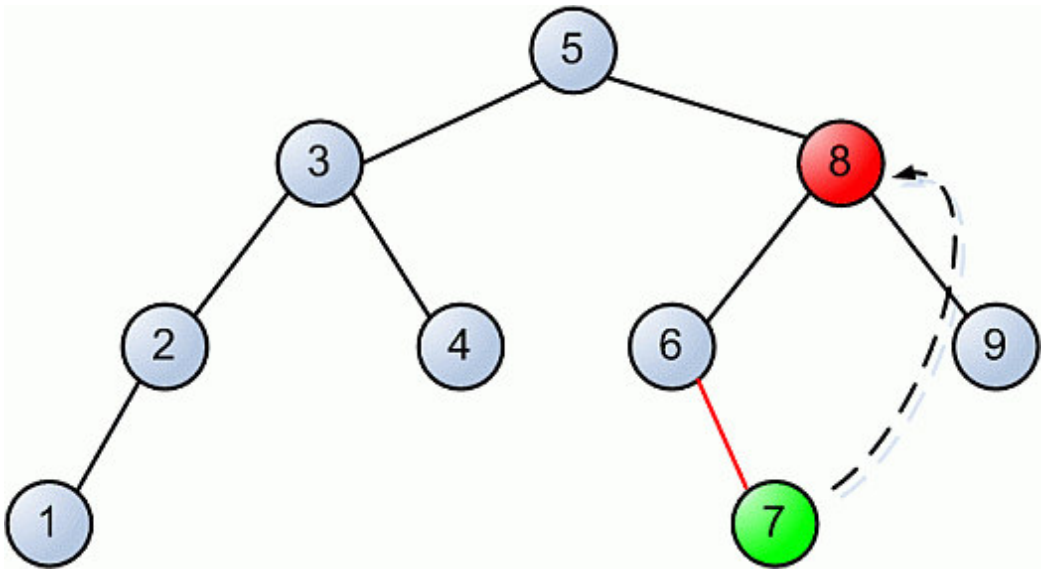
W drugim przypadku, ponieważ nie ma konfliktu między dwoma synami usuwanego węzła, wskaźnikowi wskazującemu na usuwany węzeł przypisywana jest wartość NULL (gdy usuwamy jeden z liści drzewa) lub adres syna usuwanego węzła.



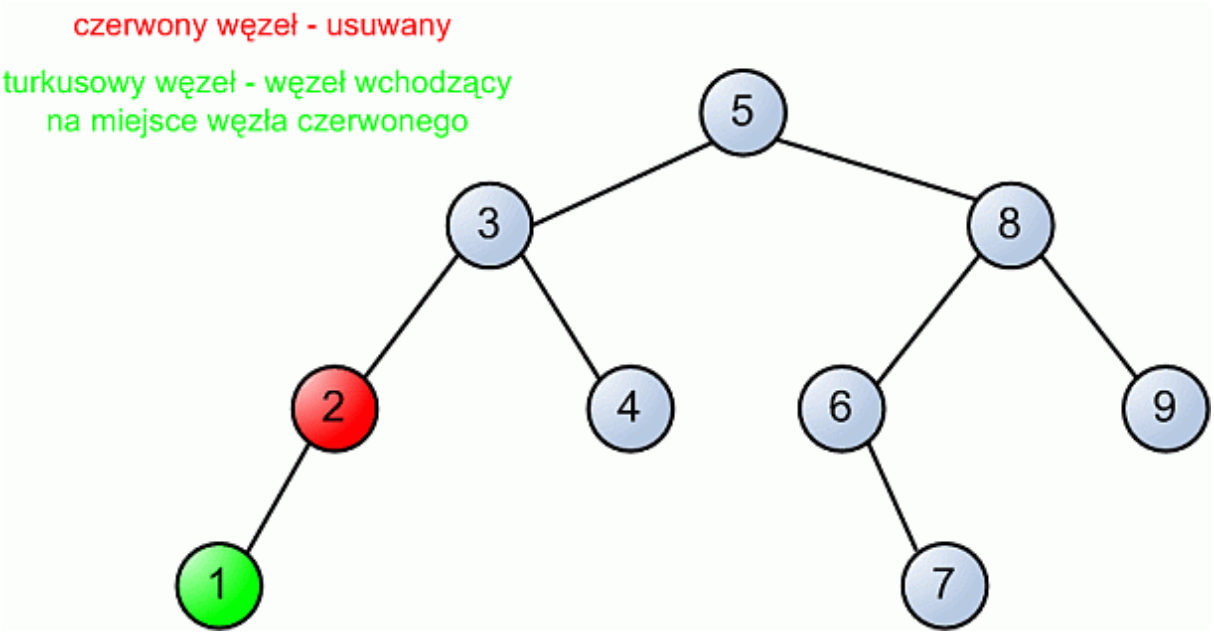

Natomiast w trzecim przypadku, gdy jest usuwany węzeł posiadający obu synów, to istnieją dwa równorzędne rozwiązania:

- usuwany węzeł zostaje zastąpiony przez skrajnie **prawy** węzeł jego lewego poddrzewa (węzeł ten posiada największą wartość z całego lewego poddrzewa)
- usuwany węzeł zostaje zastąpiony przez skrajnie **lewy** węzeł jego prawego poddrzewa (węzeł ten posiada najmniejszą wartość z całego prawego poddrzewa)

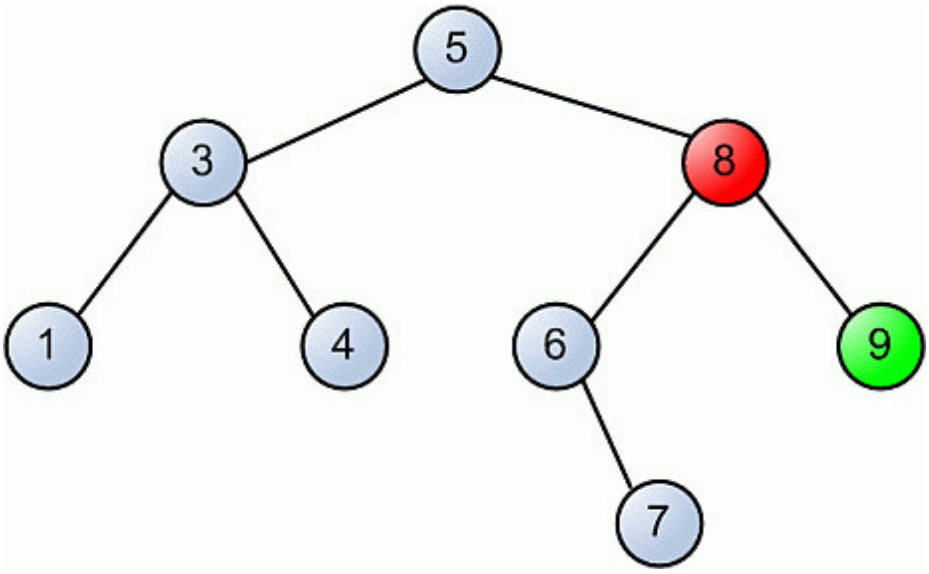
Oprócz zastąpienia wierzchołka usuwanego węzłem skrajnym, następuje także podpięcie potomka węzła skrajnego pod ojca tego węzła (oczywiście pod warunkiem, że węzeł skrajny miał syna).



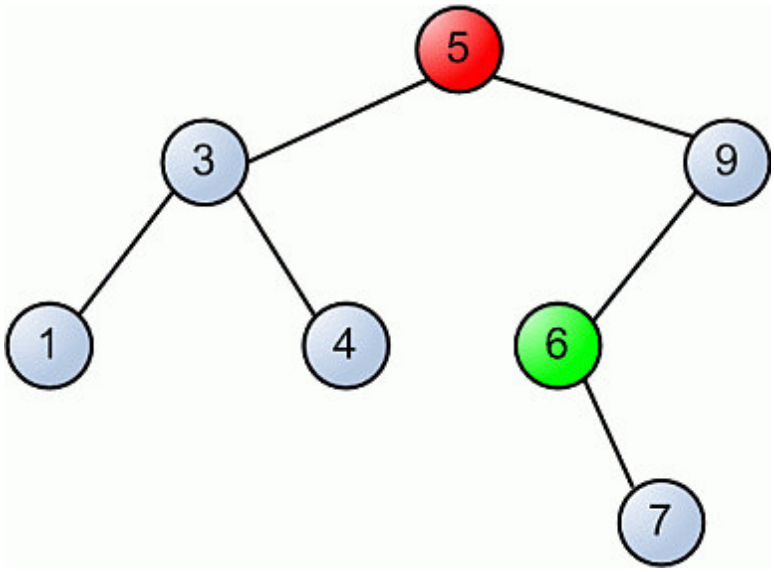

Na koniec przedstawiamy króciutki przykład podsumowujący sposób usuwania węzłów z drzewa. Na początek decydujemy się na usunięcie wierzchołka o wartości 2:



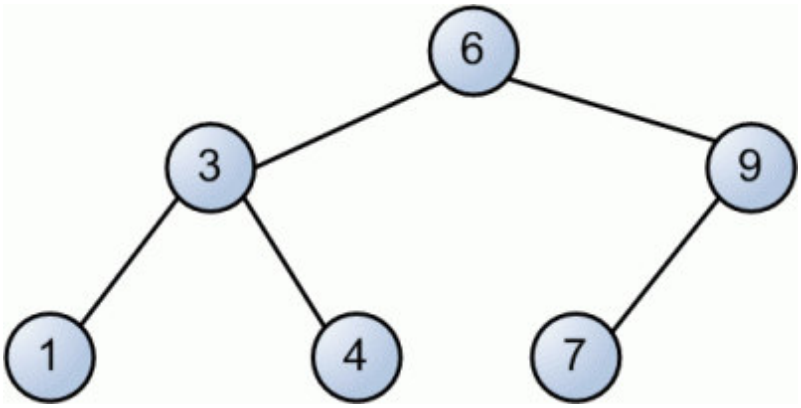

Ponieważ węzeł „2” posiada tylko jednego syna - potomek ten staje się synem swojego wcześniejszego dziadka.




W kolejnym kroku postanawiamy usunąć węzeł o wartości 8. Tym razem zachodzi trzeci przypadek – ma on dwóch synów. Z powodu prostszego prawego poddrzewa wybieramy na zastępstwo skrajnie lewy wierzchołek prawego poddrzewa węzła „8”.




W ostatnim kroku decydujemy się na usunięcie samego korzenia drzewa. Będziemy konsekwentni i ponownie wybierzemy skrajnie lewy wierzchołek prawego poddrzewa na miejsce usuwanego korzenia. Węzłem zastępującym jest wierzchołek o wartości 6



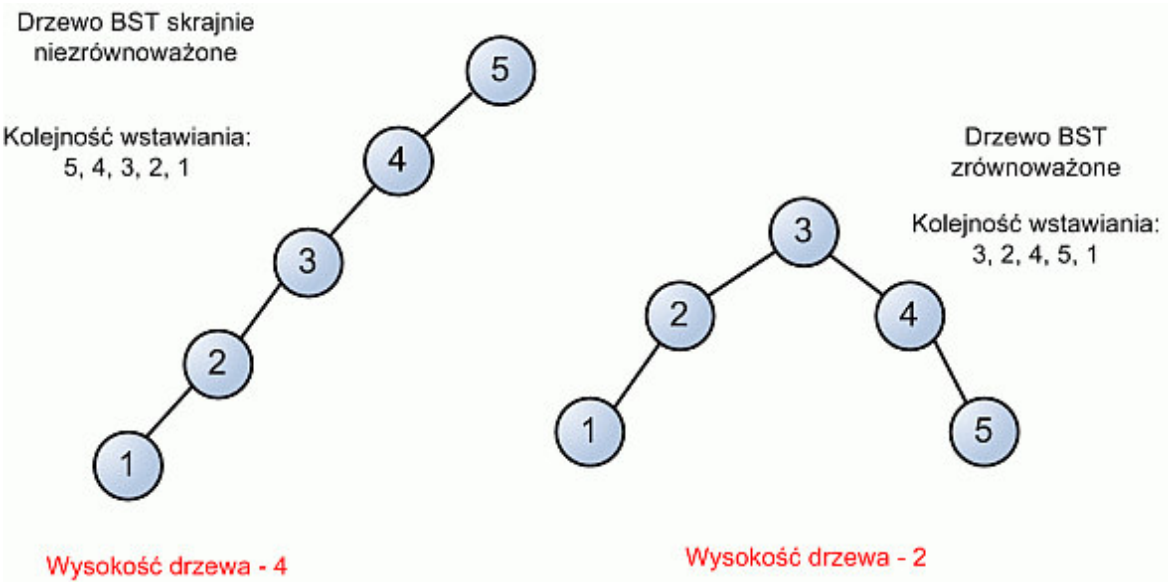

Potomek węzła „6"został podpięty pod jego dotychczasowego ojca „6" – węzeł o wartości 7 stał się lewym synem wierzchołka „9". Mamy nadzieję, że powyższe rysunki przedstawiające usuwanie węzłów okazały się pomocne w poprawnym zrozumieniu idei usuwania węzłów.

Algorytmy usuwania węzłów z drzewa BST, podobnie jak i metody wyszukiwania oraz dodawania wierzchołków, wykonują się w dobrym czasie **O(h)**, gdzie h oznacza wysokość drzewa.

## Drzewa AVL

W poprzednim podrozdziale dowiedzieliście się o drzewach przeszukiwania binarnego. Dzięki zastosowaniu drzew BST można w krótkim czasie wykonać operacje szukania elementów w zbiorze, dodawania usuwania z niego elementów. Działają one tym szybciej, im mniejszą wysokość posiada drzewo BST. Natomiast wraz ze wzrostem wartości h wysokości drzewa w stosunku do całkowitej liczby jego wierzchołków, opłacalność stosowania klasycznych drzew wyszukiwań binarnych stopniowo maleje – w skrajnym przypadku takie drzewo jest tyle samo warte, co zwykła lista. Dlatego też w pewnym momencie nastąpił silny rozwój algorytmów bazujących na drzewach BST, ale posiadających dodatkową, jakże cenną cechę: potrafiły one stale utrzymywać drzewo w postaci **zrównoważonej** (inaczej: **wyważonej**) – to znaczy takiej, która zapewnia optymalny stosunek wysokości drzewa do liczby węzłów.

Poniżej prezentujemy przykład, w którym jak na dłoni widać opisany problem:




Przy pesymistycznej kolejności wstawiania węzłów do drzewa (np. gdy wstawiane wartości są posortowane malejąco lub rosnąco) może powstać drzewo o mało zachęcającej strukturze. Jak widzimy na powyższym rysunku, mimo że wstawiliśmy tylko 5 elementów, to wysokość pierwszego drzewa jest dwukrotnie większa niż drugiego.

Najbardziej znanymi oraz najczęściej użytkowanymi w praktyce strukturami, na których stosowane są algorytmy o wspomnianej własności, są:

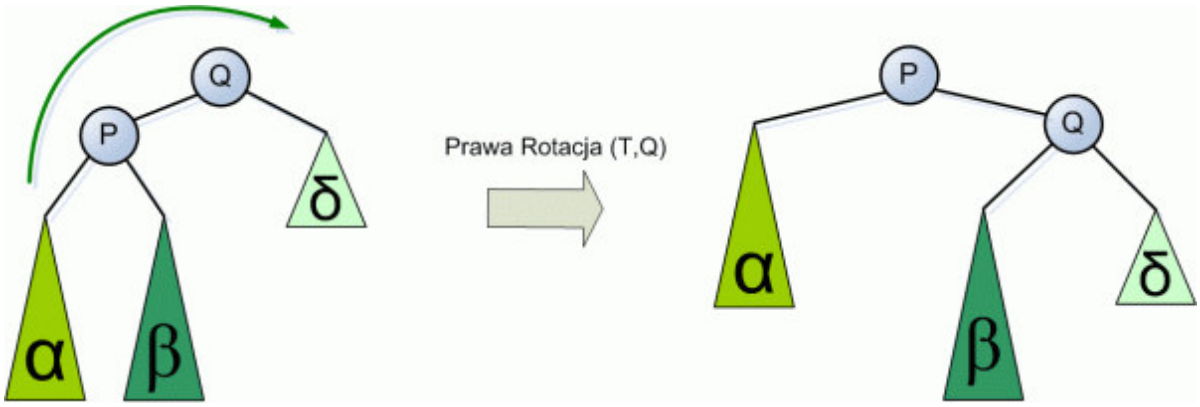
- drzewa **AVL**
- drzewa **czerwono-czarne.**

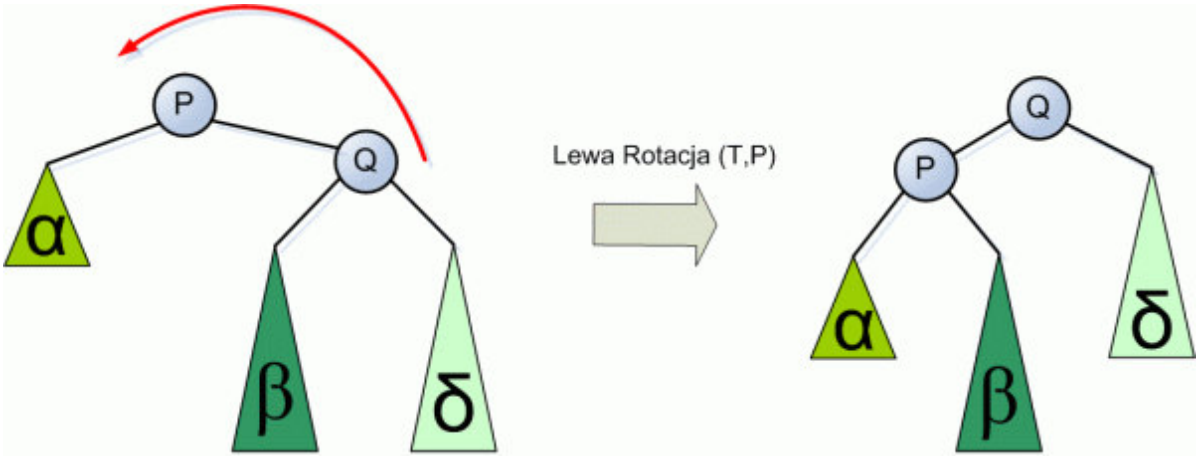
W tym rozdziale omówimy drzewa AVL, drzewami czerwono-czarnymi zajmiemy się w następnym rozdziale tej lekcji.

## Rotacja i wskaźnik wyważenia

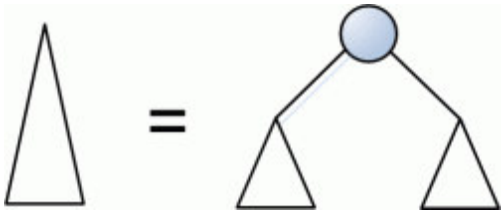
Drzewo AVL jest zrównoważonym drzewem wyszukiwań binarnych BST. Pomysłodawcami tej modyfikacji drzew BST byli dwaj rosyjscy matematycy **A**delson-**V**elskij oraz **L**andis na początku lat 60-tych ubiegłego stulecia. Drzewa AVL są równoważone w taki sposób, by różnica wysokości lewego i prawego poddrzewa dowolnego wynosiła nie więcej niż 1.

We wszystkich algorytmach równoważących drzewa wyszukiwań binarnych (nie tylko w algorytmie AVL) stosowana jest operacja **rotacji**. Zasadę lewej i prawej rotacji, wykonywanej na drzewie T względem wybranego węzła, obrazują poniższe rysunki; lewa rotacja jest tu pokazana względem węzła P, zaś prawa rotacja względem węzła Q. Po wykonaniu rotacji w prawo względem Q można wykonać rotację w lewo wględem P, przez co powrócimy do stanu początkowego:






Symbole *α, β, δ* w ogólnym przypadku oznaczają poddrzewa o dowolnej wielkości. Gdybyśmy jednak w sytuacji jak wyżej założyli, że:




to operacje rotacji pokazane na rysunkach zamieniają drzewo niewyważone (z lewej) w wyważone (z prawej strony rysunku) i po to właśnie są one stosowane.

Jak widzimy, główną czynnością podczas rotacji jest zmiana relacji między węzłami P oraz Q. Jeśli wartość w węźle P jest mniejsza niż wartość w węźle Q, to są dwie możliwości:

- albo Q jest prawym synem węzła P
- albo P jest lewym synem węzła Q.

Operacja rotacji zmienia jedną z tych zależności w drugą. Oprócz tego "przepinane" jest poddrzewo *β*, które może być lewym poddrzewem wierzchołka Q lub prawym poddrzewem wierzchołka P (wartości w poddrzewie *β* są większe od wartości w P oraz mniejsze od wartości w Q).

Po wykonaniu rotacji nadal dla każdego węzła (a więc i dla zaznaczonych na rysunku węzłów P i Q) wartości w lewym poddrzewie są mniejsze niż w wartość w węźle, natomiast w prawym poddrzewie – większe, czyli zachowany jest porządek **inorder**. Wykonanie rotacji w sposób istotny pomaga więc zrównoważyć drzewo, nie niszcząc przy tym prawidłowej struktury drzewa BST.

Tak więc:

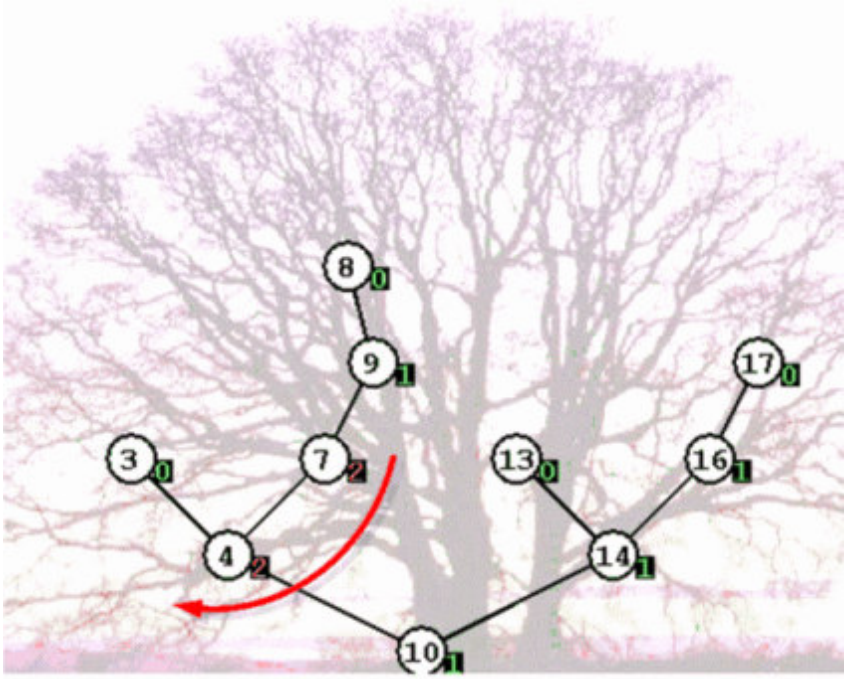
- **lewa** rotacja względem węzła polega na tym, że prawy syn tego węzła (nie może być on równy NULL) wchodzi na jego miejsce, przesuwając ojca **w lewo** - węzeł-ojciec staje się teraz lewym synem swojego dotychczasowego syna.
- **prawa** rotacja względem węzła polega na tym, że lewy syn tego węzła (nie może być on równy NULL) wchodzi na jego miejsce, przesuwając ojca **w prawo** - węzeł-ojciec staje się teraz prawym synem swojego dotychczasowego syna.

Ponieważ przesunięcie związane jest z obrotem powodującym uniesienie lub opuszczenie poddrzewa (co może zmniejszyć niewyważenie) - dlatego mówimy o rotacji. W literaturze możemy zresztą czasem spotkać nazewnictwo odwrotne, jeśli chodzi o lewe i prawe rotacje, więc nie zdziwcie się, jeśli nasza lewa rotacja będzie nazywana prawą i na odwrót.

Poniżej prezentujemy kod rotacji w lewo oraz w prawo, wynikający bezpośrednio z przedstawionych reguł rotacji.

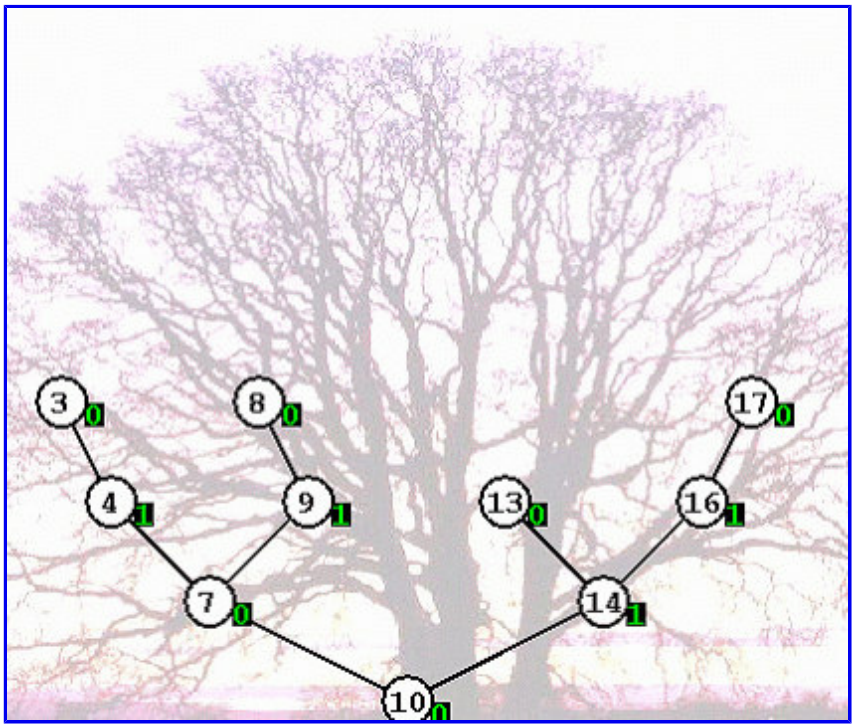
```
// rotacja w lewo względem P
if (P->Prawy == Q)  {
      P-> Prawy = Q-> Lewy;
      Q-> Lewy = P;
      P = Q;
}


// rotacja w prawo względem Q
if (Q->Lewy == P)   {
     Q->Lewy = P->Prawy;
     P-> Prawy = Q;
     Q = P;
}
```

Aby uzmysłowić sobie dokładniej istotę rotacji, przyjrzyjmy się schematowi drzewa pochodzącego z apletu, który znamy już z poprzedniego rozdziału. Tym razem przy węzłach widnieją wskaźniki wyważenia (bezwzględna różnica wysokości poddrzew), które uaktywnia się wybierając opcję "wagi" nad oknem apletu. Dwa wierzchołki, 4 oraz 7 są korzeniami drzew niewyważonych (wskaźniki równe 2).




Tym samym drzewo o powyższej strukturze jest niezrównoważone. Dokonuje się wówczas rotacji na tych poddrzewach, w których nie jest zachowane wyważenie. Zasada lewej i prawej rotacji pozostaje ta sama, mimo że drzewo w naszym aplecie "rośnie" odwrotnie niż w przykładach rysunkowych. Dla przykładu jak wyżej, jeśli węzeł 7 wejdzie na miejsce ojca (węzła 4) i przesunie go w lewo - będzie to rotacja w **lewo** względem wierzchołka 4.

W tym przypadku wykonanie tej jednej rotacji kończy się zrównoważeniem drzewa:




Jak widzimy, po przeprowadzeniu jednorazowej rotacji wszystkie węzły mają zadowalające wskaźniki, dopuszczalne dla drzewa wyważonego. Niestety, w najgorszym przypadku trzeba wykonać liczbę rotacji rzędu **lg(n)**, by całkowicie zrównoważyć drzewo składające się z n wierzchołków.

Wykonywanie rotacji względem wybranych węzłów możecie dowoli poćwiczyć uruchamiając aplet poprzez kliknięcie w powyższy obrazek. Aby wykonać rotację, należy prawym przyciskiem myszki wybrać węzeł i wówczas rotacja wykona się względem ojca tego wybranego węzła - wybrany węzeł wejdzie na jego miejsce, przesuwając ojca w lewo lub w prawo. W aplecie równoważenie drzewa nie wykonuje się automatycznie, co pozwala na ćwiczenia związane z rotacjami - trzeba umiejętnie dobierać rotacje, rozumiejąc zasadę ich działania, by stopniowo doprowadzić drzewo do równowagi. Nieprawidłowo wykonana rotacja może pogorszyć wyważenie drzewa.

W oryginalnym algorytmie budowania drzew AVL współczynnik (wskaźnik) wyważenia przechowywany w węźle jest różnicą między wysokością jego lewego oraz prawego poddrzewa (którą wysokość od której odejmujemy, to już zależy od przyjętej konwencji). Tym samym wartość tego współczynnika może być dodatnia, ujemna lub też równa 0 (w przypadku równej wysokości poddrzew). Natomiast dopuszczalna wartość wskaźnika, która nie wymaga interwencji w strukturze drzewa, to **–1, 0, 1**.

Drzewa AVL posiadają jeszcze jedną ważną cechę dotyczącą wyważenia – mianowicie są one drzewami **dokładnie wyważonymi**. Oznacza to, że dla każdego węzła liczba węzłów w jego lewym oraz prawym poddrzewie może różnić się tylko o jeden.

Jak do tej pory, omijaliśmy kwestie implementacyjne drzew AVL. Powinniśmy więc w tym momencie nadrobić zaległości. Otóż sposób przechowywania informacji o wyważeniu drzewa w **definicji węzła** jest następujący:

```
struct Twezel {
    int dane;
    Twezel  *lewy, *prawy;
    int  ww;      //  wskaźnik wyważenia węzła
}
```

Definicja węzła drzewa AVL różni się od analogicznej definicji węzła w klasycznym drzewie BST tym, że dodano pole struktury przechowujące całkowite wartości liczbowe współczynnika wyważenia. Oto cała modyfikacja.

## Operacje na drzewach AVL

Kluczowymi operacjami, przy których z drzewem AVL mogą dziać się „niepokojące” rzeczy, są operacje wstawiania węzła oraz usuwania węzła z drzewa. Wówczas może zostać zachwiana równowaga drzew i w sposób natychmiastowy trzeba to naprawić.

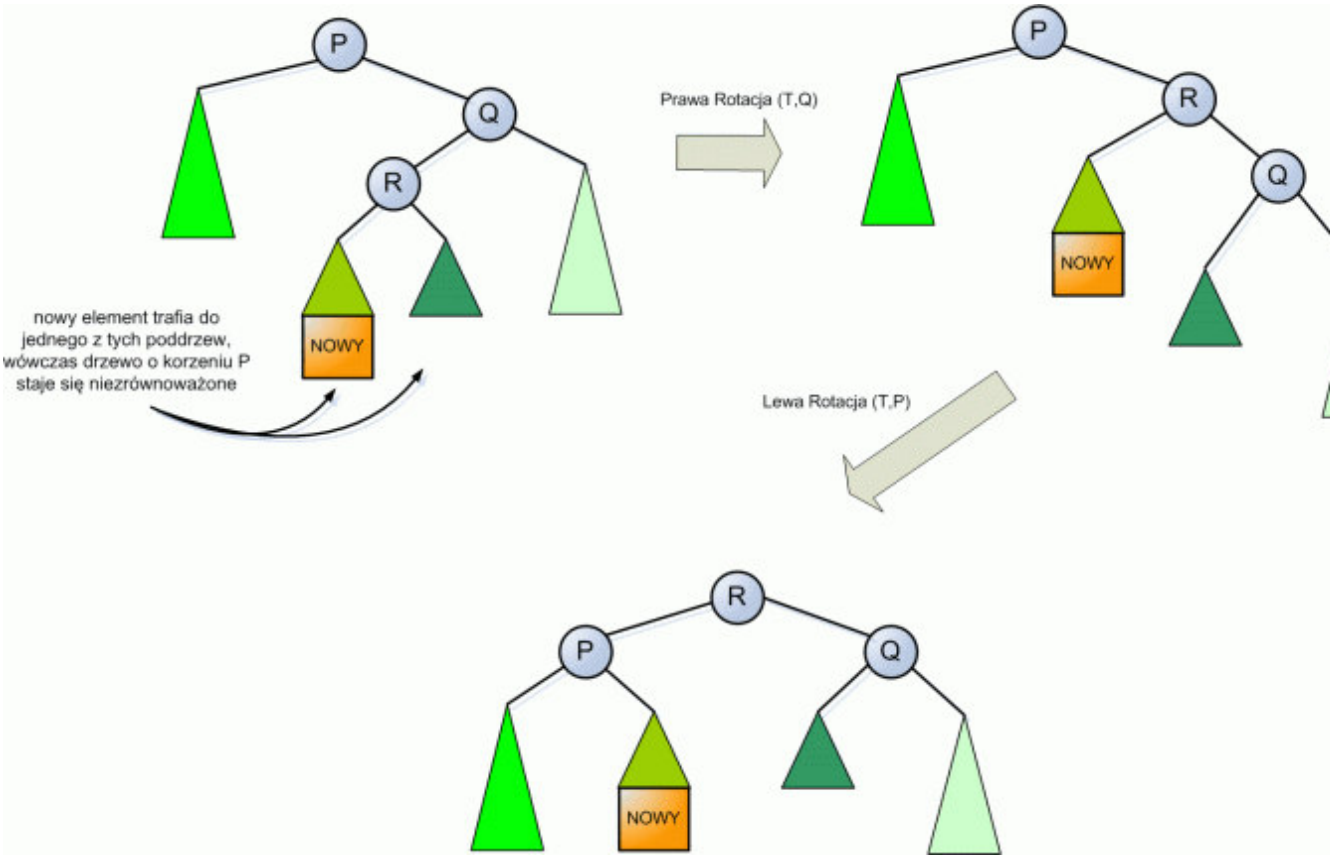
Omówmy **proces wstawiania wierzchołka** do drzewa AVL. Składa się on z kilku zasadniczych kroków:

1. Na początku przeszukujemy drzewo w celu znalezienia odpowiedniego miejsca do wstawienia – czynność ta jest wykonywana dokładnie tak samo, jak wstawianie węzła do drzewa BST.
2. Następnie algorytm zaczyna wracać od wstawionego węzła w stronę korzenia po ścieżce, którą na początku wędrował w dół. W momencie przechodzenie przez kolejne wierzchołki aktualizowane są współczynniki wyważenia ww (mogły ulec zmianie poprzez dodanie węzła), po czym następuje odpowiednia reakcja na nową wartość współczynnika:
    - Jeśli **ww** przyjęło wartość **0**, oznacza to, że poddrzewo to zostało idealnie zrównoważone (a wysokość poddrzewa o korzeniu z ww=0 nie wzrosła). Zatem drzewa zawierające wierzchołek o wartości ww=0 nie mogły przestać być wyważone. Algorytm wstawiania kończy działanie
    - Jeśli **ww** przyjęło wartość **1 lub –1** – świadczy to o tym, że poddrzewo o korzeniu w badanym wierzchołku utrzymało wyważenie, ale jego wysokość **zwiększyła się** poprzez dodanie nowego wierzchołka. Z tą informacją algorytm wędruje dalej w stronę korzenia całego drzewa.
    - Jeśli **ww** przyjęło wartość **2 lub –2** – drzewo straciło wyważenie. Niezbędna jest rotacja (czasem podwójna) względem badanego wierzchołka. Po wykonaniu rotacji drzewu zostanie przywrócone zrównoważenie.

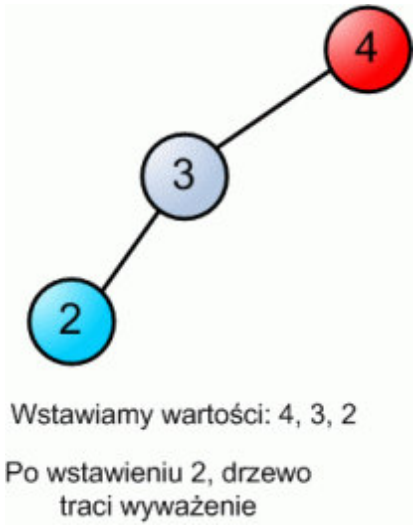
To, czy rotacja będzie **pojedyncza**, czy **podwójna** zależy od miejsca wstawienia nowego węzła. Otóż:

- Jeśli wstawiamy węzeł do prawego poddrzewa prawego syna (lub do lewego poddrzewa lewego syna) i konieczna będzie rotacja (z powodu utraty wyważenia drzewa), wówczas należy wykonać pojedynczą rotację - taką, jaka widnieje na schemacie ogólnym przedstawiającym ideę rotacji
- Natomiast jeżeli wstawiony zostanie węzeł do lewego poddrzewa prawego syna (lub do prawego poddrzewa lewego syna) – niezbędna będzie podwójna rotacja. Przyjmijmy, że wystąpił pierwszy wariant – nowy węzeł znalazł się w lewym poddrzewie prawego syna. Wówczas wykonujemy dwie części składające się na podwójną rotację (oznaczmy ojca , dla którego ww= 2 lub -2, jako P, a jego syna jako Q):
    - wpierw wykonujemy prawą rotację względem Q
    - następnie wykonujemy lewą rotację względem P

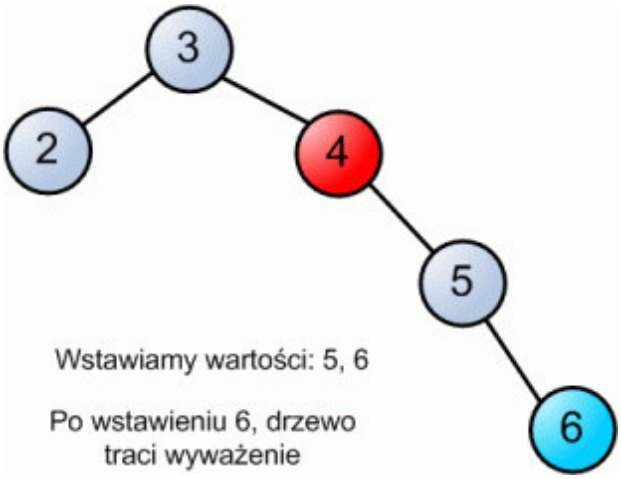
  Po tak przeprowadzonej podwójnej rotacji drzewo znów posiada prawidłową strukturę AVL. Ideę rotacji podwójnej przedstawia schematycznie poniższy rysunek:




Na zakończenie proponujemy prosty przykład liczbowy. Do pustego drzewa wstawiamy wartości 4, 3, 2.

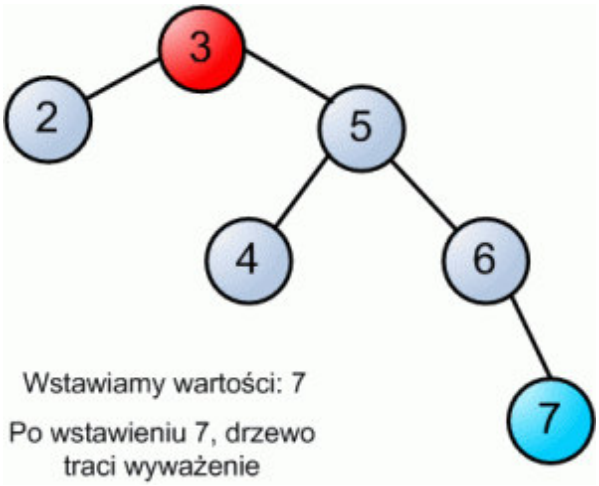



Po wstawieniu tej trzeciej wartości, drzewo staje się niezrównoważone – należy wykonać prawą rotację względem „4" (na niebiesko oznaczone będą węzły, których wstawienie zburzyło wyważenie drzewa, z kolei na czerwono będą zaznaczone węzły, na których należy przeprowadzić rotację):

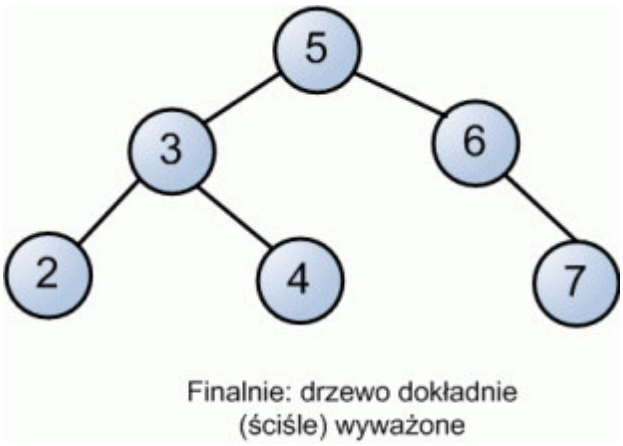



Po wykonaniu rotacji drzewo zostało zrównoważone, ale teraz wstawiamy kolejno wartości 5 oraz 6. Wstawienie tej drugiej

wartości wymaga, by przeprowadzona została lewa rotacja względem „4”:




W ostatnim kroku dodajemy element „7”. Niestety, i tym razem musimy wykonać rotację – dokładnie lewą rotację względem korzenia „3”.




Ostatecznie drzewo AVL zostało dokładnie wyważone. Resztę ćwiczeń proponujemy wykonać z użyciem apletu.

W sposób podobny do dodawania węzła działa **algorytm usuwania wierzchołka** z drzewa AVL:

1. Na początku przeszukujemy drzewo w celu znalezienia miejsca, w którym znajduje się węzeł do usunięcia
2. Następnie wykonywane są czynności związane z usuwaniem wierzchołka – analogiczne do usuwania węzła z drzewa BST.
3. Następnie algorytm zaczyna wracać od usuniętego węzła w stronę korzenia po ścieżce, którą na początku wędrował w dół. W momencie przechodzenie przez kolejne wierzchołki aktualizowane są współczynniki wyważenia ww (mogły ulec zmianie z powodu usunięcia węzła), po czym następuje odpowiednia reakcja na nową wartość współczynnika:
    a. Jeśli **ww** przyjęło wartość **1 lub -1**, oznacza to, że wysokość poddrzewa o korzeniu z ww= 1 lub -1 nie zmieniła się. Zatem drzewa zawierające wierzchołek o wartości ww= 1 lub -1 nie mogły przestać być wyważone. Algorytm usuwania kończy działanie.
    b. Jeśli **ww** przyjęło wartość **0** – świadczy to o tym, że poddrzewo o korzeniu w badanym wierzchołku utrzymało wyważenie, ale jego wysokość **zmniejszyła się** poprzez usunięcie wierzchołka. Z tą informacją algorytm wędruje dalej w stronę korzenia całego drzewa.
    c. Jeśli **ww** przyjęło wartość **2 lub –2** – drzewo straciło wyważenie. Niezbędna jest rotacja względem badanego wierzchołka w celu przywrócenia wyważenia drzewa. Jednakże w porównaniu z dodawaniem wierzchołka, jednokrotna rotacja nie musi być wystarczająca. W najgorszym układzie może być konieczne przeprowadzenie kolejnych **rotacji we wszystkich** wierzchołkach znajdujących się na drodze powrotu do korzenia.

Operacje wstawiania, usuwania, a także wyszukiwania wierzchołka w drzewie zrównoważonym AVL posiadają złożoność obliczeniową O(lg n), gdzie n jest liczbą wierzchołków w drzewie. Mimo, że w porównaniu z klasycznymi drzewami BST czas modyfikacji drzewa trochę wzrósł, to zyskano za cenę tego ogromną zaletę: czas wyszukiwania elementu w pesymistycznym przypadku nadal wynosi tylko **O(lg n)**, a w przypadku drzew BST może on drastycznie wzrosnąć aż do O(n).

## Drzewa czerwono-czarne

W tym, ostatnim już, podrozdziale wracamy do wspomnianej tylko grupy drzew wyważonych, jakimi są drzewa czerwono-czarne (ang. **Red-Black Tree**, w skrócie **RBT**). Są one, razem z drzewami AVL, najpopularniejszymi drzewami binarnego wyszukiwania, w których stosuje się równoważenie drzewa.

Drzewa RBT zostały stworzone nieco później niż drzewa AVL, gdyż nastąpiło to w połowie lat 70-tych XX wieku i obecnie są coraz powszechniej używane w wielu prężnie rozwijających się dziedzinach informatyki (m.in. w geometrii obliczeniowej).

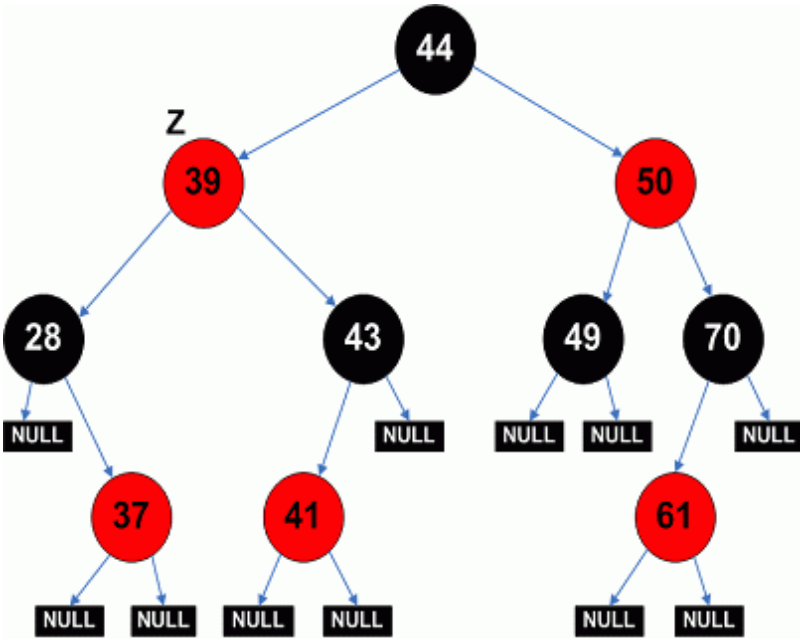
Każdy węzeł drzewa RBT zawiera dodatkową informację - **kolor**, który może być albo czerwony, albo czarny. Ten dodatkowy bit informacji w węźle pozwala na wykonywanie operacji równoważenia dzrewa, pełni więc tę samą rolę jak wskaźnik wyważenia w drzewie AVL.

Aby zwykłe drzewo wyszukiwań binarnych stało się drzewem czerwono-czarnym, musi spełnić następujące warunki:

---

- każdy węzeł jest **czerwony** lub **czarny**
- korzeń drzewa jest koloru czarnego
- każdy liść drzewa (NULL/Nil) jest koloru czarnego
- jeśli węzeł jest czerwony, to obaj jego synowie są czarni (tym samym **każdy czerwony węzeł ma czarnego ojca**)
- każda prosta ścieżka prowadząca z ustalonego wierzchołka do liścia ma tyle samo czarnych węzłów (jest to tzw. czarna głębokość węzła).

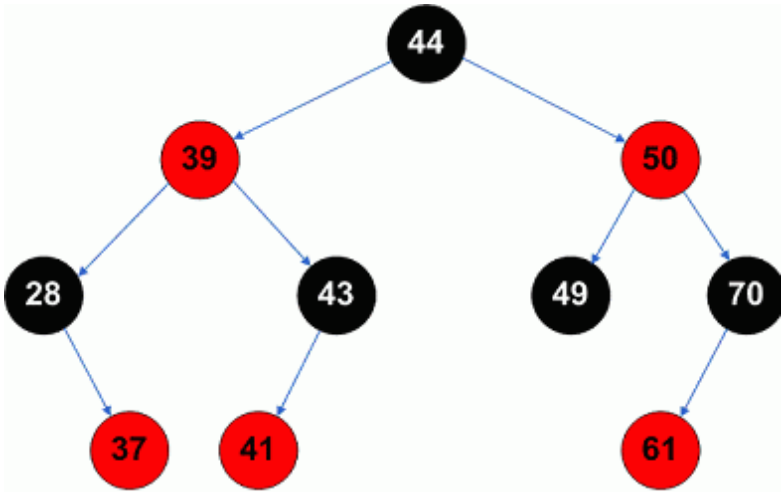
---




Poprzez ustalenie warunków na następstwo kolorów na ścieżkach, gwarantowane jest, że każda ścieżka jest co najwyżej dwukrotnie dłuższa niż dowolna inna ścieżka prowadząca z tego samego wierzchołka. Wynika to z faktu, że najkrótsza możliwa ścieżka zawiera kolejno same węzły czarne, a najdłuższa - czerwone i czarne na przemian. A ponieważ wiemy, że na każdej ścieżce musi być tyle samo węzłów czarnych, to czerwonych może być co najwyżej drugie tyle. Klasyfikuje to drzewa RBT do grupy **w przybliżeniu zrównoważonych**.

Jakie jeszcze wartościowe cechy posiadają drzewa RBT? Po pierwsze, można udowodnić, że wysokość drzewa czerwono-czarnego o n węzłach wewnętrznych (a zatem nie licząc liści NULL/Nil) wynosi co najwyżej 2 lg(n+1). Dzięki gwarancji maksymalnie takiej wartości wysokości drzewa, możemy zbudować metody wyszukiwania węzłów w drzewie RBT o złożoności **O(lg n)**, gdyż z podrozdziału o drzewach binarnego wyszukiwania wiemy, że operacje wyszukiwania w drzewie są liniowo zależne od wysokości drzewa. Algorytmy dodawania i usuwania węzłów do drzew RBT również są w stanie wykonywać się w czasie O(lg n).

Z ciekawostek warto dodać, że na ogół nie rysuje się liści drzewa RBT, gdyż wiadomo, że wszystkie liście są czarne i nie wnoszą żadnych istotnych informacji. Poniżej widnieje prostszy sposób narysowania drzewa czerwono-czarnego:




Porównajmy drzewa czerwono-czarne z drzewami AVL:

---

- przywrócenie własności RBT po wstawieniu/usunięciu węzła wymaga przeprowadzenia maksymalnie **dwóch rotacji**, dzięki czemu przy niekorzystnym układzie danych w drzewach RBT wykonywana jest znacznie mniejsza liczba rotacji niż w drzewach AVL (w których chcemy przywrócić wyważenie). Zatem drzewa RBT są bardziej przydatne, gdy często musimy węzeł z drzewa usuwać lub do niego dodawać
- z kolei drzewa AVL lepiej sprawdzają się w operacjach wyszukiwania (są zazwyczaj niższe, lepiej wyważone, ich wysokość nie przekracza wartości 1,44 lg(n) ).

---

Przyszedł moment na część najważniejszą, czyli opis sposobu, w jaki są dodawane i usuwane węzły z drzewa czerwono-czarnego. Podczas tych operacji zaburzona bywa prawidłowa struktura drzewa, przez co w celu jej przywrócenia należy zmienić kolory pewnych węzłów, a także "poprzepinać" niektóre połączenia między wierzchołkami (m.in. za pomocą poznanych już wcześniej rotacji).

Zdecydowaliśmy się przedstawić w tej lekcji opis jedynie dodawania wierzchołka do drzewa RBT. Metoda usuwania węzła z takiego drzewa jest równie złożona i dlatego też w celu uniknięcia przeładowania tej lekcji dosyć trudną wiedzą pozostawiamy ją dla wyłącznie dla chętnych. Wyczerpujący opis operacji modyfikujących drzewa RBT można znaleźć w książce autorstwa *Cormena, Leisersona, Rivesta, Steina, pt. "Wprowadzenie do algorytmów"*.

Opis algorytmu dodawania węzła wydaje się długi, dlatego zalecamy wnikliwe wczytanie się w niego, niemniej jednak niektóre elementy mogą wydawać się niejasne. Aby rozwiać powstałe wątpliwości, z pomocą przyjdą nam proste przykłady rysunkowe na końcu tego rozdziału.

Procedura **wstawWezel** (węzeł **z**, drzewo **RBT**)

```
A.  Wykonujemy zwykłą operację wstawiania węzła z jak do  drzewa BST
B.  Kolorujemy węzeł z na czerwono
C.  Wykonujemy pętlę while: dopóki węzeł z ma czerwonego ojca, następuje  przekolorowani
    i wykonanie  rotacji,  w sposób zależny od następujących  przypadków:
      1.    Ojciec węzła z jest prawym  synem swego ojca. Stryjem y węzła z jest wówczas
            (z definicji) lewy syn jego dziadka.
         1a. Stryj y istnieje i jest czerwony
         1b. Stryj y nie istnieje lub jest czarny i węzeł z jest lewym
                   synem swego ojca
         1c. Stryj y nie istnieje lub jest czarny i węzeł z jest prawym
                   synem swego ojca
      2.  Ojciec węzła z jest lewym  synem swego ojca. Stryjem y węzła z jest
          wówczas (z definicji) prawy syn jego dziadka.
         2a. Stryj y istnieje i jest czerwony
         2b. Stryj y nie istnieje lub jest czarny i węzeł z jest prawym
                   synem swego ojca
         2c. Stryj y nie istnieje lub jest czarny i węzeł z jest lewym
                   synem swego ojca
D.  Korzeń drzewa kolorujemy na czarno
```

Znamy już schemat działania algorytmu. Poniżej przedstawiamy instrukcje, jakie są wykonywane w zależności od poszczególnych przypadków zaistniałych w punkcie C:

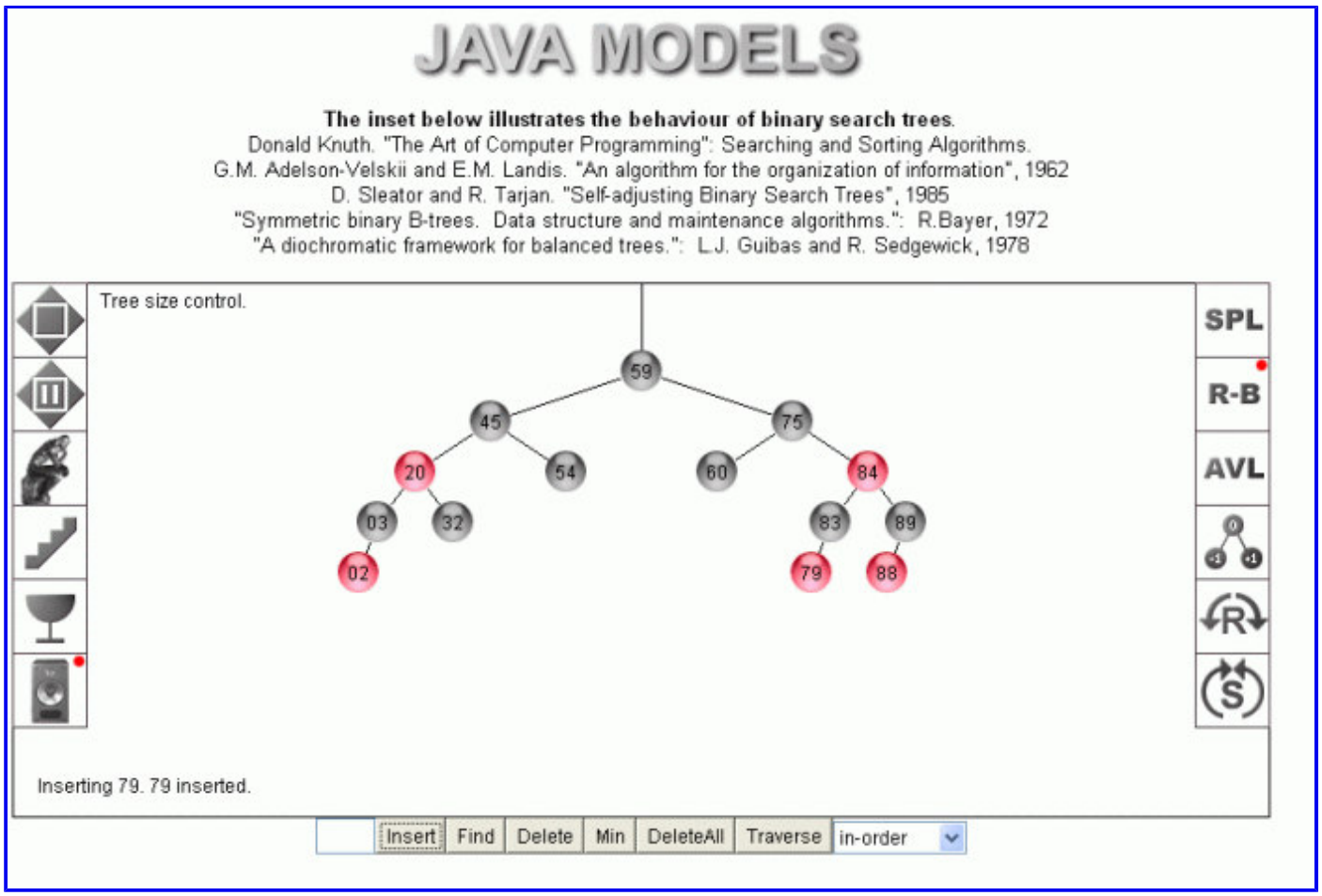
- **przypadek 1a i 2a**:
  - ojciec(z).color = black
  - stryj(z).color = black
  - dziadek(z).color = red
  - z = dziadek(z)

- **przypadek 1b**:
  - z = ojciec(z);
  - wykonaj prawą rotację względem z
  - wykonaj operacje dla Przypadku 1c
- **przypadek 1c**:
  - ojciec(z).color = black
  - dziadek(z).color = red
  - wykonaj lewa rotację względem dziadka(z)
- **przypadek 2b i 2c**: - realizuje się w sposób analogiczny do przypadków odpowiednio 1b i 1c, należy jedynie zamienić rotację prawą na lewą i na odwrót

Wykonanie pętli while (punkt C. algorytmu) zostaje powtórzone tylko wówczas, gdy zajdzie przypadek **1a** lub **2a** - wtedy węzeł **z** zostaje przesunięty o 2 poziomy w górę drzewa. Operacje dla przypadków **1b, 2b, 1c, 2c** wykonują się tylko raz (co potwierdza tezę, że do przywrócenia struktury drzewa czerwono-czarnego po wstawieniu węzła wystarczą maksymalnie dwie rotacje).

Warto zapamiętać, że dla efektywniejszej implementacji wszystkie liście NULL/Nil oraz ojca korzenia zastępuje się jednym wspólnym wartownikiem, który jest zwykłym węzłem koloru czarnego (inne parametry tego węzła nas nie interesują). W takiej sytuacji mamy zapewnione, że stryj zawsze istnieje.

Zachęcamy do odwiedzenia strony (kliknijcie w zrzut ekranu poniżej)
**http://webpages.ull.es/users/jriera/Docencia/AVL/AVL%20tree%20applet.htm**
zawierającej świetny aplet z zaimplementowanymi drzewami czerwono-czarnymi (i nie tylko), na których można dowoli poćwiczyć i poobserwować, w jaki sposób drzewo RBT (w aplecie oznaczone jako R-B) jest modyfikowane.

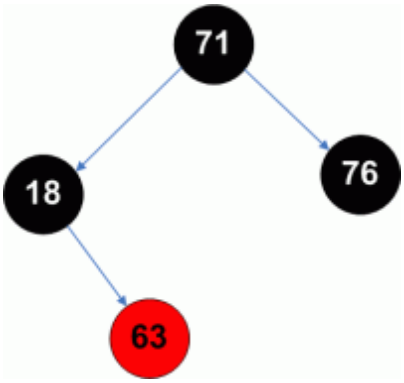



## Przykłady

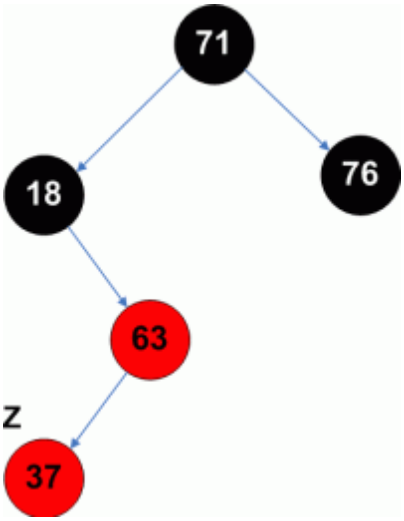
Niezależnie od ćwiczeń z apletem, do których zachęcamy, proponujemy jeszcze prześledzić wspólnie przykładowy przebieg modyfikacji drzewa po wstawieniu do niego nowego wierzchołka.

## Przykład 1.

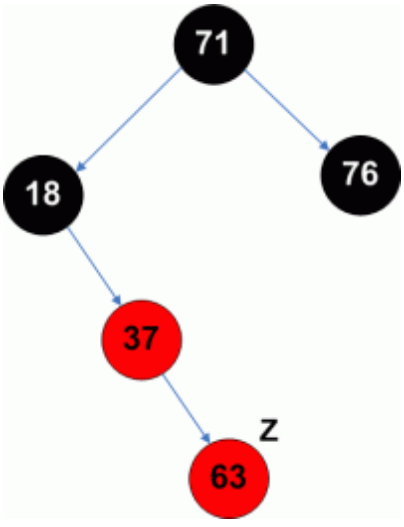
Pierwszy przykład jest krótki i prosty. Drzewo wejściowe RBT wygląda tak, jak na schemacie poniżej:




Następnie dodajemy do niego węzeł z kluczem 37 (i malujemy na czerwono):



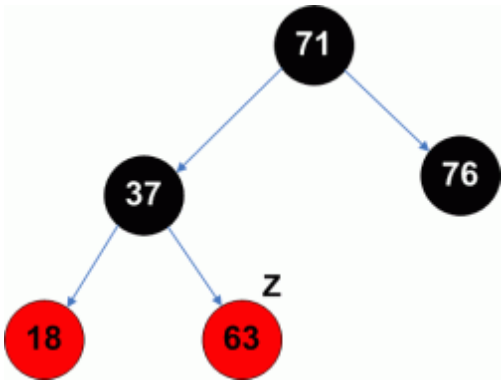

Węzeł 37 staje się węzłem **z**. Ponieważ ojciec naszego węzła **z** jest czerwony, drzewo trzeba poprawić. Jak widzimy na rysunku, węzeł **z** (37) jest lewym synem swojego ojca (63), natomiast ojciec węzła **z** jest prawym synem swojego ojca (czyli dziadka (18) węzła **z**). Zachodzi więc przypadek **1b**. Nowym węzłem **z** staje się ojciec dotychczasowego węzła **z** (czyli teraz będzie nim 63), a następnie wykonujemy względem niego prawą rotację. Po tych operacjach otrzymujemy poniższe drzewo:




Ojciec węzła **z** ponownie jest czerwony, czyli drzewo to nadal nie posiada wszystkich własności RBT. Wykonujemy teraz pozostałe kroki dla tego przypadku według **1c**:

- zmieniamy kolor ojca węzła **z** na czarny (chodzi o węzeł 37)
- kolor dziadka **z** zmieniamy na czerwony (węzeł 18)
- na koniec wykonujemy lewą rotację względem dziadka węzła **z**

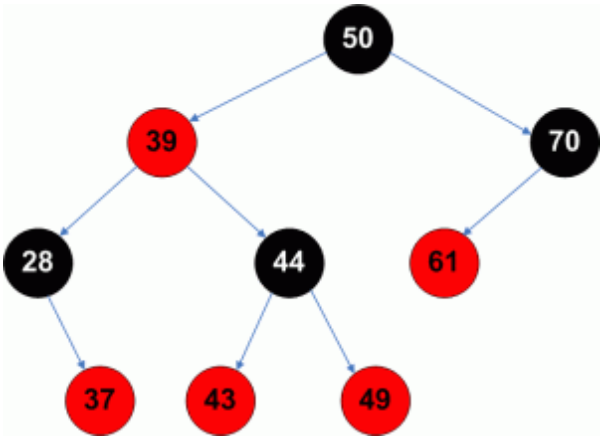
Po tych operacjach węzeł **z** ma wreszcie czarnego ojca. Otrzymujemy drzewo, które posiada już wszystkie własności czerwono-czarne:



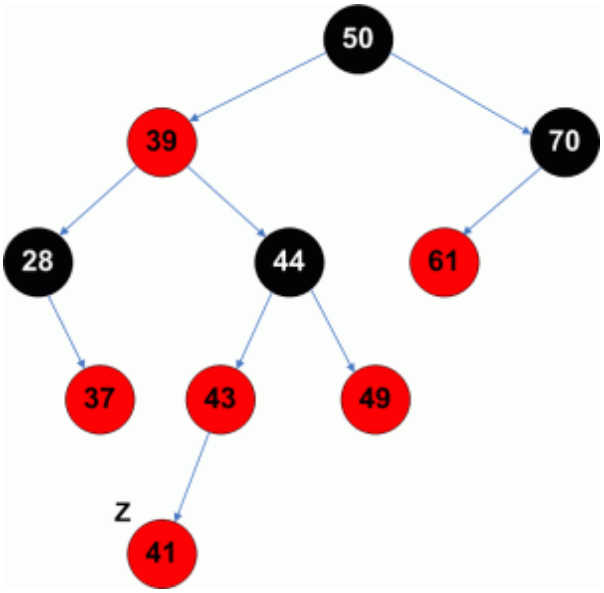

## Przykład 2.

To nie koniec przygód z drzewami RBT. Tym razem drugi przykład dotyczy większego drzewa, ale nie jest aż tak bardzo zawiły. Zachęcamy do prześledzenia, jak w tym przypadku będzie się zmieniało nasze drzewo.

Początkowe drzewo czerwono-czarne wygląda tak, jak widnieje na rysunku poniżej. Planujemy do drzewa dodać węzeł o kluczu 41.



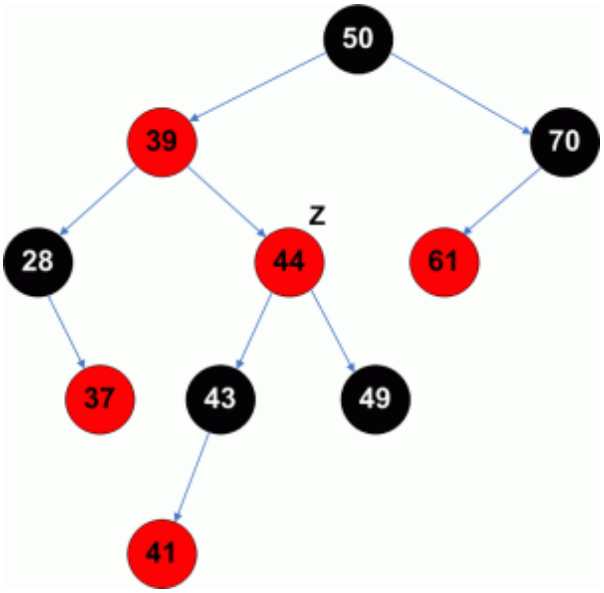

Dodajemy węzeł 41 (oznaczony jest on na rysunku symbolem z). Po wstawieniu w odpowiednie miejsce drzewa, został on pomalowany na kolor czerwony.




Węzeł 43 (rodzic 41-ego) jest czerwony, jest również lewym synem swego ojca, z kolei stryj węzła 41-ego również jest czerwony (chodzi o węzeł 49), a zatem zachodzi przypadek **2a**. Kolorujemy odpowiednio węzły:

- węzeł 43 na czarno
- węzeł 49 na czarno
- węzeł 44 na czerwono

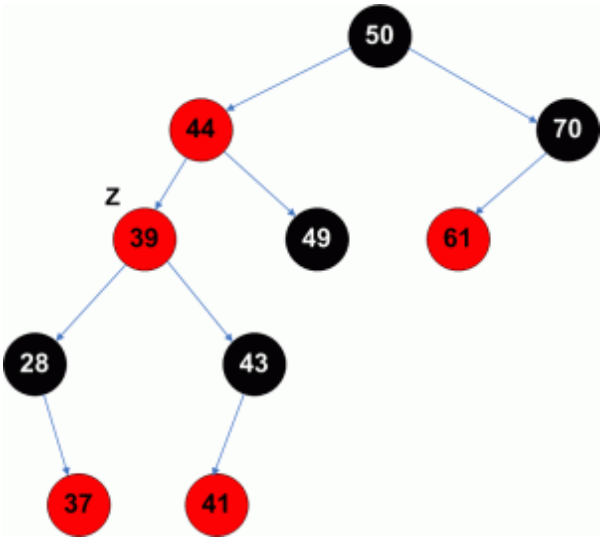
Na koniec węzeł 44 staje węzłem z. Ale ma on czerwonego ojca, więc drzewo wciąż nie jest drzewem RBT. Efekt wymienionych przed chwilą operacji widzimy poniżej:




W tym momencie ojciec węzła **z** jest czerwony (węzeł 39), jest on również lewym synem swego ojca. Z kolei stryj węzła **z** jest czarny (węzeł 70). Następnie należy sprawdzić, jakim synem jest węzeł **z** - okazuje się, że jest prawym synem węzła 39. Wniosek - zachodzi przypadek **2b**. W konsekwencji:

- nowym węzłem **z** staje się ojciec dotychczasowego węzła **z** (czyli 39)
- wykonujemy lewą rotację względem nowego węzła **z**

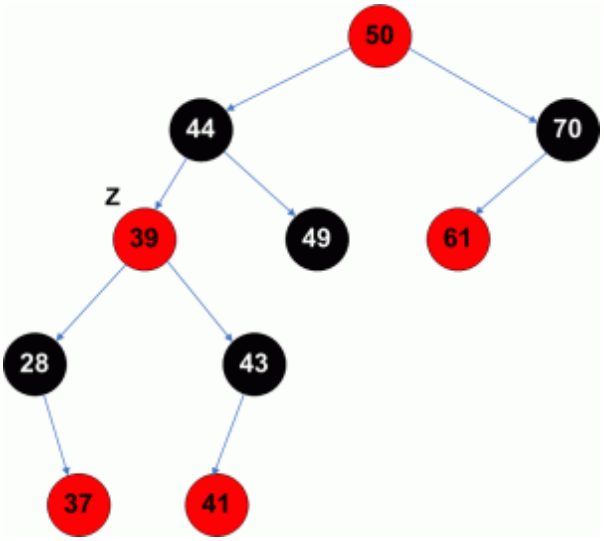
Co z tego wynikło, zobaczymy poniżej:




Wykonujemy ciąg dalszy według przypadku **2c**:

- kolorujemy na czarno ojca węzła **z** (czyli węzeł 44 staje się czarny)
- kolorujemy na czerwono dziadka węzła **z** (czyli węzeł 50 jest malowany na czerwono)

Efekt tych zmian jest widoczny na poniższym schemacie:




Na koniec została (zgodnie z **2c**) do wykonania prawa rotacja względem węzła 50, czyli dziadka węzła **z**. Ponieważ po wykonaniu wspomnianej rotacji węzeł **z** zyskał czarnego ojca, zatem drzewo ma już w tym momencie wszystkie własności drzewa RBT. Na tym kończy się proces wstawiania węzła.

Ostatecznie nasze drzewo posiada strukturę taką, jaka widnieje na rysunku poniżej.