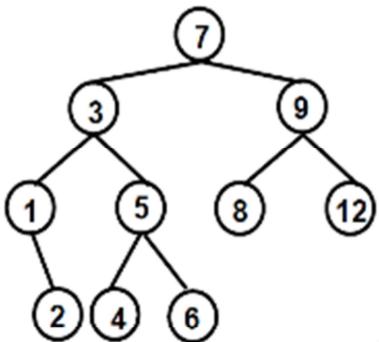


SDIZO-BST

Drzewo poszukiwań binarnych (ang. Binary Search Tree, czyli drzewo BST), w którym dla każdego węzła x , musi być spełniony następujący warunek:

wartość każdego elementu leżącego w lewym poddrzewie węzła x jest nie większa niż wartość węzła x , natomiast wartość każdego elementu leżącego w prawym poddrzewie węzła x jest nie mniejsza niż wartość tego węzła.



```
struct node
{
    int key;      klucz (wartość wierzchołka)
    node *left;   wsk .na lewego potomka
    node *right;  wsk. na prawego potomka
    node *parent; wsk. na rodzica
}
```

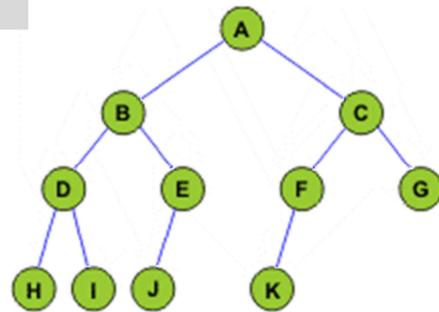
Drzewo BST służy do przechowywania wartości unikalnych czyli kluczy (wraz z dodatkowymi danymi). Dopuszczalne jest występowanie wielu takich samych wartości, ale rzadko się z tego korzysta.

Drzewo BST jest drzewem binarnym ukorzenionym (podobnie jaki kopiec), ale ponieważ nie jest prawie pełne, to nie można zastosować tablicy do jego przechowywania. Należy zatem zastosować klasyczną strukturę dla grafów jak przedstawiono na slajdzie (struktura wierzchołka – *struct node*). Z punktu widzenia programistycznego drzewo jest „widziane” tylko poprzez wskaźnik do korzenia.

SDIZO-BST

A L_A R_A=> A(BL_BR_B)R_A ...=>ABDHIEJCFKG

```
void preorder(node *p)
{
    if(p==NULL)    return;
    P(p);
    preorder(p->left);
    preorder(p->right);
}
```



L_A A R_A=>=>HDIBJEAKFCG

```
void inorder(node *p)
{
    if(p==NULL)    return;
    inorder(p->left);
    P(p);
    inorder(p->right);
}
```

L_A R_A A=> ...=>HIDJEBKFGCA

```
void postorder(node *p)
{
    if(p==NULL)    return;
    postorder(p->left);
    postorder(p->right);
    P(p);
}
```

Jedną z operacji wykonywanych na grafie jest operacja odwiedzenia wszystkich wierzchołków i wykonania operacji w odwiedzonym wierzchołku. Dla drzewa binarnego ukorzenionego są jest pewna specyficzna grupa metod składająca się z 3 algorytmów nazywanych: *preorder*, *inorder* i *postorder*

Ich postać w postaci algorytmów rekurencyjnych przedstawiono na slajdzie. Ogólnie każdy algorytm składa się z 3 elementów: wywołania algorytmu dla prawego i lewego potomka oraz wywołania operacji P dla danego wierzchołka. Nazewnictwo metod zależy od miejsca umieszczenia operacji P wśród wywołań algorytmu dla każdego potomka. I tak dla *preorder* najpierw wykonujemy operację P a dopiero później wywołujemy operację dla prawego i lewego potomka (poddziewa). Prefiks *pre* oznacza *przed* (stąd operacja P jest pierwsza). W *inorder* operacja P jest umieszczona pomiędzy wywołaniami algorytmu dla potomków (słowo *in* po angielsku oznacza *w* – w domyśle pośrodku). Słowo *post* oznacza *po* stąd operacja P jest na końcu.

Założymy, że operacja P wyświetla etykietę odwiedzonego wierzchołka. Operację przechodzenia po drzewie zaczynamy od korzenia. W związku z tym dla metody *preorder* najpierw zostanie wyświetlony wierzchołek w którym jesteśmy i następnie algorytm zostanie wywołany dla lewego i prawego poddrzewa. Dla drzewa na slajdzie zobrazujemy jako ciąg: AL_AR_A, gdzie A oznacza symbol wyświetlonego wierzchołka, L_A wywołanie algorytmu dla lewego poddrzewa wierzchołka A, R_A dla prawego poddrzewa wierzchołka A

Analogicznie jest dla pozostałych metod.

Kontynuując działanie algorytmu *preorder* wywołujemy algorytm dla lewego poddrzewa wierzchołka A (L_A). Lewe poddrzewo wierzchołka A to poddrzewo którego korzeniem jest

B. Zatem można to zapisać:

$AL_A R_A \Rightarrow A (B L_B R_B) R_A \Rightarrow$

$AB L_B R_B R_A \Rightarrow (\text{rozwijamy pierwszy nierożwinięty symbol } L \text{ lub } R \text{ czyli } L_B) \Rightarrow AB (DL_D R_D) R_B R_A \Rightarrow$

$ABDL_D R_D R_B R_A \Rightarrow (\text{rozwijamy LD}) \Rightarrow ABD(HL_H R_H) R_D R_B R_A \Rightarrow$

$ABDHL_H R_H R_D R_B R_A \Rightarrow (\text{rozwijamy } L_H\text{-lewe poddrzewo wierzchołka H jest puste}) \Rightarrow$

$ABDHR_H R_D R_B R_A \Rightarrow (\text{rozwijamy } R_H\text{-prawe poddrzewo wierzchołka H jest puste}) \Rightarrow$

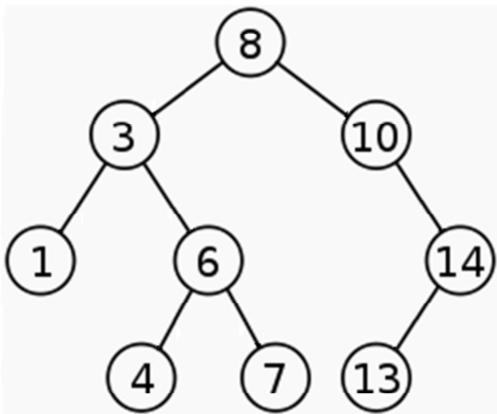
$ABDHR_D R_B R_A \Rightarrow (\text{rozwijamy } R_D) \Rightarrow ABDH(IL_I R_I) R_B R_A \Rightarrow$

$ABDHIL_I R_I R_B R_A \Rightarrow (\text{rozwijamy } L_I\text{-lewe poddrzewo wierzchołka I jest puste}) \Rightarrow$

$ABDHIR_I R_B R_A \Rightarrow (\text{rozwijamy } R_I\text{- prawe poddrzewo wierzchołka I jest puste}) \Rightarrow$

$ABDHIR_B R_A \Rightarrow (\text{rozwijamy RB}) \Rightarrow ABDHI(ELERE)R_A \Rightarrow$

I tak dalej $\Rightarrow ABDHIEJCFKG$



Przryswoać do zeszytu

SDIZO-BST

Wyszukiwanie w drzewie

```
1 BST_TREE_SEARCH (Node, Key):
2   if (Node == NULL) or (Node->Key == Key)
3     return Node
4   if Key < Node->Key
5     return BST_TREE_SEARCH (Node->Left, Key)
6   return BST_TREE_SEARCH (Node->Right, Key)
```

```
1 ITERATIVE_BST_TREE_SEARCH (Node, Key):
2   while ((Node != NULL) and (Node->Key != Key))
3     if (Key < Node->Key)
4       Node = Node->left
5     else
6       Node = Node->right
7   return Node
```

4

Na slajdzie przedstawiano algorytm wyszukiwania wartości *Key* w drzewie w dwóch wersjach – rekurencyjnej i iteracyjnej. W algorytmie, jeśli klucz *Key* nie zostanie znaleziony algorytm zwróci wartość NULL, a jeżeli wartość zostanie znaleziona to zwróci wskaźnik na wierzchołek zawierający wartość *Key*

Wyszukiwanie zawsze zaczyna się od korzenia, czyli parametrem algorytmu musi być wskaźnik na korzeń i poszukiwana wartość.

Jeżeli chodzi o złożoność wyszukiwania w drzewie BST to zależy ono od struktury drzewa. W wersji optymistycznej (gdy zapełnionej są wszystkie poziomy oprócz ostatniego) złożoność wynosi $O(\log n)$, w pesymistycznym $O(n)$

SDIZO-BST

Szukanie wartości minimalnej / maksymalnej w BST

```
BST_SEARCH_MIN_KEY(Node):
    while (Node->left != NULL)
        Node = Node->left
    return Node
```

```
BST_SEARCH_MAX_KEY(Node):
    while (Node->right != NULL)
        Node = Node->right
    return Node
```

5

Patrząc na drzewo BST łatwo zauważyc, że minimalny element drzewa jest elementem najbardziej wysuniętym w lewo, a maksymalny – w prawo w stosunku do korzenia. Zatem zaczynając od korzenia wystarczy iść lewym (prawym) wskaźnikiem potomka aby dojść do skrajnego elementu, który jest elementem minimalnym (maksymalnym). Ostatni element minimalny(maksymalny) ma wskaźnik na lewego(prawego) potomka ustawiony na NULL.

Funkcje zwracają wierzchołek zawierający element minimalny lub maksymalny.

SDIZO-BST wstawianie klucza

```
1 BST_TREE_INSERT_NODE(Root, InsertNode)
2 y = NULL //adres rodzica
3 x = Root
4 while (x != NULL)
5     y = x
6     if (InsertNode->Key < x->Key)           Wstawianie klucza:
7         x = x->Left                         a)szukamy rodzica
8     else                                     b)wstawiamy
9         x = x->Right

10    InsertNode->Parent = y // ustawiamy adres rodzica
11        // w dodawanym wierzchołku
12    if (y == NULL) //drzewo do którego wstawiamy klucz jest puste
13        Root = InsertNode
14    else
15        if (InsertNode->Key < y->Key)
16            y->Left = InsertNode
17        else
18            y->Right = InsertNode
```

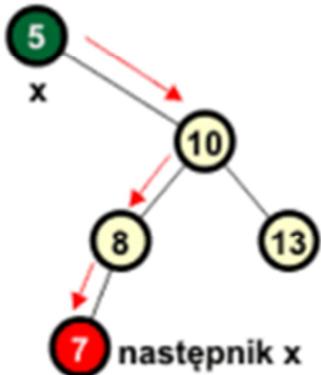
6

Wstawianie nowego elementu (klucza) do drzewa polega na znalezieniu jego rodzica i odpowiednim zmodyfikowaniu pola rodzica (aby jego pole wskaźnika na lewego lub prawego potomka wskazywało na dodawany wierzchołek) oraz odpowiedniego pola dodawanego elementu (wskaźnika na rodzica).

Procedura wyszukiwania rodzica jest podobna do procedury wyszukiwania klucza na poprzednich slajdach. Główna różnica polega na tym, że w przypadku wyszukiwania wartości klucza algorytm szukania zwraca NULL jak klucz nie istnieje (a tak zazwyczaj jest jak dodajemy nowy klucz), natomiast w tym przypadku powinniśmy się zatrzymać krok wcześniej. Dlatego zmienna *y* oznacza analizowany wierzchołek w poprzednim kroku, natomiast zmienna *x* wierzchołek analizowany w bieżącym kroku. Dlatego po zakończeniu pierwszej części w zmiennej *y* znajduje się wierzchołek będący rodzicem dla dodawanego wierzchołka. Druga część algorytmu (linie 10-17) to proces dołączania wierzchołka. W liniach 14-17 wybieramy czy dołączyć do prawego, czy lewego potomka.

Należy zauważyc, że dołączany wierzchołek zawsze będzie liściem.

SDIZO-BST – Wyznaczanie następnika



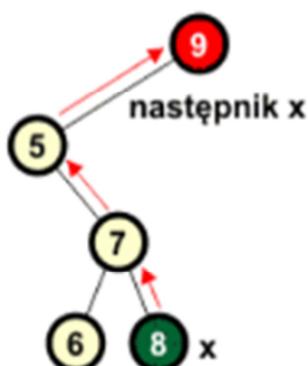
Przypadek 1

Węzeł x posiada prawego syna – następnikiem jest wtedy węzeł o minimalnym kluczu w poddrzewie, którego korzeniem jest prawy syn. Wykorzystujemy tutaj algorytm wyszukiwania węzła o najmniejszym kluczu w prawym poddrzewie.

BST_SEARCH_MIN_KEY(x->right)

7

Następnik klucza x to taki klucz w drzewie, którego wartość jest najbliższa x ale $y > x$. Analogicznie definiujemy poprzednika. Wyznaczanie następnika (poprzednika) jest istotne w operacji usuwania klucza. Wyszukiwanie następnika można przedstawiono w różnych topografiach drzewa BST. Na obecnym slajdzie omówiono przypadek, gdy węzeł zawierający klucz x ma prawego potomka.

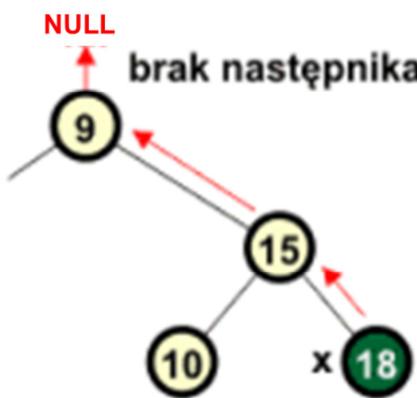
**Przypadek 2a**

Węzeł x nie posiada prawego syna. W takim przypadku, idąc w górę drzewa, musimy znaleźć pierwszego ojca, dla którego nasz węzeł leży w lewym poddrzewie. Tutaj również nie musimy porównywać węzłów. Po prostu idziemy w górę drzewa i w węźle nadrzędnym sprawdzamy, czy przyszliśmy od strony lewego syna. Jeśli tak, to węzeł ten jest następnikiem. Jeśli nie, to kontynuujemy marsz w górę drzewa. Wymaga to zapamiętywania adresów kolejno mijanych węzłów.

8

Kolejny przypadek to taki, gdy węzeł x nie ma prawego syna. Inna wersja algorytmu przedstawionego na slajdzie dla tego przypadku polega na tym, że poruszamy się w górę (po wskaźniku rodzica) do momentu natrafienia na pierwszy węzeł, którego wartość jest większa od x. Ten węzeł będzie następnikiem x

SDIZO-BST Wyznaczanie następnika



Przypadek 2b

Węzeł x nie posiada prawego syna. Idąc w górę drzewa, dochodzimy do korzenia, a następnie do adresu NULL, na które wskazuje pole *parent* korzenia drzewa BST. W takim przypadku węzeł x jest węzłem o największym kluczu i nie posiada następnika.

9

Ostatni przypadek opisuje sytuację, gdy węzeł x nie ma poprzenika.

SDIZO-BST Wyznaczanie następnika i poprzednika

```
1 BST_FIND_SUCCESSOR(Node)      <-wyszukiwanie nastepnika
2 if (Node->right != NULL)
3     return BST_SEARCH_MIN_KEY(Node->right)
4 Node_tmp = Node->parent
5 while (Node_tmp != NULL and Node_tmp->left != Node)
6     Node = Node_tmp
7     Node_tmp = Node_tmp->parent
8 return Node_tmp
```

```
1 BST_FIND_PREDECESSOR(Node)    <-wyszukiwanie poprzednika
2 if (Node->left != NULL)
3     return BST_SEARCH_MAX_KEY(Node->left)
4 Node_tmp = Node->parent
5 while (Node_tmp != NULL and Node_tmp->right != Node)
6     Node = Node_tmp
7     Node_tmp = Node_tmp->parent
8 return Node_tmp
```

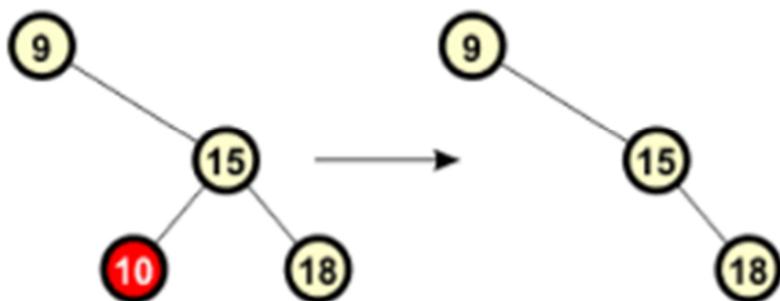
10

Na slajdzie przedstawiono dwa algorytmy - jeden wyszukuje następnika, a drugi poprzednika. Oba działają wg opisu na poprzednich slajdach. Omówmy pierwszy algorytm (drugi jest analogiczny). Algorytm zwraca węzeł następnika. Parametrem algorytmu jest wierzchołek (węzeł) dla którego należy znaleźć następnika. W liniach 2,3 rozpatrzono jest przypadek, gdy węzeł zawiera prawe poddrzewo. Pozostałe linie przedstawiają przypadek poruszania się ku górze. Jest tu zmienna Node_tmp, która reprezentuje rodzica Node. Ta para przemieszcza się ku górze dopóki jest spełniony warunek w linii 5. Jeżeli Node_tmp jest równy NULL to oznacza, brak następnika, natomiast, gdy spełniony jest warunek Node_tmp->left==Node, to oznacza, że Node jest lewym potomkiem Node_tmp i wg tego co jest dwa slajdy wcześniej Node_tmp jest następnikiem.

SDIZO-BST Usuwanie węzła

Przypadek 1

Usuwany węzeł (10) jest liściem, tzn. nie posiada synów. W takim przypadku po prostu odłączamy go od drzewa i usuwamy.



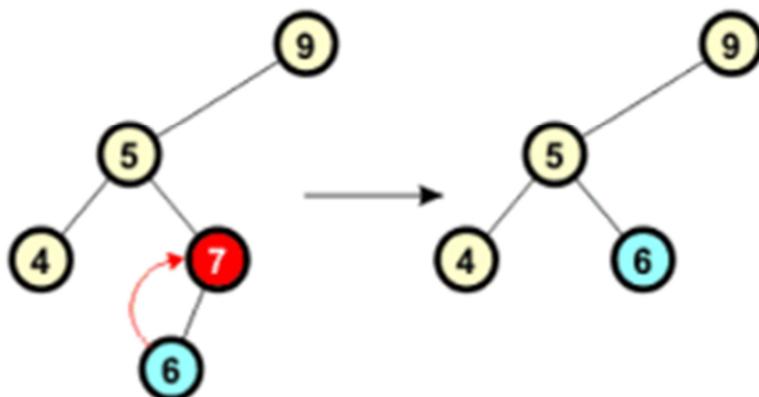
11

W usuwaniu węzła (podobnie jak w szukaniu następnika/ poprzednika) również można wyróżnić parę przypadków. Na bieżącym slajdzie przedstawiono najprostszą sytuację, gdy usuwany węzeł jest liściem. Wówczas wystarczy w tym przypadku zwolnić pamięć zajmowaną przez usuwany węzeł natomiast i zmodyfikować rodzica tzn. ustawić na NULL wskaźnik na potomka (prawy lub lewy), który wcześniej wskazywał na usuwany węzeł.

SDIZO-BST Usuwanie węzła

Przypadek 2

Usuwany węzeł (7) posiada tylko jednego potomka. Węzeł zastępujemy jego potomkiem (razem z ewentualnym poddrzewem potomka), po czym węzeł usuwamy z pamięci.

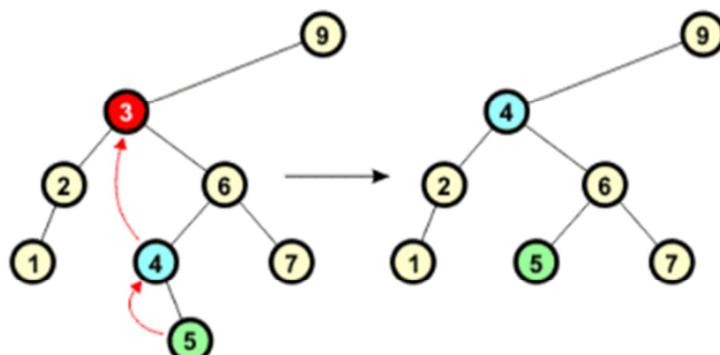


12

Przypadek drugi, gdy usuwany węzeł ma tylko jednego potomka. Potomkiem może być zarówno pojedynczy wierzchołek jak i całe poddrzewo.

SDIZO-BST Usuwanie węzła

Przypadek 3 Usuwany węzeł (3) posiada dwóch synów. Znajdujemy węzeł będący następcą usowanego węzła. Przenosimy dane i klucz z następcy do usowanego węzła, po czym następcę usuwamy z drzewa – do tej operacji można rekurencyjnie wykorzystać tę samą procedurę lub zastąpić następcę przez jego prawego syna (następnik nigdy nie posiada lewego syna). **Jako wariant można również zastępować usuwany węzeł jego poprzednikiem.**



13

Ostatni przypadek jest wtedy gdy nie występują dwa poprzednie przypadki. Usuwany węzeł zastępujemy następcą (lub poprzednikiem) i usuwamy następcę (poprzednika). Usuwanie następcy (poprzednika) będzie się zawsze odbywało wg przypadku 1 lub 2, gdyż następnik będzie albo liściem, albo będzie miał co najwyżej jednego potomka (potomek może reprezentować albo jeden wierzchołek, albo całe poddrzewo)

Z punktu widzenia implementacji sam proces zastępowania usuwanego elementu następcą polega na skopiowaniu klucza następcy z ewentualnymi dodatkowymi danymi do usuwanego węzła nie zmieniając wskaźników na potomków i rodzica tego węzła.

SDIZO-BST Usuwanie węzła

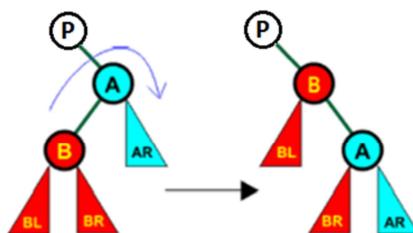
```
BST_TREE_DELETE (Root, DeleteNode):
    if (DeleteNode->Left==NULL) or (DeleteNode->Right==NULL)
        y=DeleteNode
    else
        y=BST_FIND_SUCCESSOR(DeleteNode)
        if (y->Left != NULL)  x=y->Left //następnik może mieć tylko
        else      x=y->Right           // co najwyżej jednego syna albo wcale
        if (x!=NULL)
            x->parent = y->parent
        if (y->parent == NULL) //y jest korzeniem
            Root = x
        else
            if (y == y->parent->Left)
                y->parent->Left = x
            else
                y->parent->Right = x
        if (y != DeleteNode)
            DeleteNode->Key = y->Key
            // Jeśli mamy inne pola, to je także należy skopiować
        return y
```

14

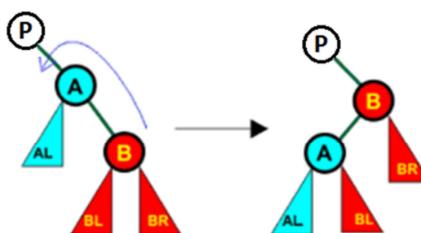
Powyżej przedstawiono algorytm usuwania węzła uwzględniający wcześniej opisane przypadki

SDIZO-BST Rotacje

Rotacja w prawo



Rotacja w lewo



15

Rotacje służą do zmiany struktury drzewa, natomiast nie zmieniają zawartości kluczy. Na slajdzie przedstawiono dwa typy rotacji: w prawo i w lewo. W każdej rotacji można wyróżnić węzeł względem którego rotacja wystąpi (będzie to węzeł A)

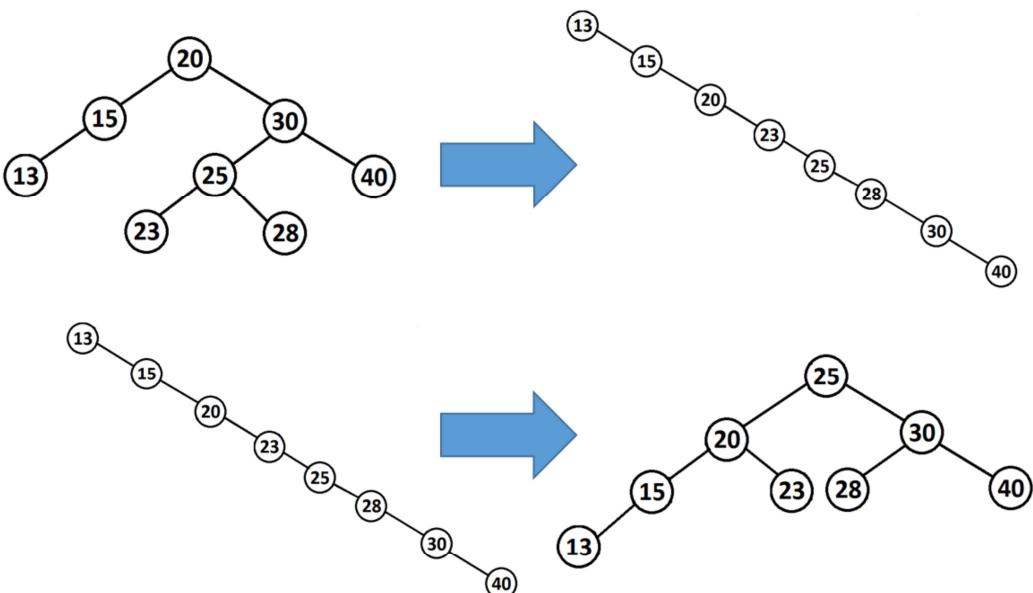
Skupmy się na rotacji w prawo. AR reprezentuje prawe poddrzewo węzła A, lewym potomkiem węzła A jest węzeł B wraz ze swoim lewym (BL) i prawym poddrzewem (BR). W wyniku rotacji węzeł B zajmuje miejsce węzła A, węzeł A zostaje „pociągnięty” w dół wraz z AR. Ponieważ w wyniku rotacji prawy potomek węzła B wskazuje na A to należy gdzieś przepiąć jego prawe poddrzewo BR. Przed rotacją poddrzewo BR reprezentuje klucze większe od B a mniejsze od A (zgodnie z definicją drzewa BST). Po rotacji wartości w lewym poddrzewie wierzchołka A również reprezentują ten sam zakres liczb, więc można wstawić tam BR

Należy zauważyć, że rotacja w prawo jest możliwa wtedy, gdy istnieje lewy potomek węzła A czyli B. Wartości BR, BL, AR mogą być NULL.

Rotacja w lewo działa analogicznie.

SDIZO-BST Algorytm DSW (Day-Stout-Warren)

ETAP 1 – PROSTOWANIE ETAP 2 - RÓWNOWAŻENIE



16

Jak wspomiano złożoność wyszukiwania w BST w przypadku pesymistycznym wynosi $O(n)$, a optymistycznym $O(\log n)$. Aby przypadek optymistyczny miał miejsce drzewo powinno być zrównoważone. Poniżej przedstawiono definicje drzewa zrównoważonego i doskonale zrównoważonego.

Drzewo binarne jest wyważone (zrównoważone), gdy wysokość lewego i prawego poddrzewa każdego jego wierzchołka nie różni się o więcej niż jeden.

Drzewo jest doskonale zrównoważone, gdy dodatkowo wszystkie jego liście znajdują się na maksymalnie dwóch poziomach.

Drzewo BST może nie być zrównoważone (przykład podano w zadaniu), gdzie w najgorszym przypadku drzewo może przybrać formę liniową.

Aby drzewo było zrównoważone należy je zrównoważyć wykonując algorytm równoważenia drzewa.

Przedstawiony algorytm równoważenia drzewa DSW (nazwa pochodzi od pierwszych liter nazwisk jego twórców) jest algorymem dwuteapowym

W pierwszym etapie należy doprowadzić drzewo do struktury liniowej wykonując szereg rotacji w prawo.

W drugim etapie drzewo linowe jest przekształcane w zrównoważone za pomocą rotacji w lewo.

W wyniku działania algorytmu DSW drzewo jest doskonale zrównoważone.

ETAP 1 – PROSTOWANIE

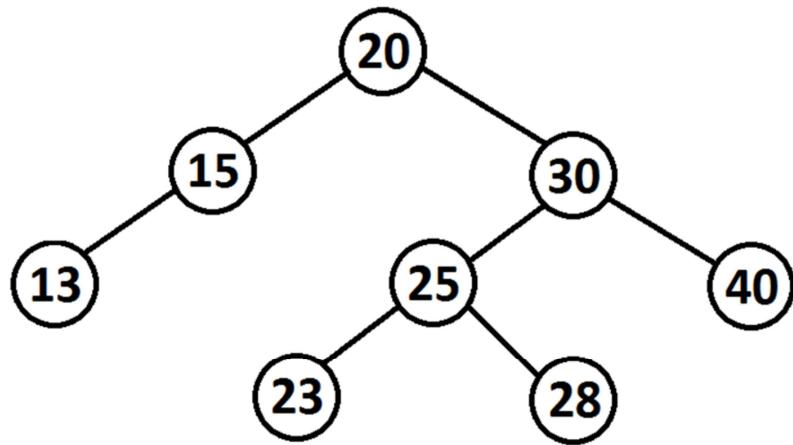
CreateLinearTree (Root)

```
tmp = Root; //tmp to zmienna tymczasowa
while tmp != NULL
    if tmp->Left != NULL
        wykonaj rotację w prawo względem tmp
        tmp = tmp->parent (po rotacji tmp przesunie się w
                            dół a ma zostać w tym samym miejscu)
    else
        tmp = tmp->Right
```

17

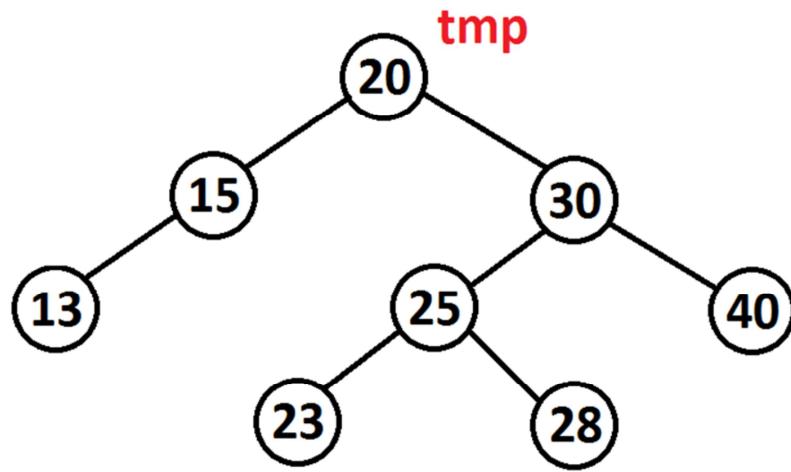
Na slajdzie przedstawiono pierwszy etap algorytmu DSW - prostowanie drzewa

SDIZO-BST Algorytm DSW



18

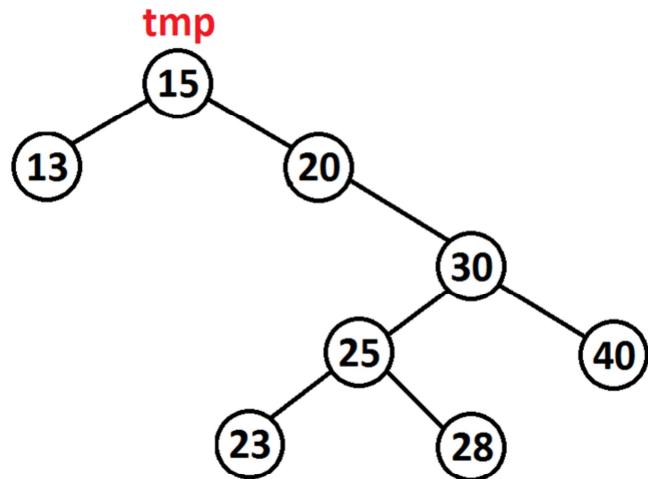
Powyżej przedstawiono drzewo BST, które na przykładzie którego zostanie pokazane działanie algorytmu DSW



19

Zaczynamy od prostowania drzewa. Na początku (zgodnie z przedstawionym algorytmem prostowania) przyporządkujemy zmienną *tmp* do korzenia.

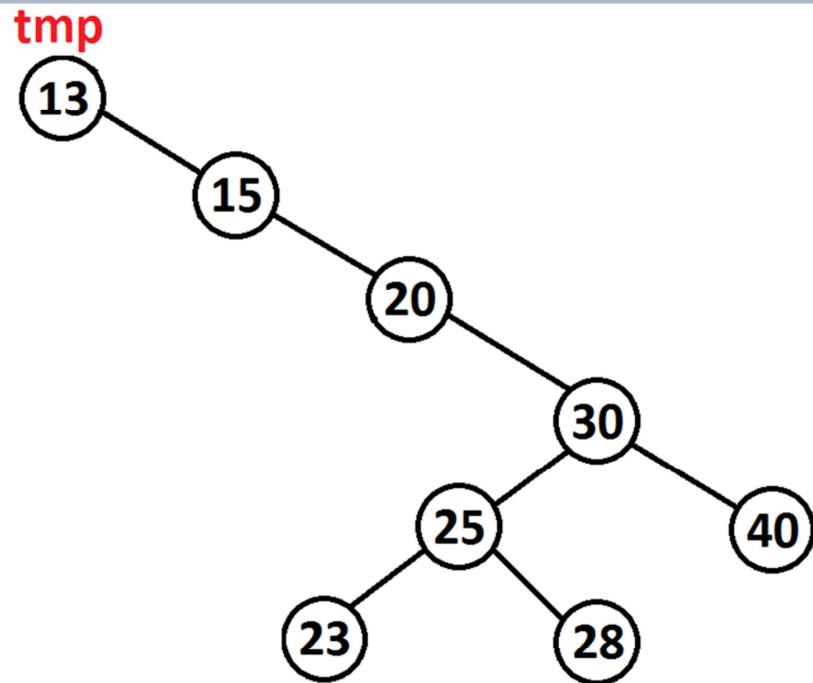
Zgodnie z algorytmem jeżeli wierzchołek *tmp* posiada lewego potomka dokonujemy rotacji względem *tmp* (w tym przypadku posiada)



20

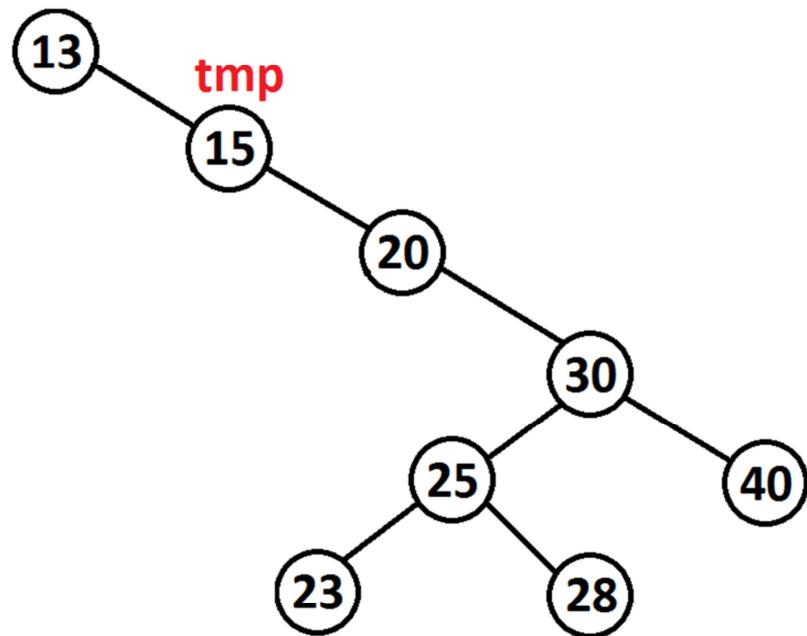
Po dokonanej rotacji względem 20 tmp zostanie przesunięty na 15. Sprawdzamy, czy tmp posiada lewego potomka (posiada) i dokonujemy rotacji w prawo względem 15

SDIZO-BST Algorytm DSW



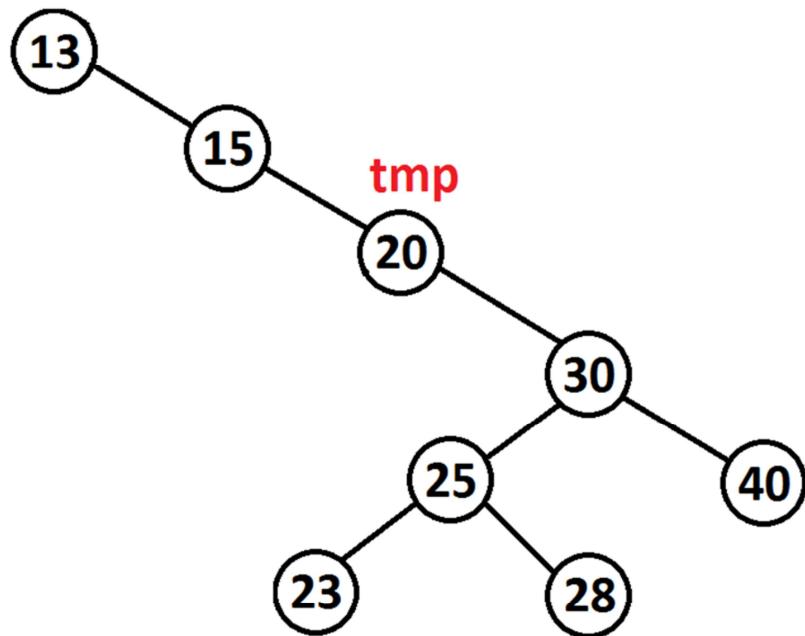
Po dokonanej rotacji względem 15, *tmp* zostanie przesunięty na 13. Ponieważ *tmp* nie ma lewego potomka przesuwamy *tmp* w dół, aż trafimy na wierzchołek, który posiada lewego potomka lub *tmp* przyjmie wartość NULL

SDIZO-BST Algorytm DSW



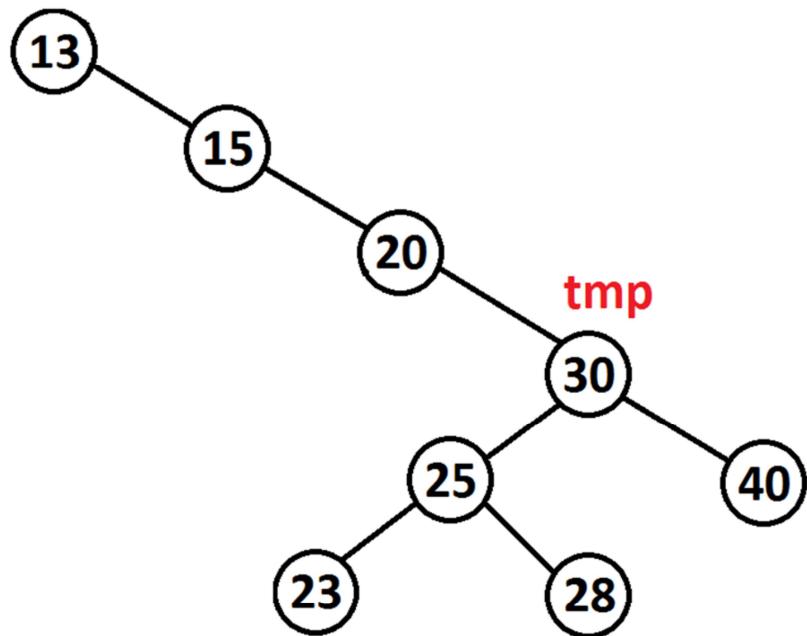
15 nie posiada lewego potomka, więc idziemy dalej w dół (na 20)

SDIZO-BST Algorytm DSW



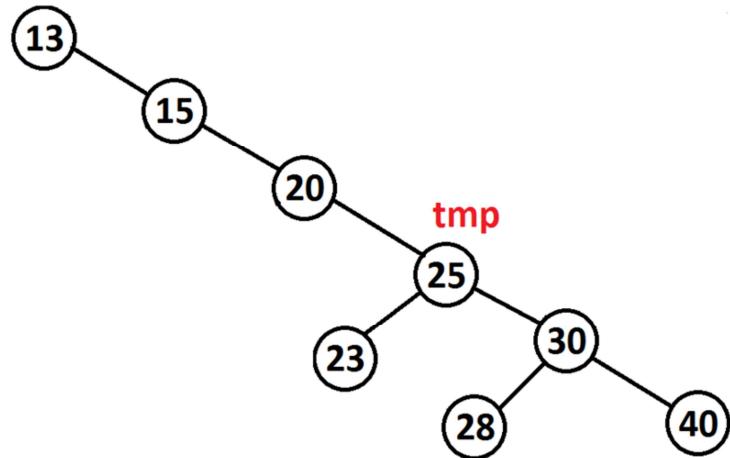
20 nie posiada lewego potomka, więc idziemy dalej w dół (na 30)

SDIZO-BST Algorytm DSW



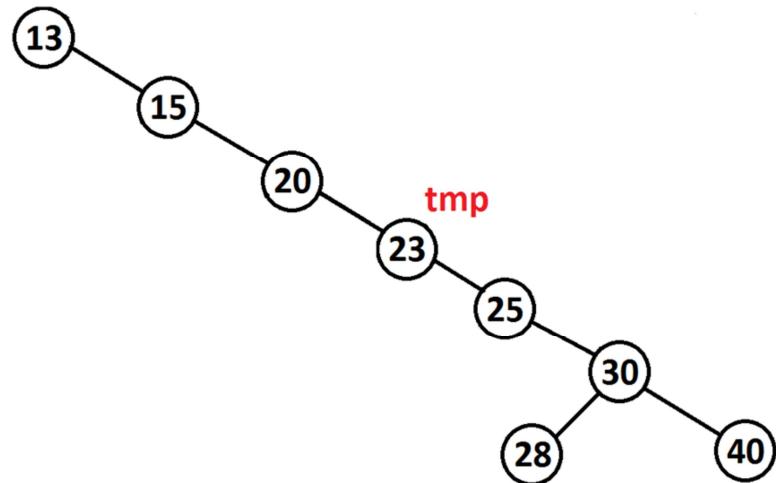
30 posiada lewego potomka wobec tego robimy rotację w prawo względem 30

SDIZO-BST Algorytm DSW

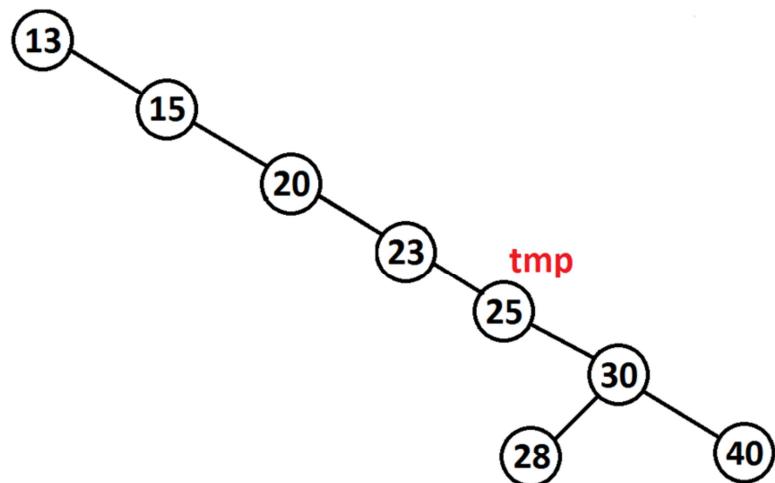


W wyniku rotacji w miejsce 30 wyląduje 25. Węzeł 25 ma lewego potomka wobec tego wykonujemy rotację w prawo względem *tmp* (czyli 25)

SDIZO-BST Algorytm DSW

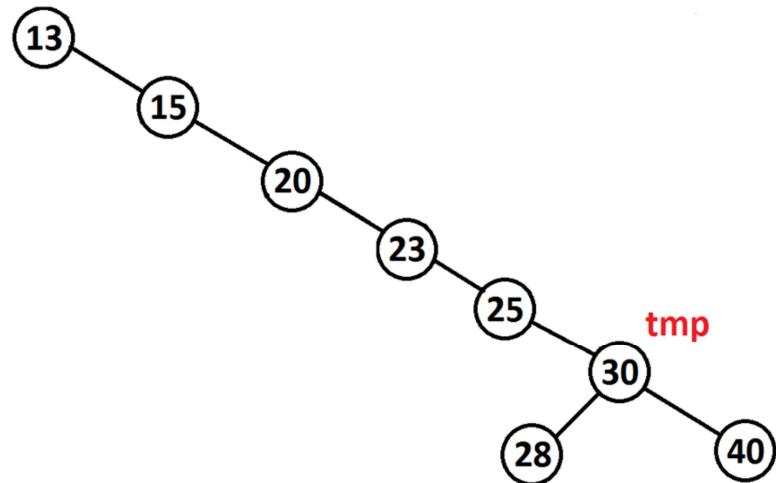


Po wykonaniu rotacji względem 25 tmp jest ustawiony na 23. Ponieważ węzeł 23 nie posiada lewego potomka, wobec tego przesuwamy go w dół (na 25)



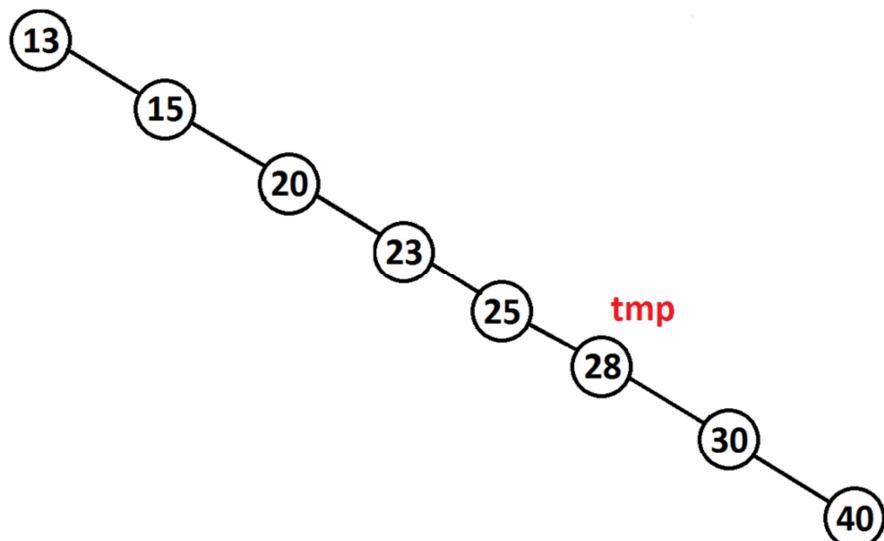
27

Ponieważ węzeł 25 nie posiada lewego potomka, wobec tego przesuwamy go w dół (na 30)



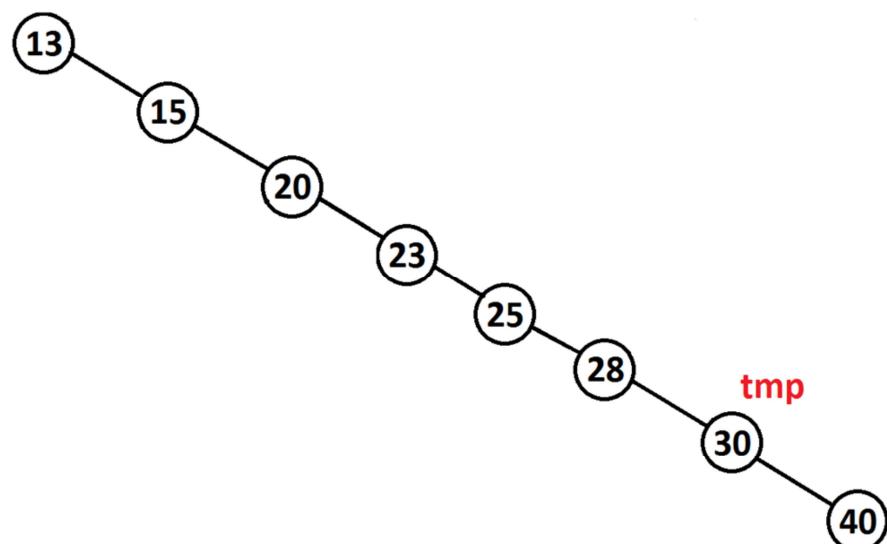
28

Ponieważ 30 posiada lewego potomka wobec tego robimy rotację w prawo względem 30



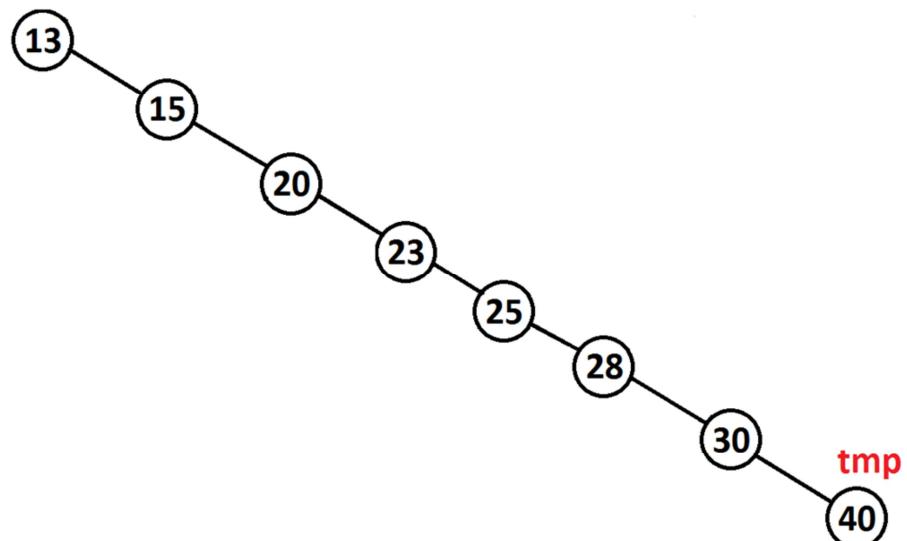
29

Po wykonaniu rotacji względem 30 tmp jest ustawiony na 28. Ponieważ węzeł 28 nie ma lewego potomka, wobec tego przesuwamy go w dół (na 30)



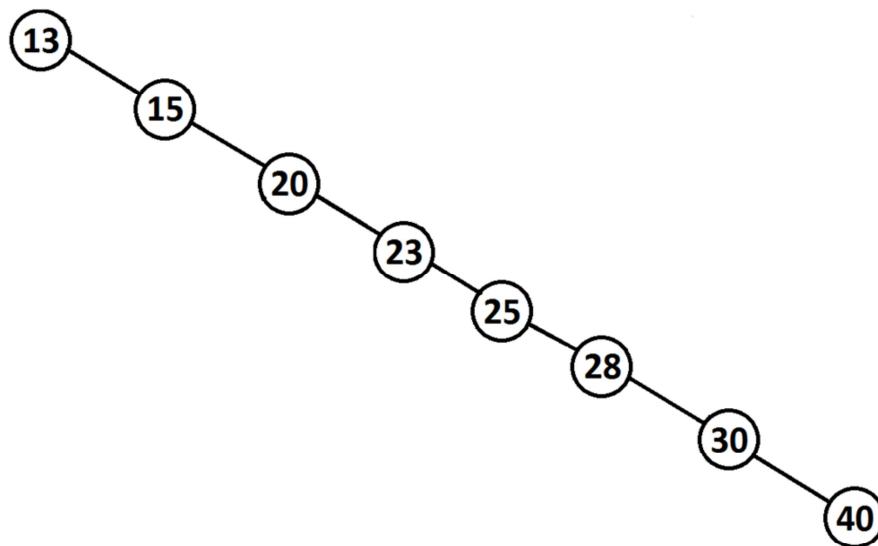
30

Ponieważ węzeł 30 nie posiada lewego potomka, wobec tego przesuwamy go w dół (na 40)



31

Ponieważ węzeł 40 nie posiada lewego potomka, wobec tego przesuwamy go w dół (na NULL)



32

Zmienna *tmp* osiągnęła wartość NULL co oznacza koniec pierwszego etapu - drzewo jest wyprostowane

Łatwo zauważyc, że przedstawiony wcześniej algorytm prostowania nie ustawia wartości nowego korzenia (należałoby go zmodyfikować)

ETAP 2 – RÓWNOWAŻENIE

CreateBalancedTree (Root, n) //n-liczba węzłów

$$m = 2^{\lfloor \log_2(n+1) \rfloor} - 1$$

wykonaj **n-m** rotacji w lewo , startując od początkowego wierzchołka co drugi wierzchołek

while m > 1

$$m = \lfloor m/2 \rfloor$$

wykonaj m rotacji w lewo , startując od początkowego wierzchołka co drugi wierzchołek

m – ilość wierzchołków w drzewie w części zapełnionej

33

Powyżej przedstawiono drugi etap – równoważenie.

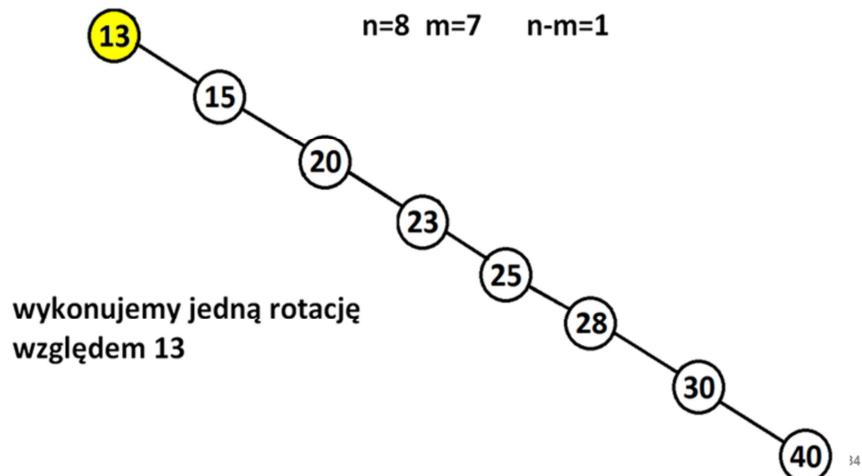
Na początku obliczana jest tajemnicza liczba m. Mówiąc o ilości wierzchołków, które znajdują się w poziomach całkowicie zapełnionych (jak w kopcu). Zgodnie z tym na zerowym poziomie jest jeden wierzchołek, a na kolejnych dwa razy więcej niż poprzednim.

Na początku wykonujemy n-m rotacji co drugi wierzchołek. Na następnych slajdach przedstawiono działanie drugiej części algorytmu.

SDIZO-BST Algorytm DSW

$$m = 2^{\lfloor \log_2(n+1) \rfloor} - 1$$

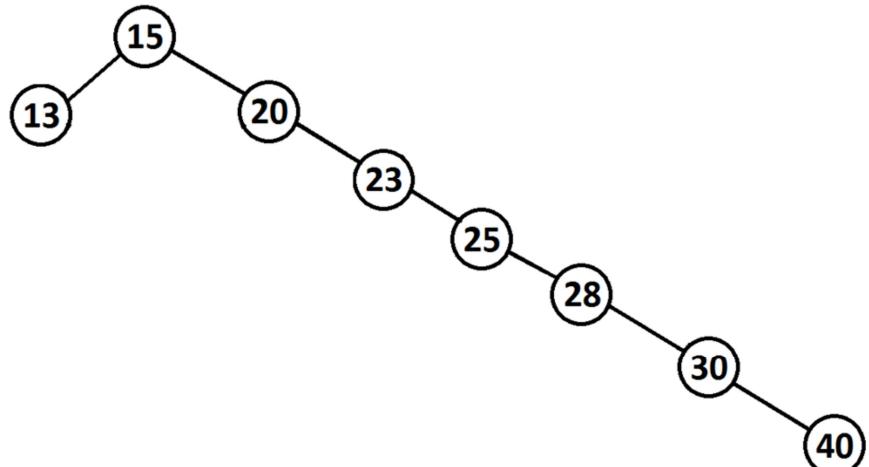
wykonaj $n-m$ rotacji w lewo , startując od początkowego wierzchołka co drugi wierzchołek



Mając wyprostowane drzewo BST dokonujemy $n-m$ kolejnych rotacji startując od początkowego wierzchołka co drugi wierzchołek. Ponieważ $n-m=1$ to dokonujemy pojedynczej rotacji względem pierwszego wierzchołka czyli 13. Jeżeli $n-m > 1$ to kolejnymi wierzchołkami do rotacji byłby wierzchołek 20, kolejny 25, 30, itd..

SDIZO-BST Algorytm DSW

po wykonaniu n-m rotacji



Po wykonywaniu pierwszej rotacji (czyli n-m rotacji)

CreateBalancedTree (Root, n) //n-liczba węzłów

$$m = 2^{\lceil \log_2(n+1) \rceil} - 1$$

wykonaj **n-m** rotacji w lewo , startując od początkowego wierzchołka co drugi wierzchołek

while $m > 1$

$$m = \lfloor m/2 \rfloor$$

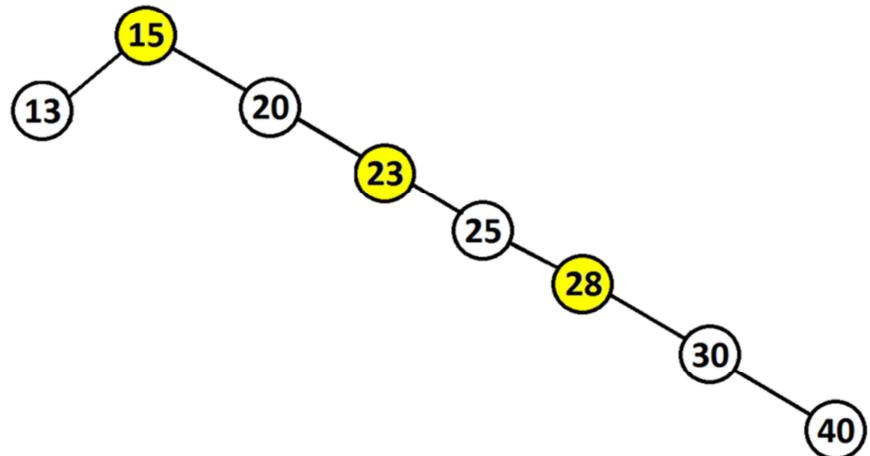
wykonaj **m** rotacji w lewo , startując od początkowego wierzchołka co drugi wierzchołek

36

Teraz druga część.

SDIZO-BST Algorytm DSW

$m = \lfloor m/2 \rfloor = \lfloor 7/2 \rfloor = 3 \rightarrow \text{wykonujemy 3 rotacje}$

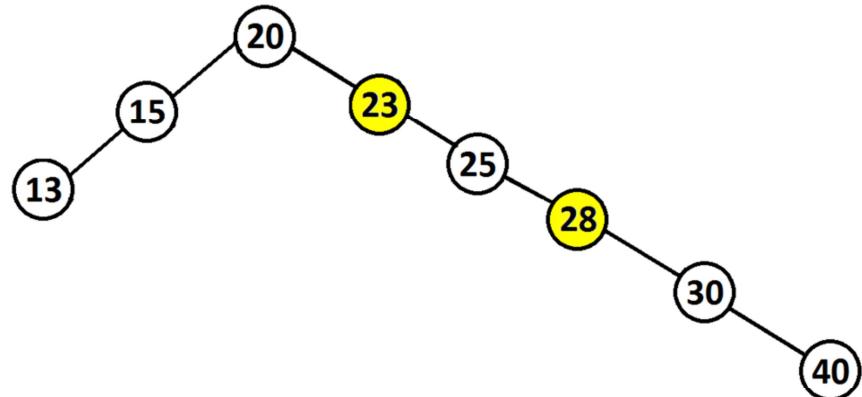


37

Zaznaczamy (na żółto) co drugi wierzchołek względem który będziemy wykonywać rotację w lewo (począwszy od korzenia)

SDIZO-BST Algorytm DSW

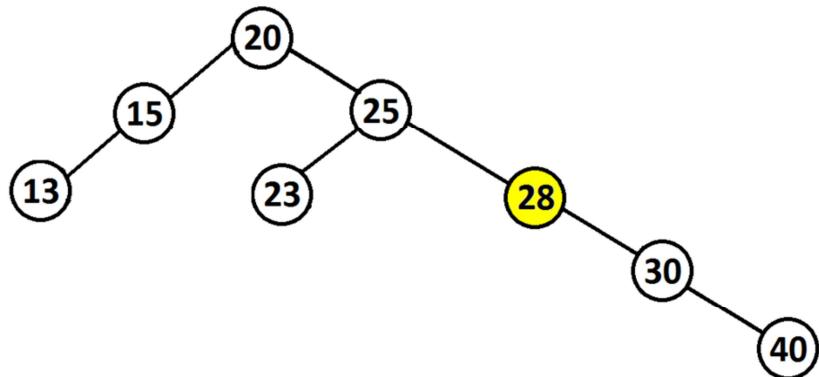
wykonujemy kolejną rotację względem 23



38

Po wykonaniu pierwszej rotacji względem 15.

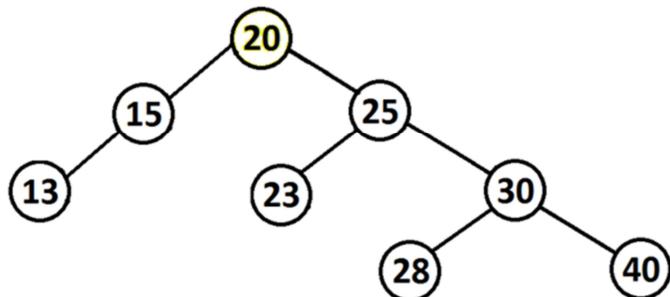
wykonujemy 3 rotacje (względem 28)



39

Po wykonaniu drugiej rotacji względem 23

SDIZO-BST Algorytm DSW



40

Po wykonaniu trzeciej rotacji względem 28

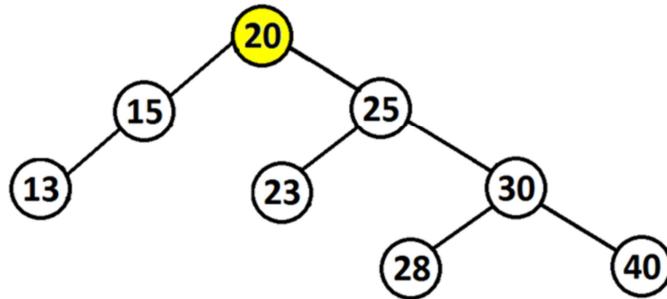
SDIZO-BST Algorytm DSW

```
while m > 1  
    m = ⌊ m/2 ⌋
```

wykonaj m rotacji w lewo , startując
od początkowego wierzchołka co drugi wierzchołek

SDIZO-BST Algorytm DSW

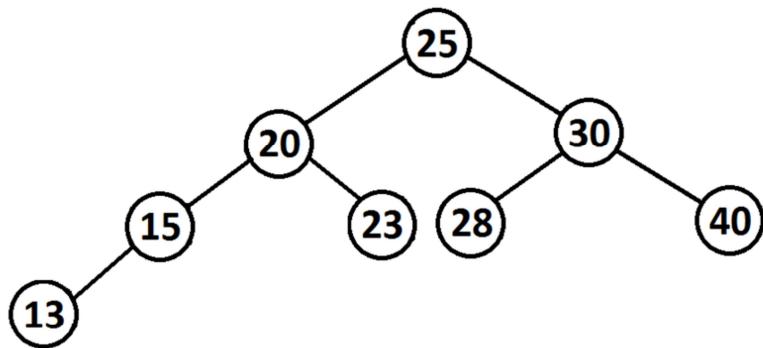
$m = \lfloor m/2 \rfloor = \lfloor 3/2 \rfloor = 1 \Rightarrow \text{wykonujemy 1 rotację}$



42

Po wykonaniu pierwszej pętli obliczamy m dla drugiej pętli (wynosi 1) – zatem wykonujemy rotację względem 20

po wykonaniu wszystkich rotacji - drzewo zrównoważone



43

Po wykonaniu drugiego obiegu pętli $m=0$, więc kończymy algorytm. Jak policzymy ilość wierzchołków na zapełnionych poziomach to wyjdzie $4+2+1 = 7$, czyli tyle, ile wynosi wyliczona liczba m na początku.

Zadanie.

Oszacować złożoność algorytmu DSW.