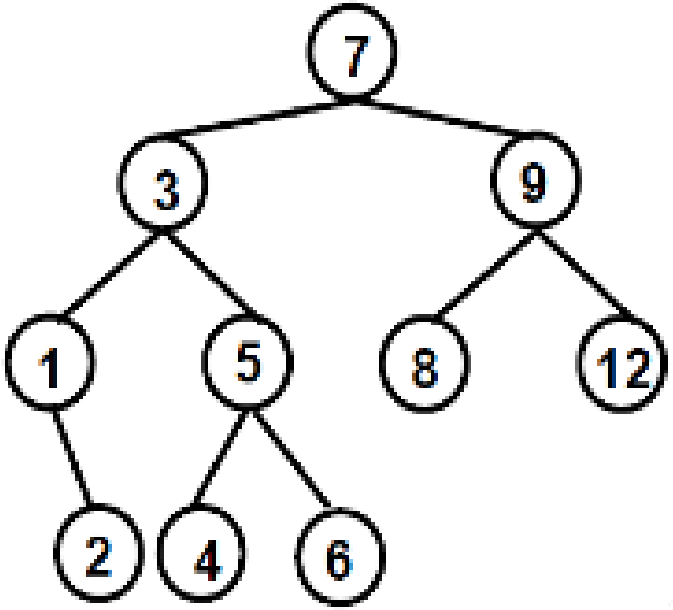
SDIZO-BST

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

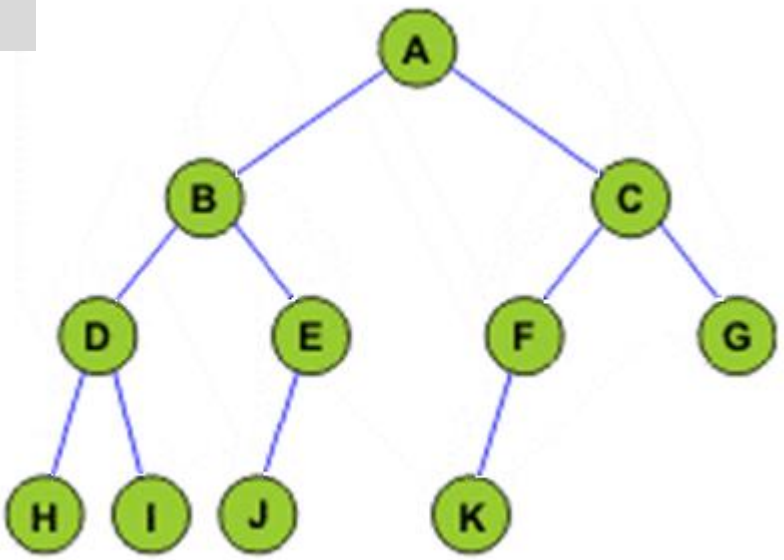



```
struct node
{
    int  key;          klucz (wartość wierzchołka)
    node  *left;     wsk .na lewego potomka
    node  *right;  wsk. na prawego potomka
    node *parent;    wsk. na rodzica
}
```

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);
}
```

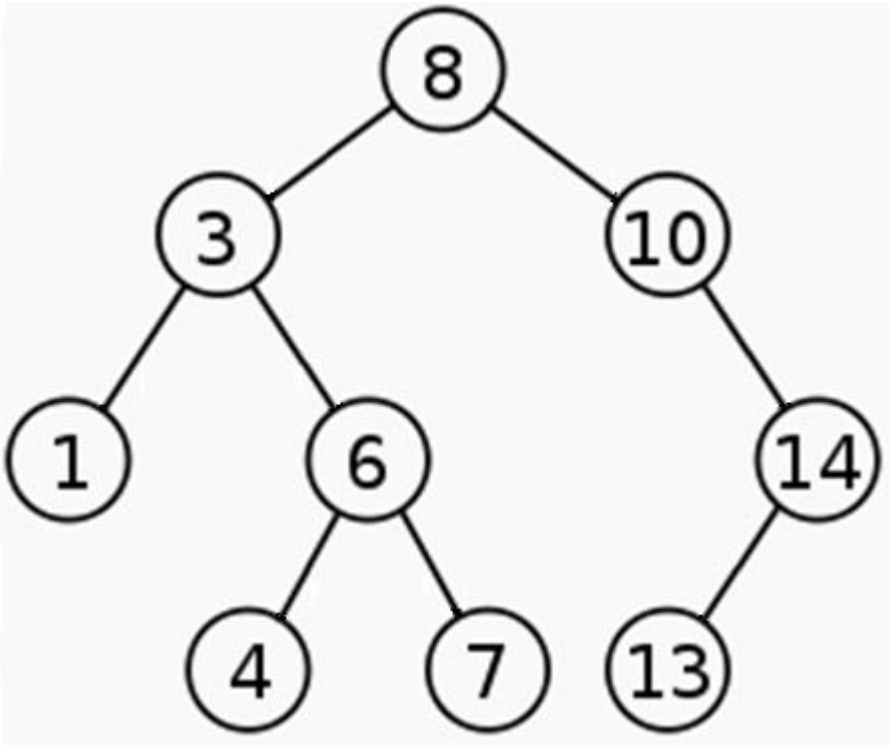


L_A A R_A=> ….=>HDIBJEAKFCG

```
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);
}
```



Przerysować do zeszytu

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
```

Szukanie Min/Max

```
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
```

SDIZO-BST wstawianie klucza

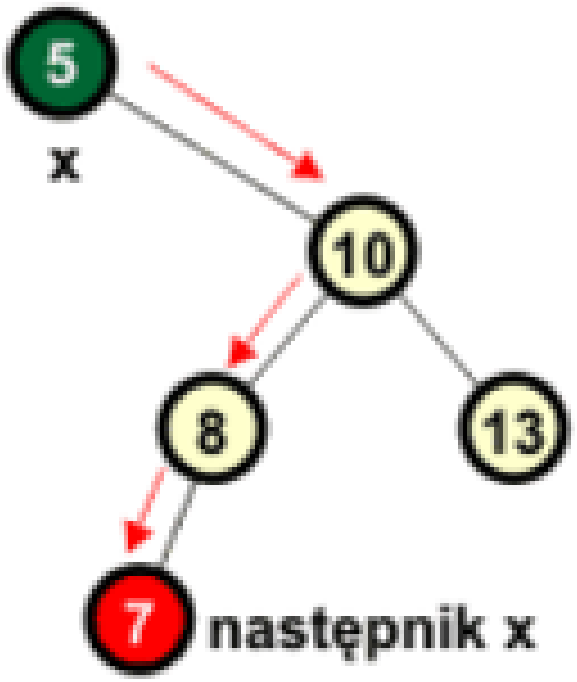
```
1 BST_TREE_INSERT_NODE(Root, InsertNode)
2   y = NULL   //adres rodzica
3    x = Root
4    while (x != NULL)
5         y = x
6         if (InsertNode->Key  <  x->Key)
7              x = x->Left
8         else
9              x = x->Right

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

Wstawianie klucza:

a)szukamy rodzica

b)wstawiamy

SDIZO-BST – Wyznaczanie następnika

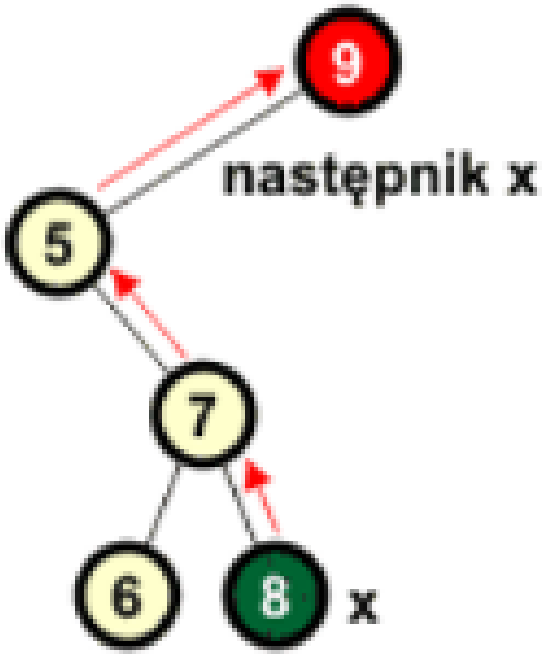


Przypadek 1

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)

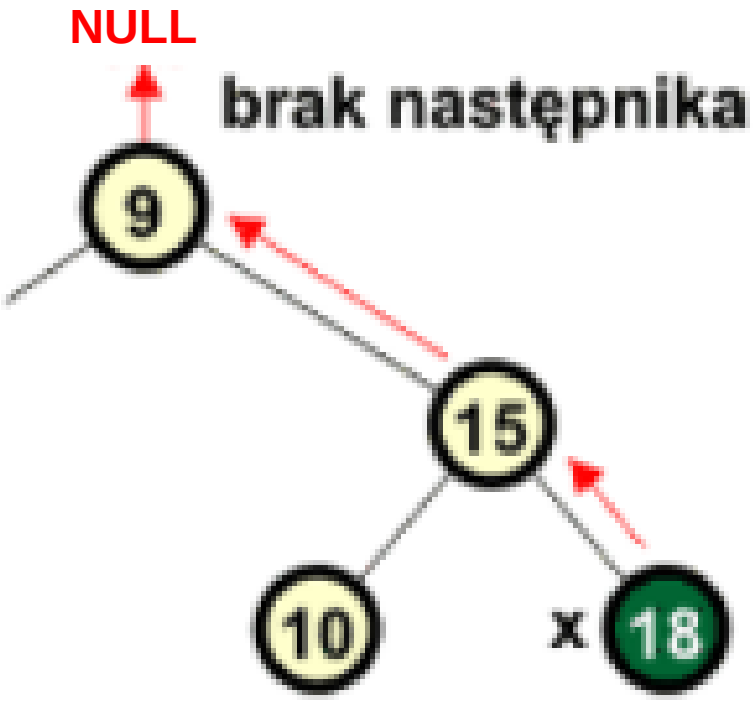
Wyznaczanie następnika



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.

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.

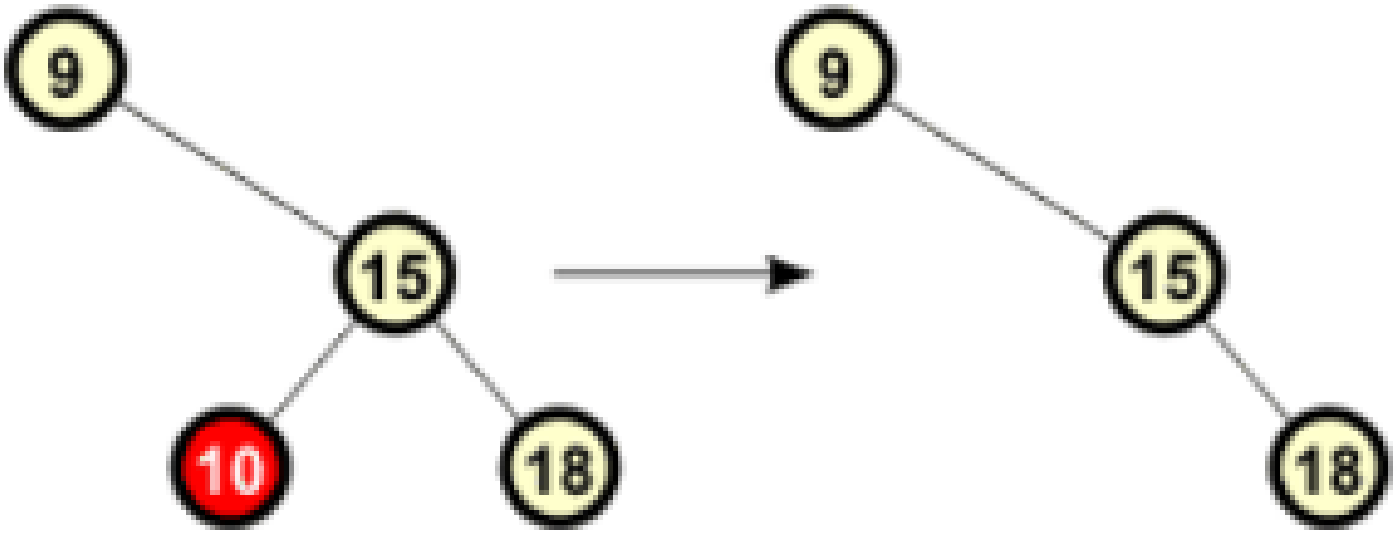
SDIZO-BST Wyznaczanie następnika i poprzednika

```
1 BST_FIND_SUCCESSOR(Node)          <-wyszukiwanie następnika
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
```

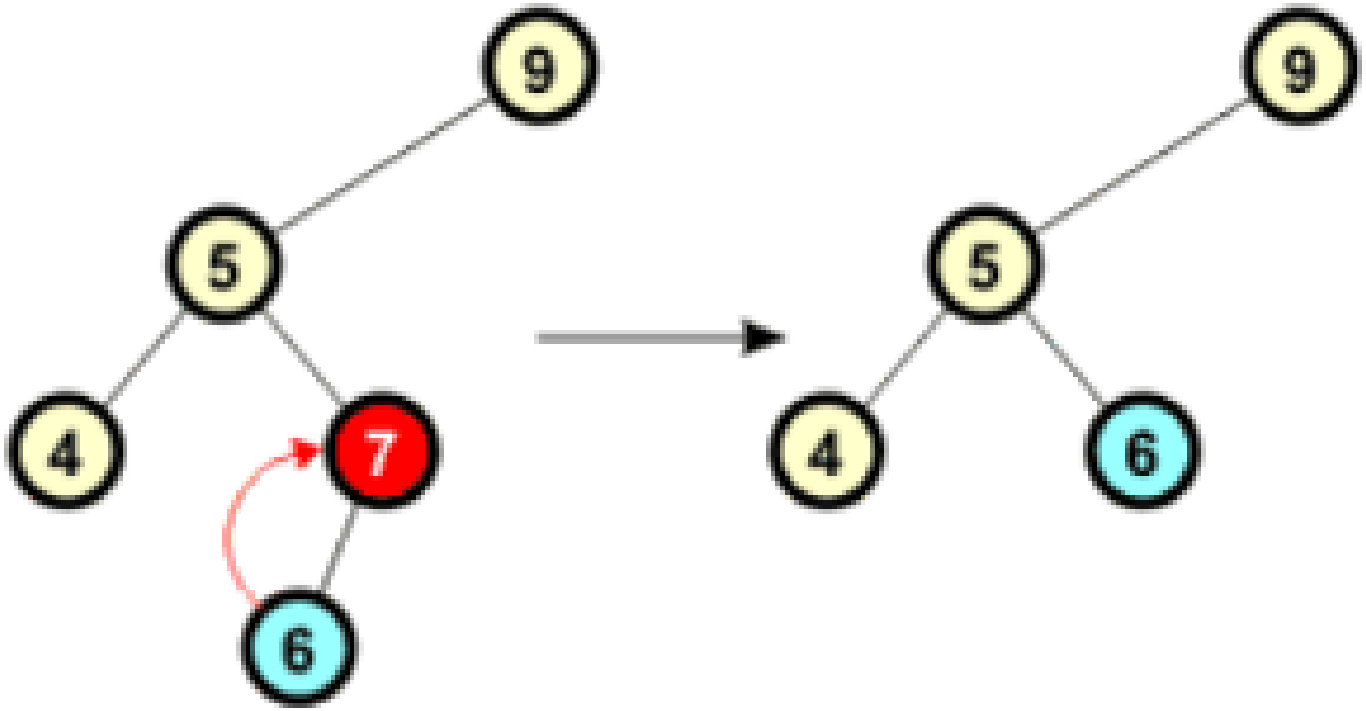
Przypadek 1

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



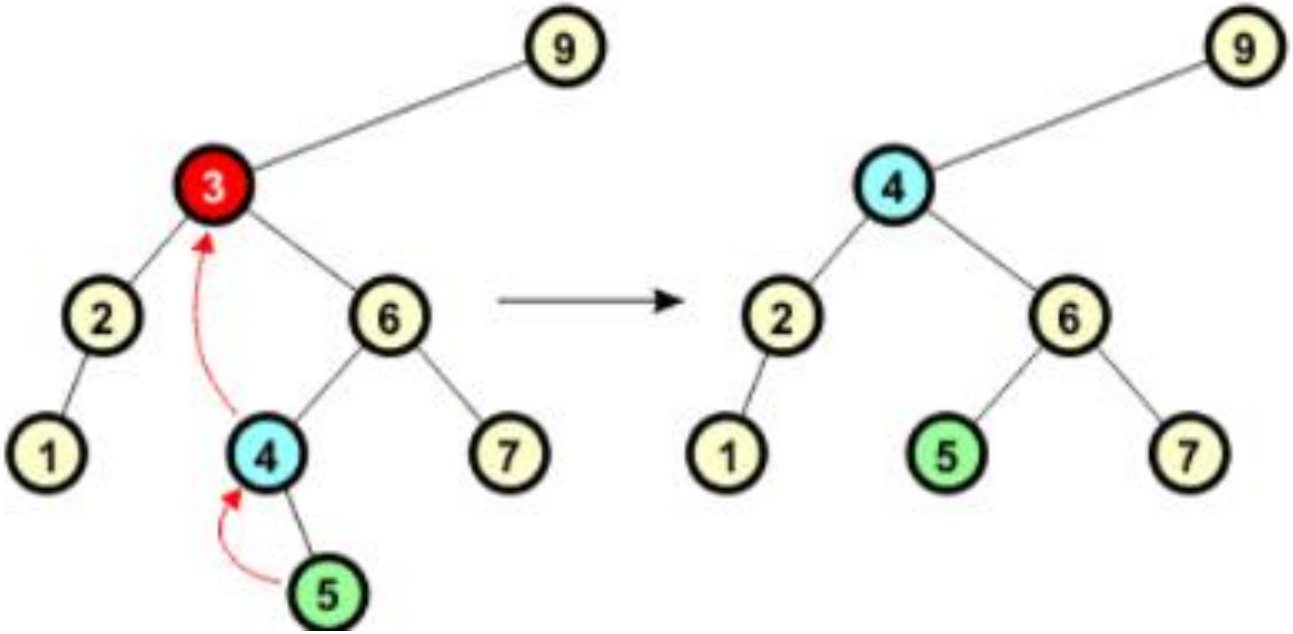
Przypadek 2

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



Usuwanie węzła

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

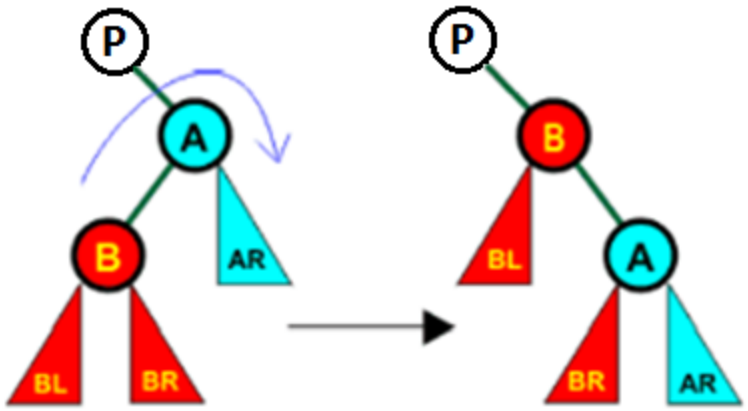


SDIZO-BST Usuwanie węzła

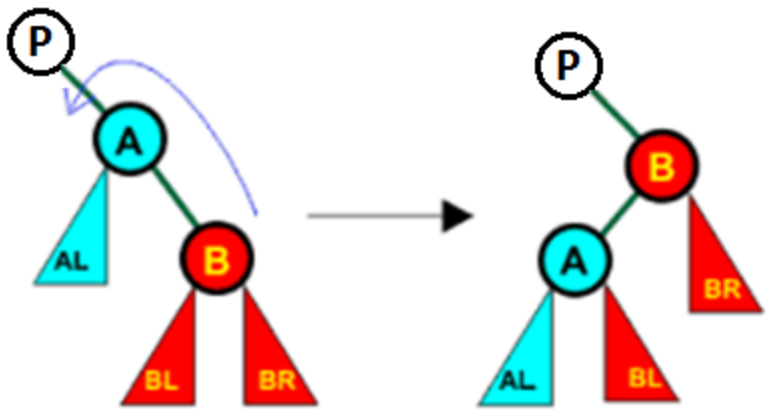
```
BST_TREE_DELETE (Root, DeleteNode):
    if (DeleteNode->Left==NULL) or (DeleteNode->Right==NULL)
        y=DeleteNode
    else
        y=BST_FIND_SUCCESSOR(DeleteNode)
    if (y->Left != NULL)    x=y->Left  //następnik może mieć tylko
      else     x=y->Right                  // co najwyżej jednego 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
```

SDIZO-BST Rotacje

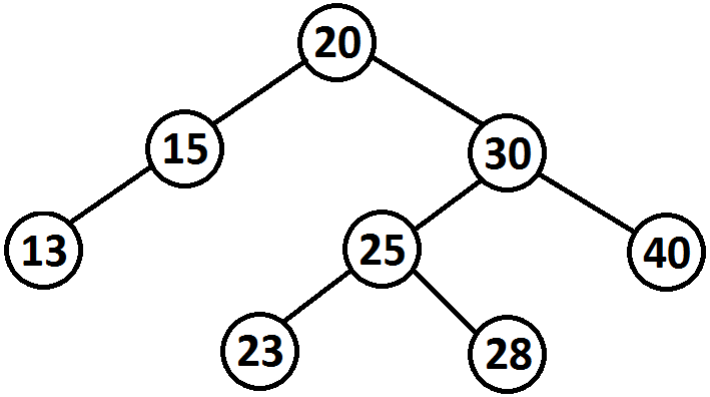
Rotacja w prawo

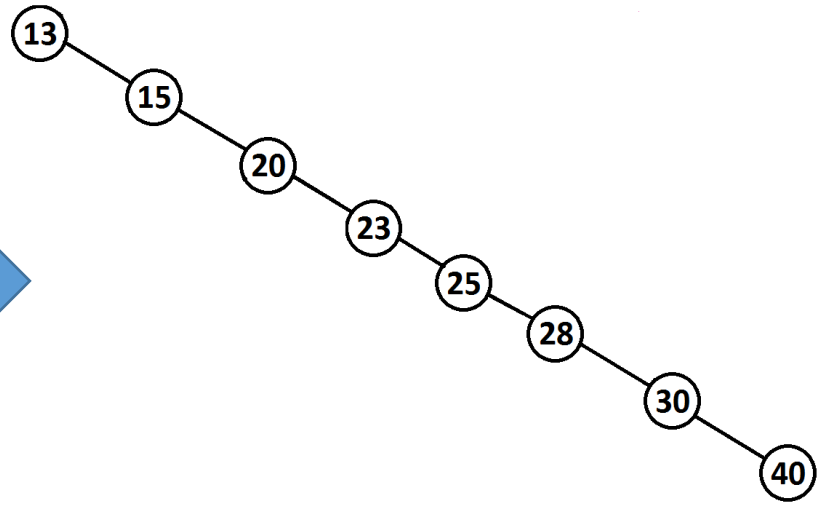


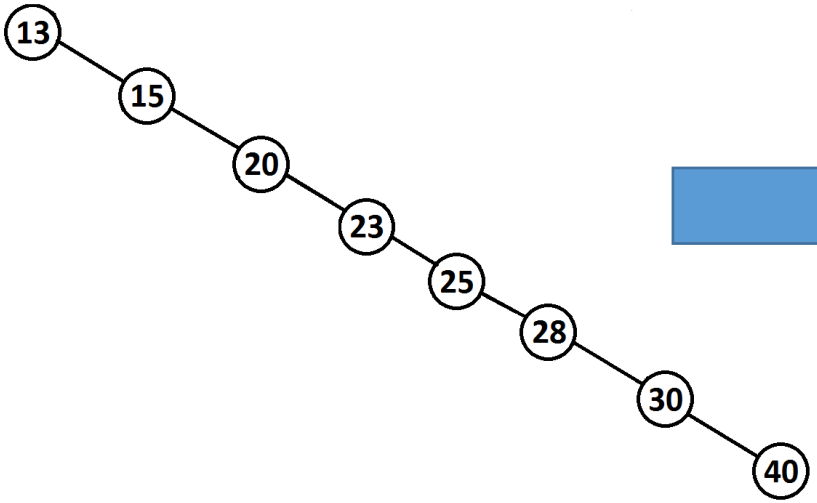
Rotacja w lewo

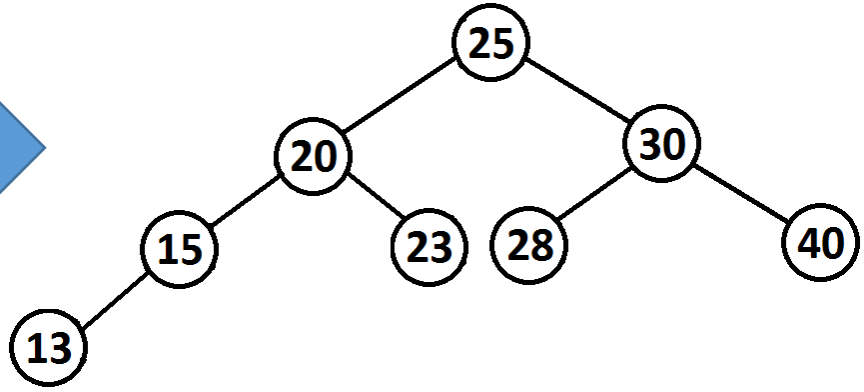


ETAP 1 – PROSTOWANIE
ETAP 2 - RÓWNOWAŻENIE





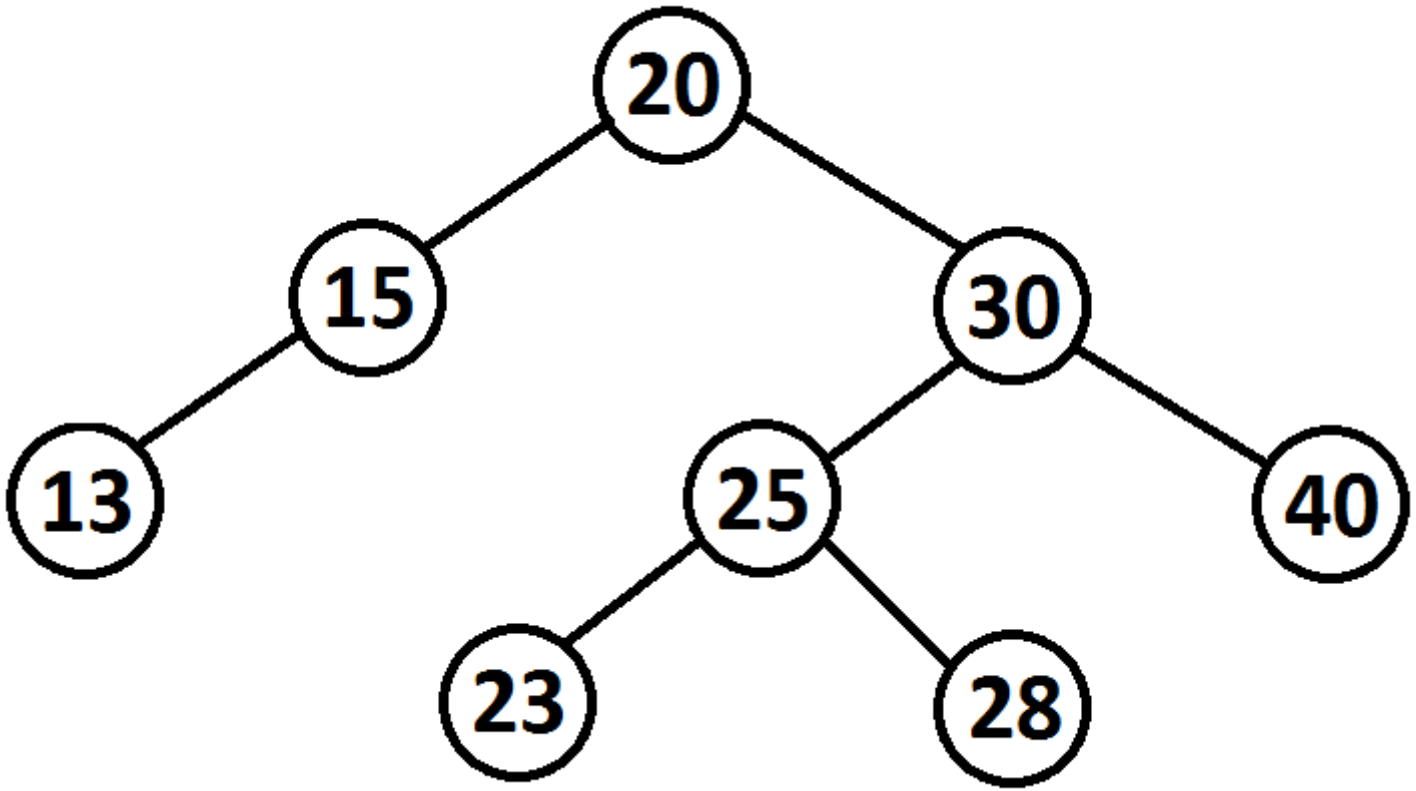


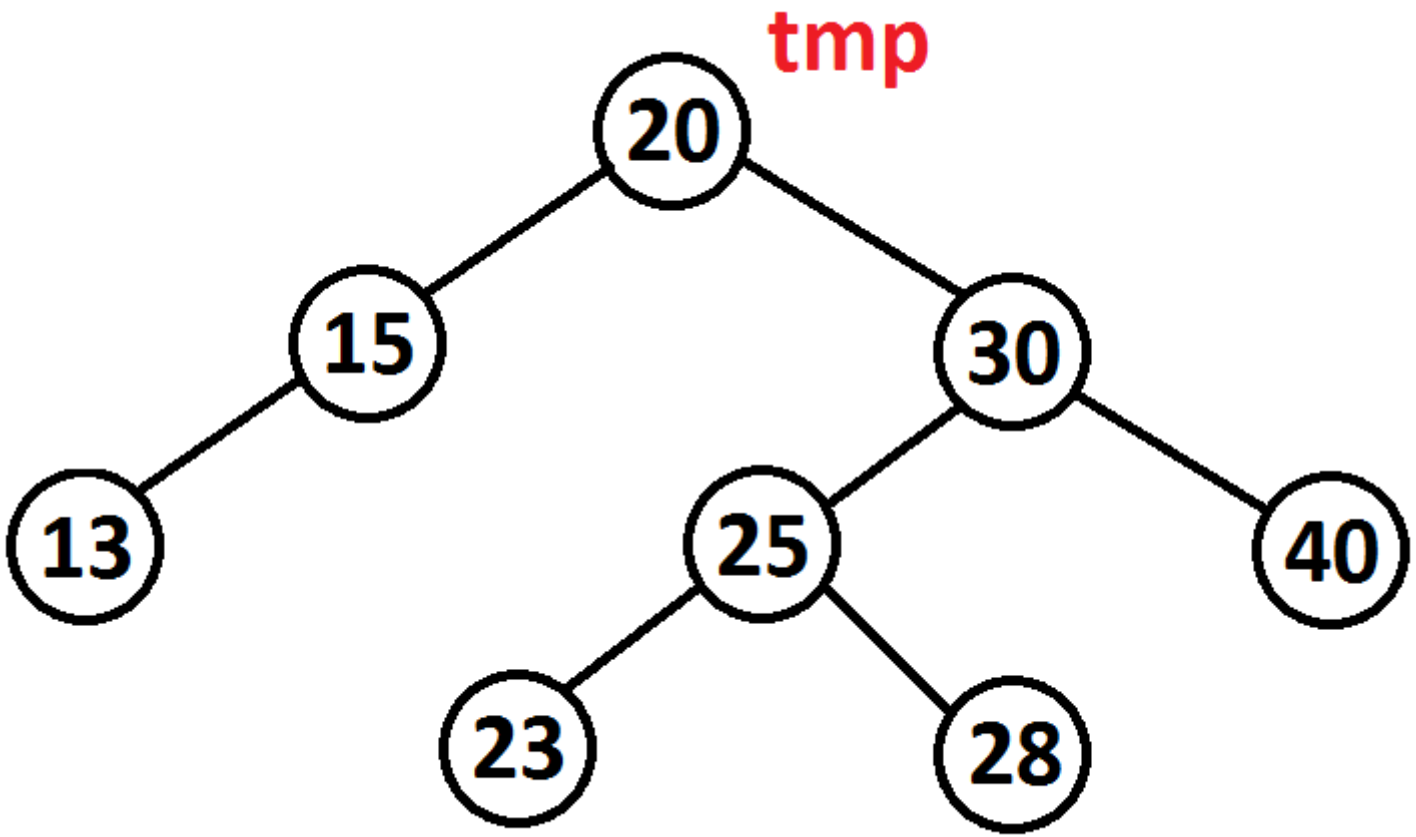


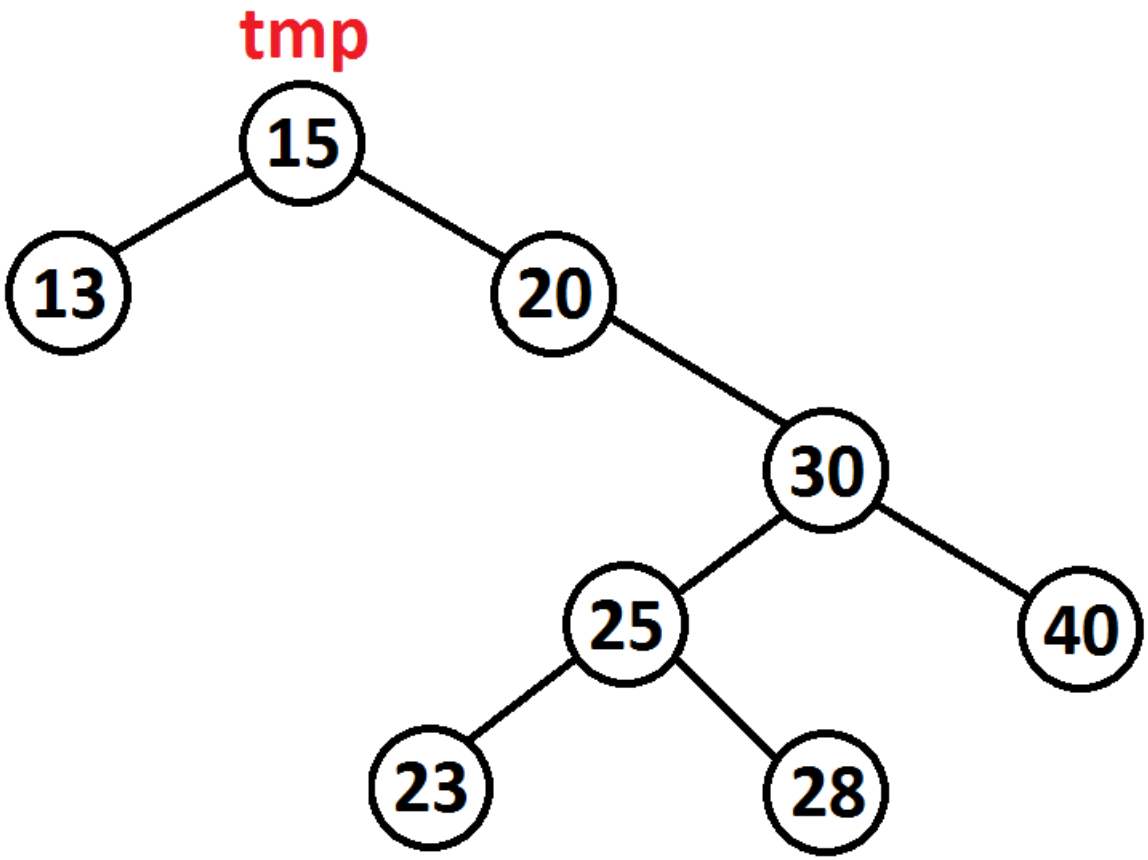
ETAP 1 – PROSTOWANIE

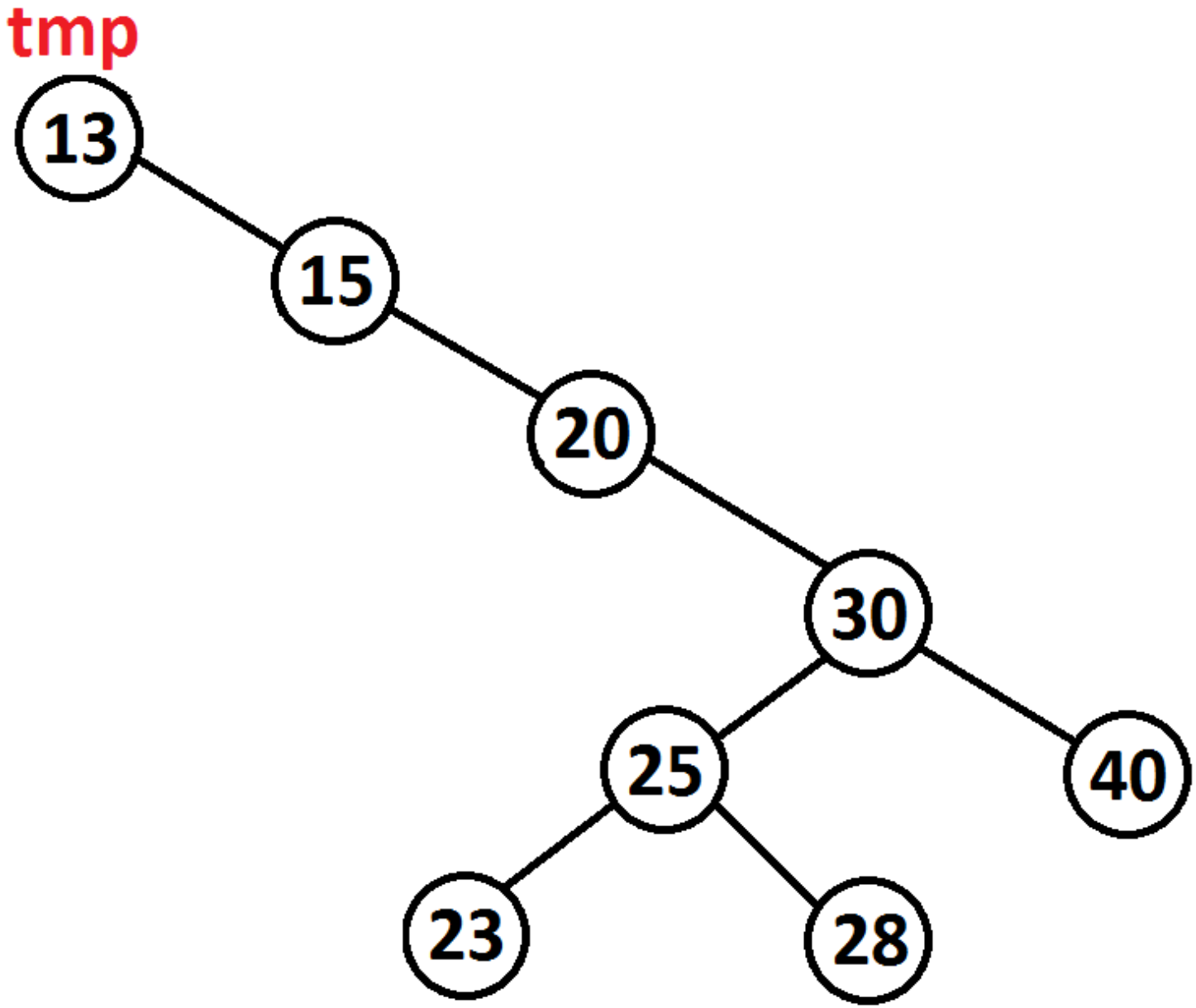
CreateLinearTree (Root)

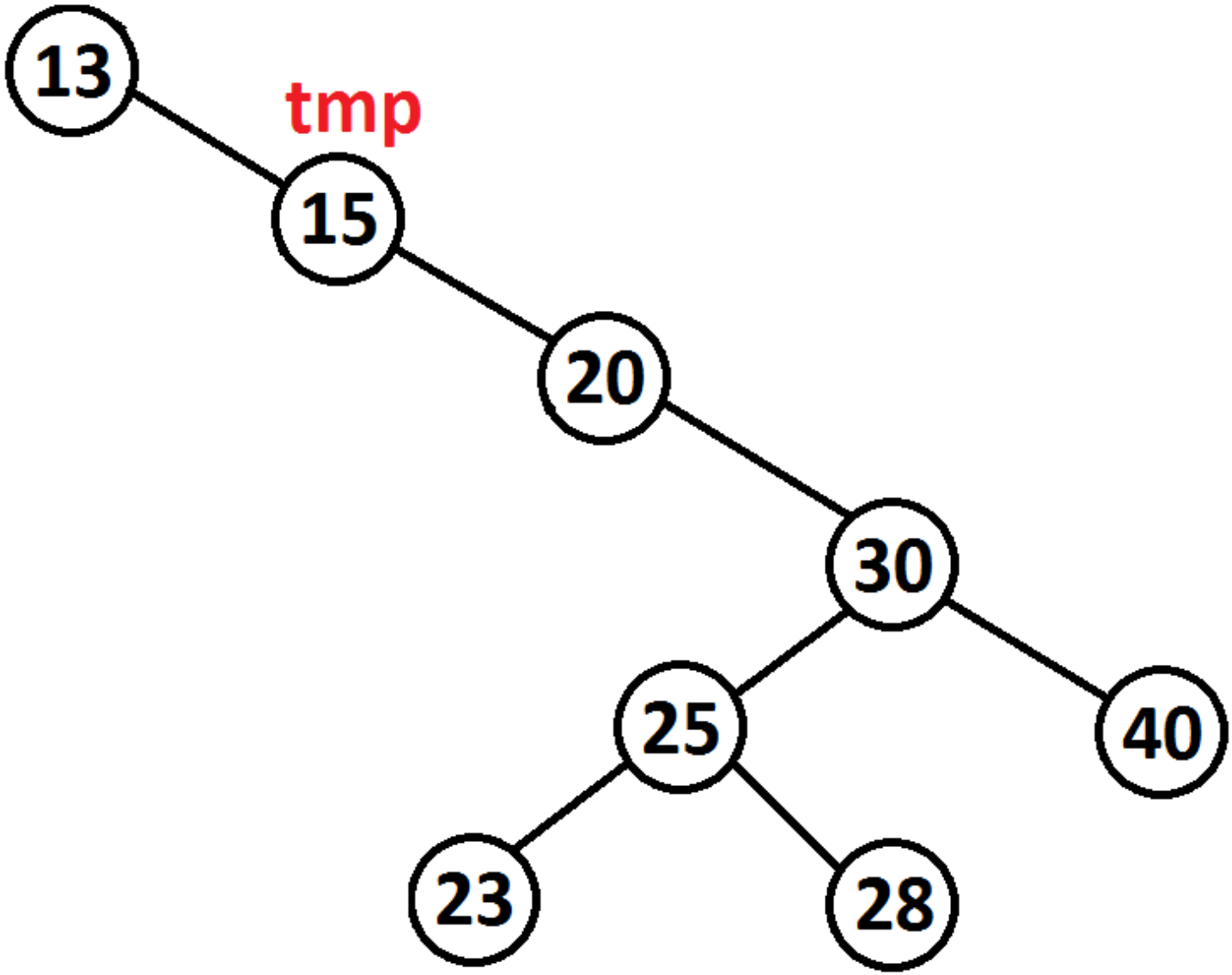
```
tmp = Root; //tmp to zmienna tymczasowa
while tmp != NULL
    if tmp->Left  != NULL
        wykonaj rotację w prawo względem tmp
        tmp = tmp->parent (po rotacji tmp przesunie się w
                           dół a ma zostać w tym samym miejscu)
    else
        tmp = tmp->Right
```

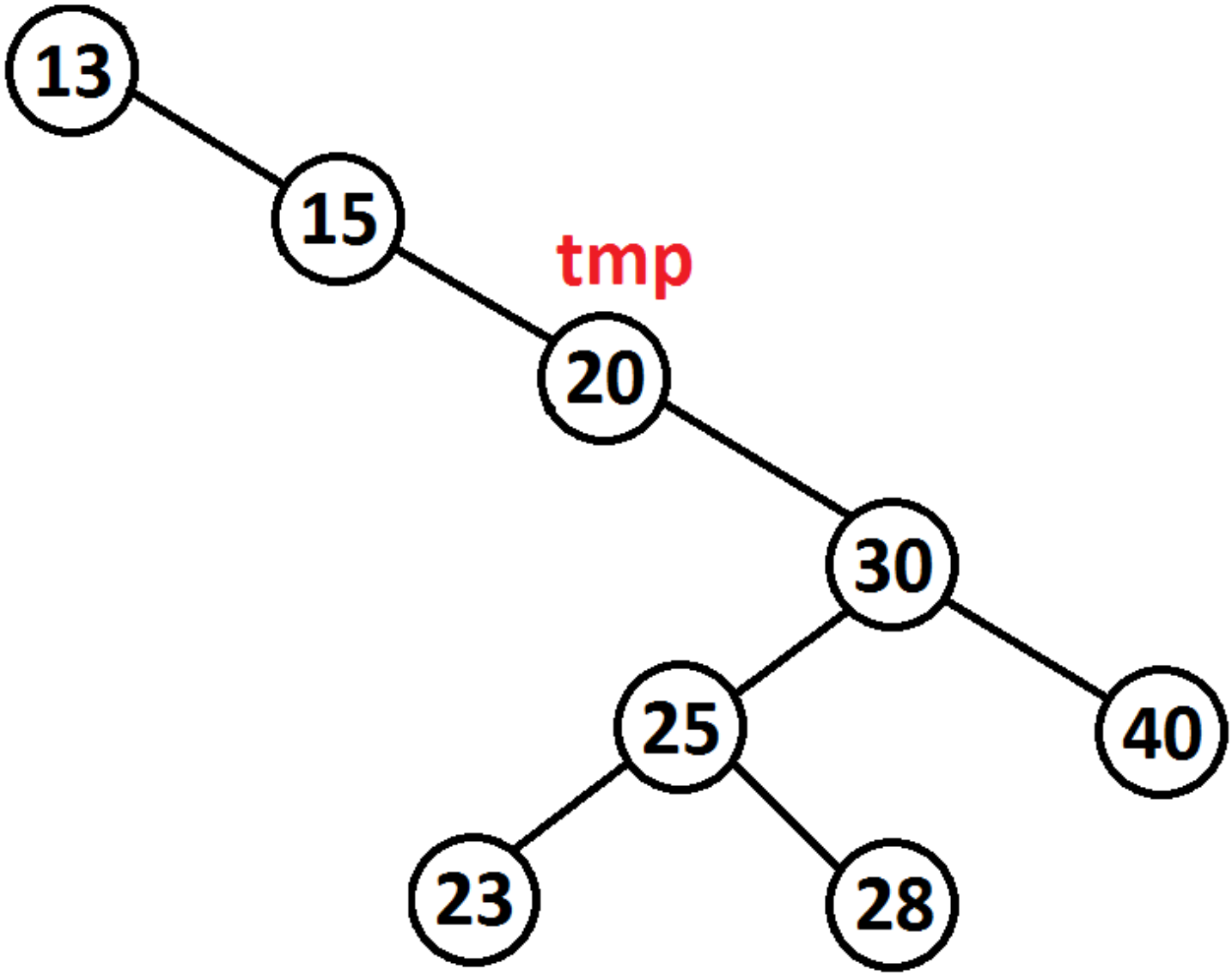

20
15
30
13
25
40
23
28

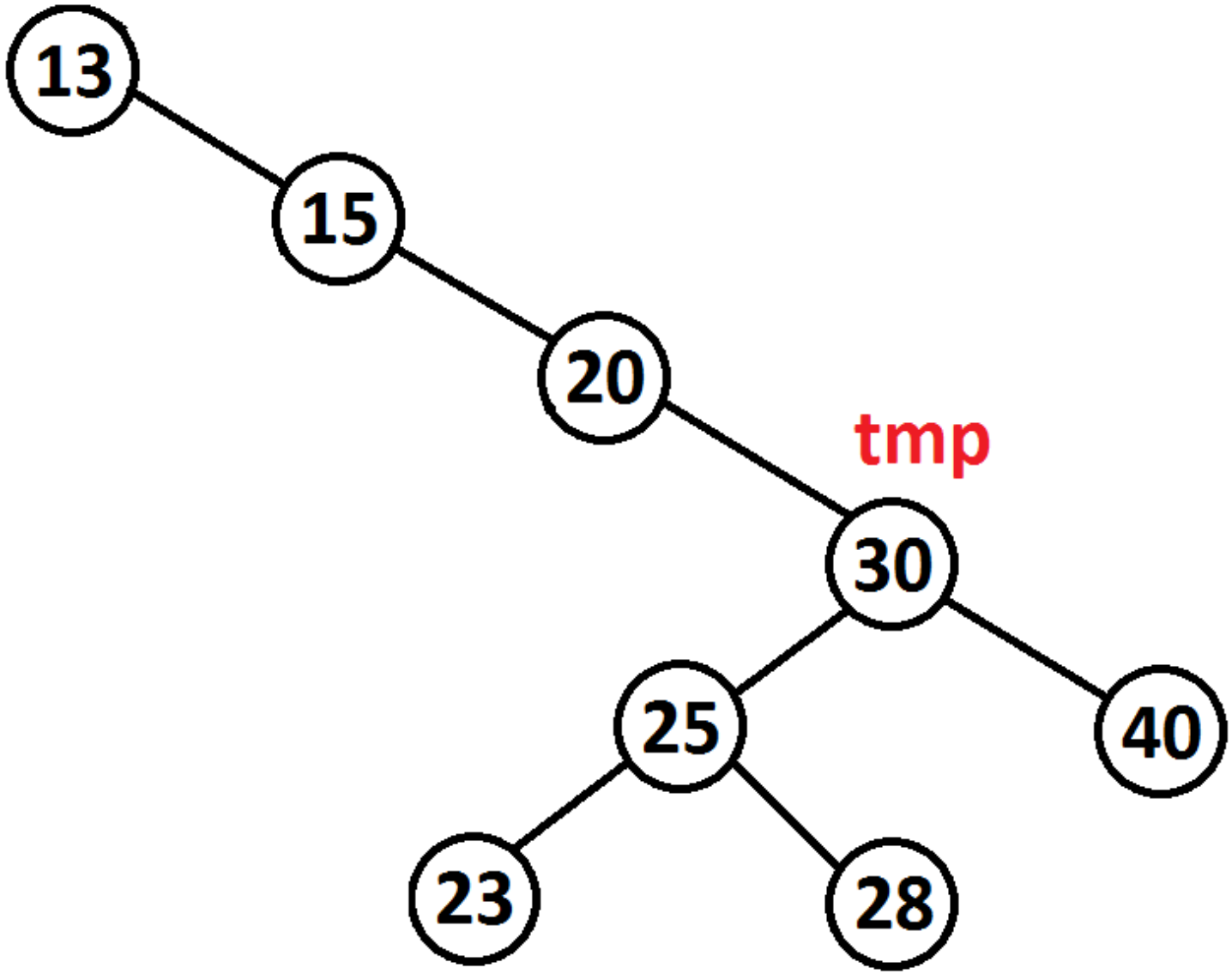

tmp
20
15
30
13
25
40
23
28

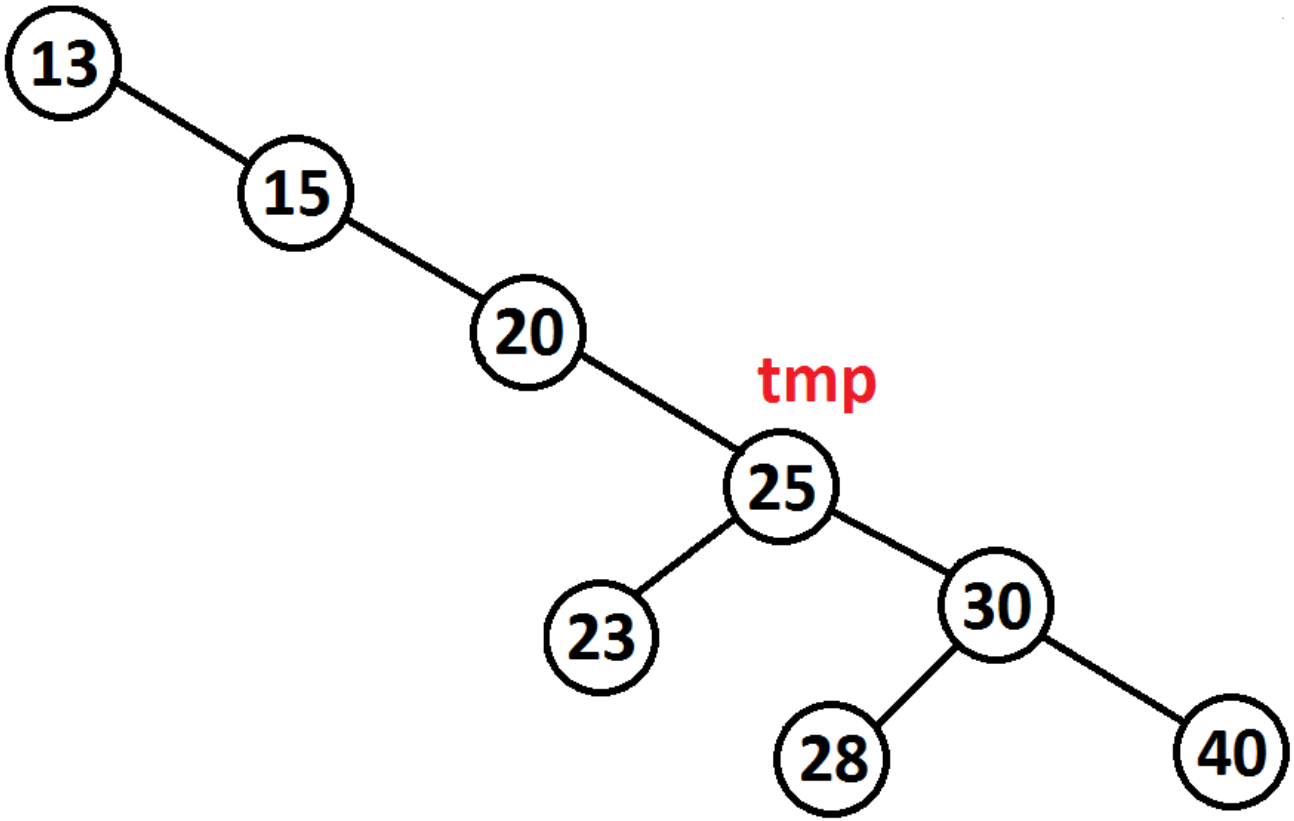

tmp
15
13
20
30
25
40
23
28

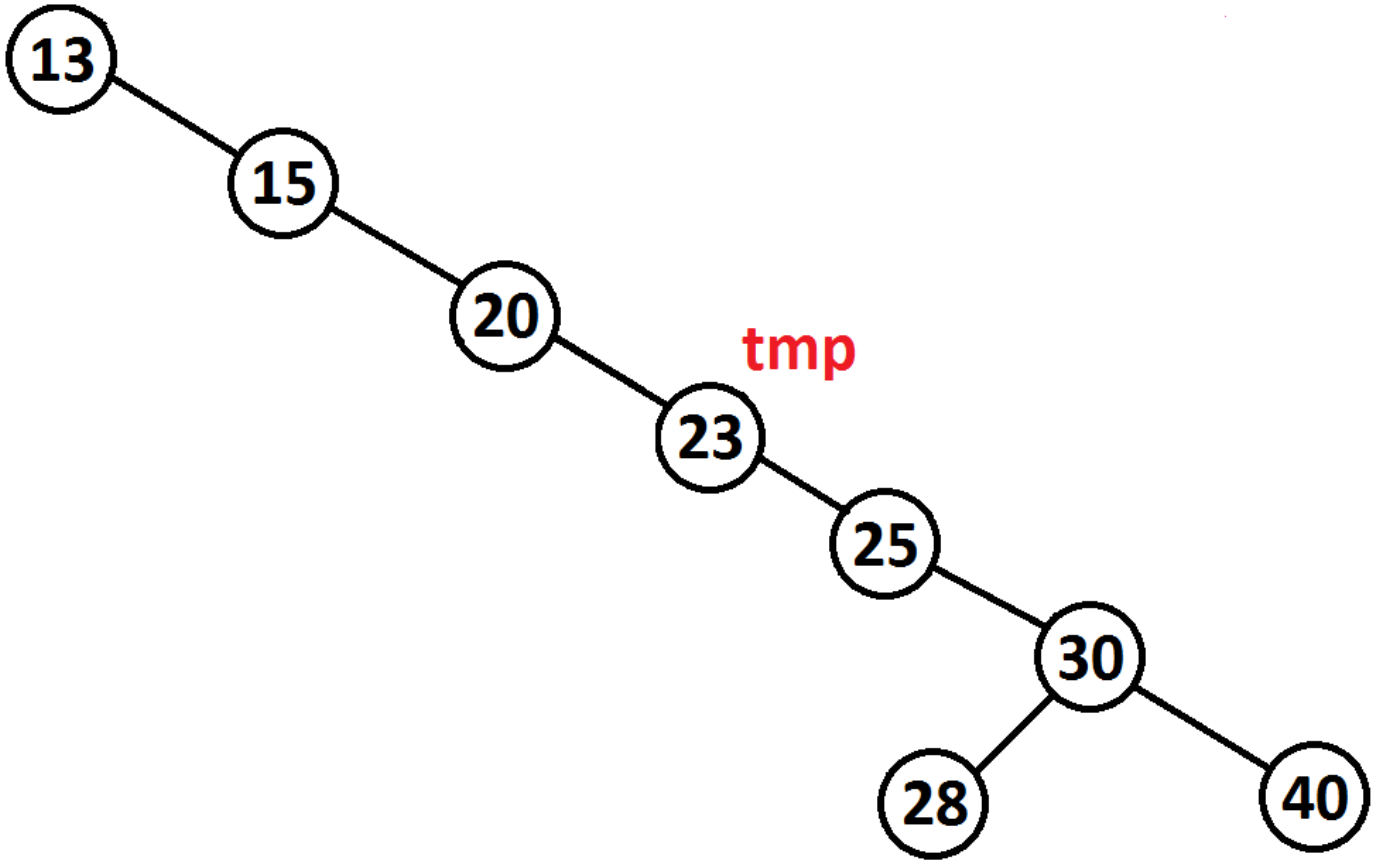

tmp
13
15
20
30
25
40
23
28

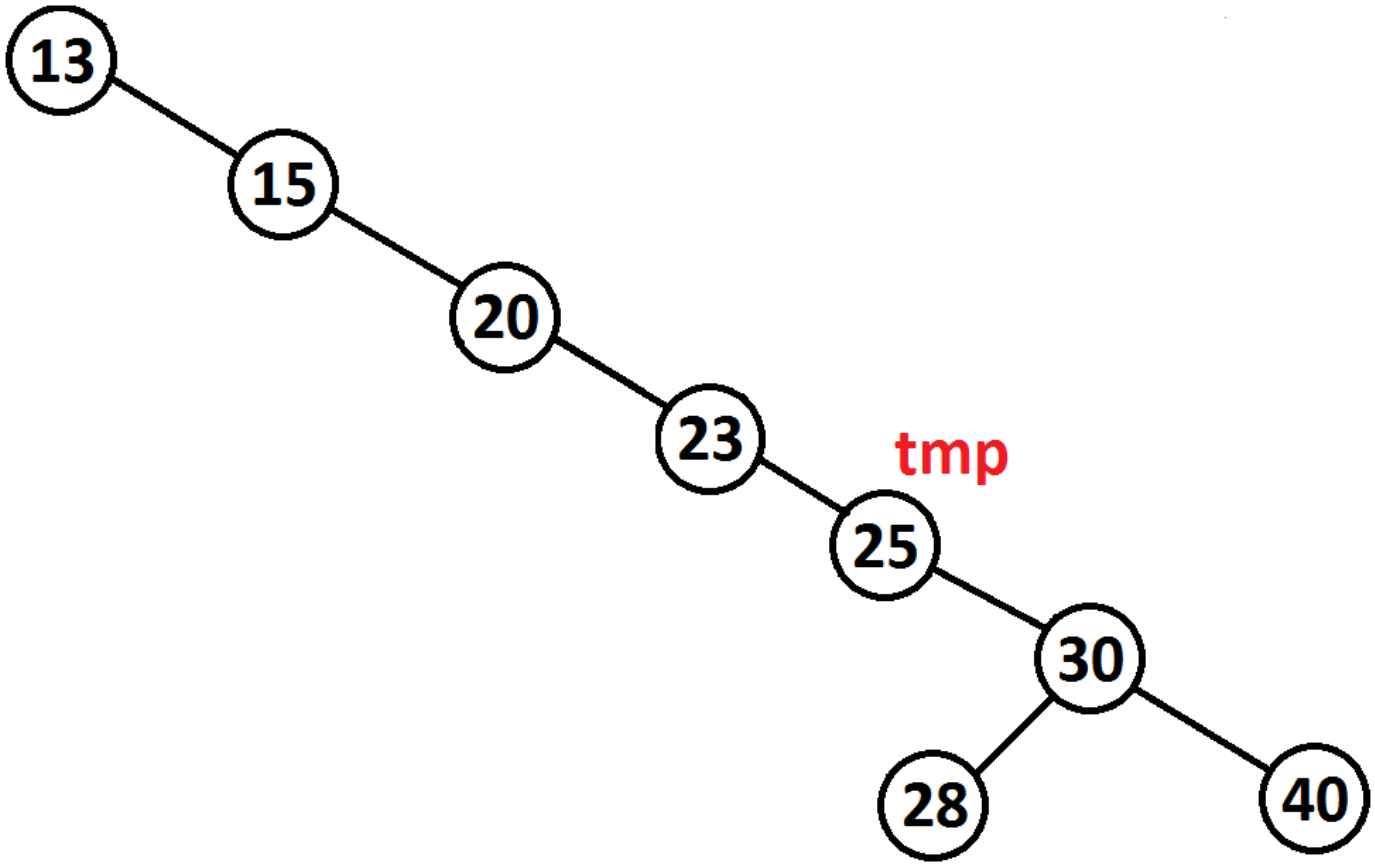

13
tmp
15
20
30
25
40
23
28

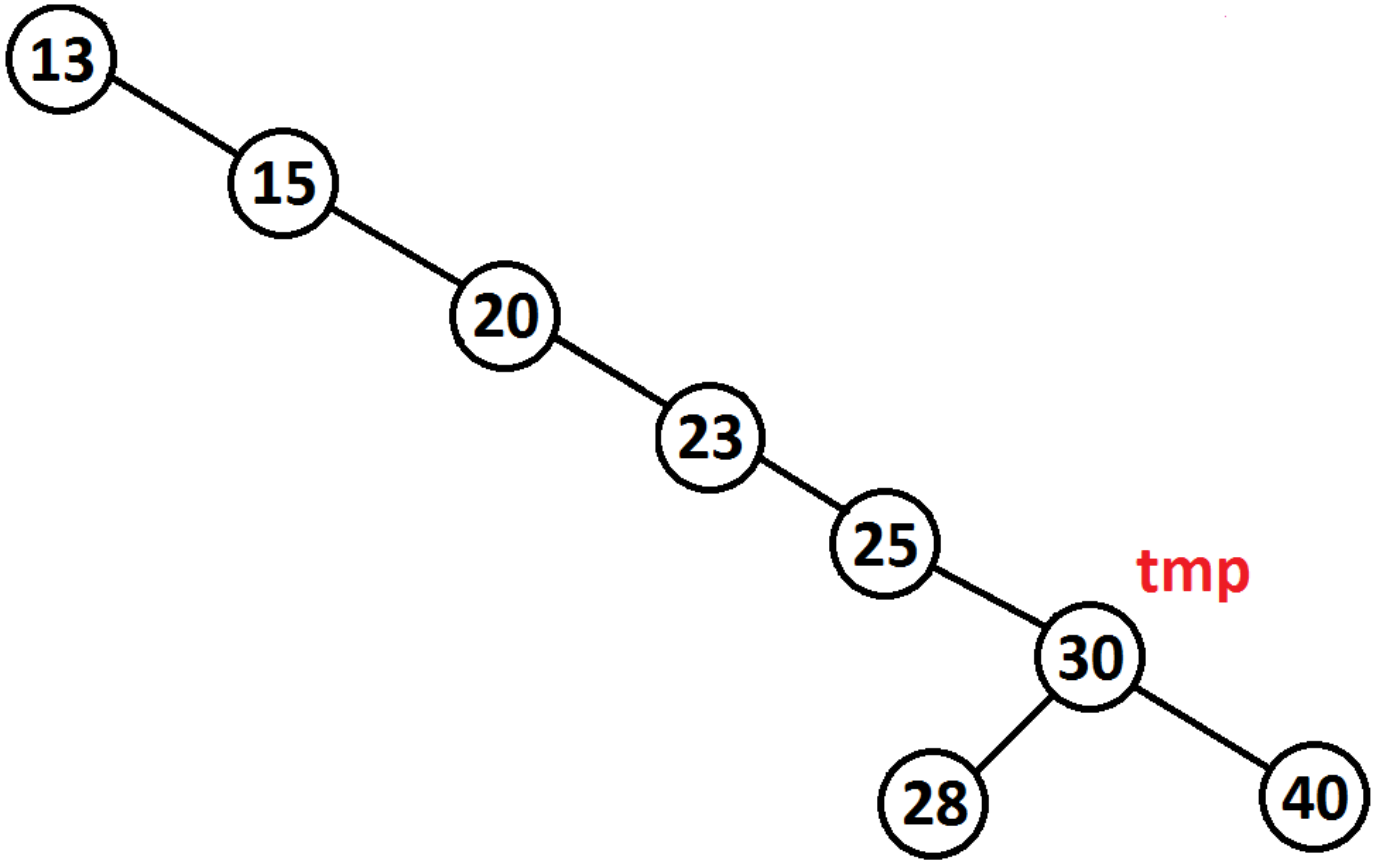

13
15
tmp
20
30
25
40
23
28

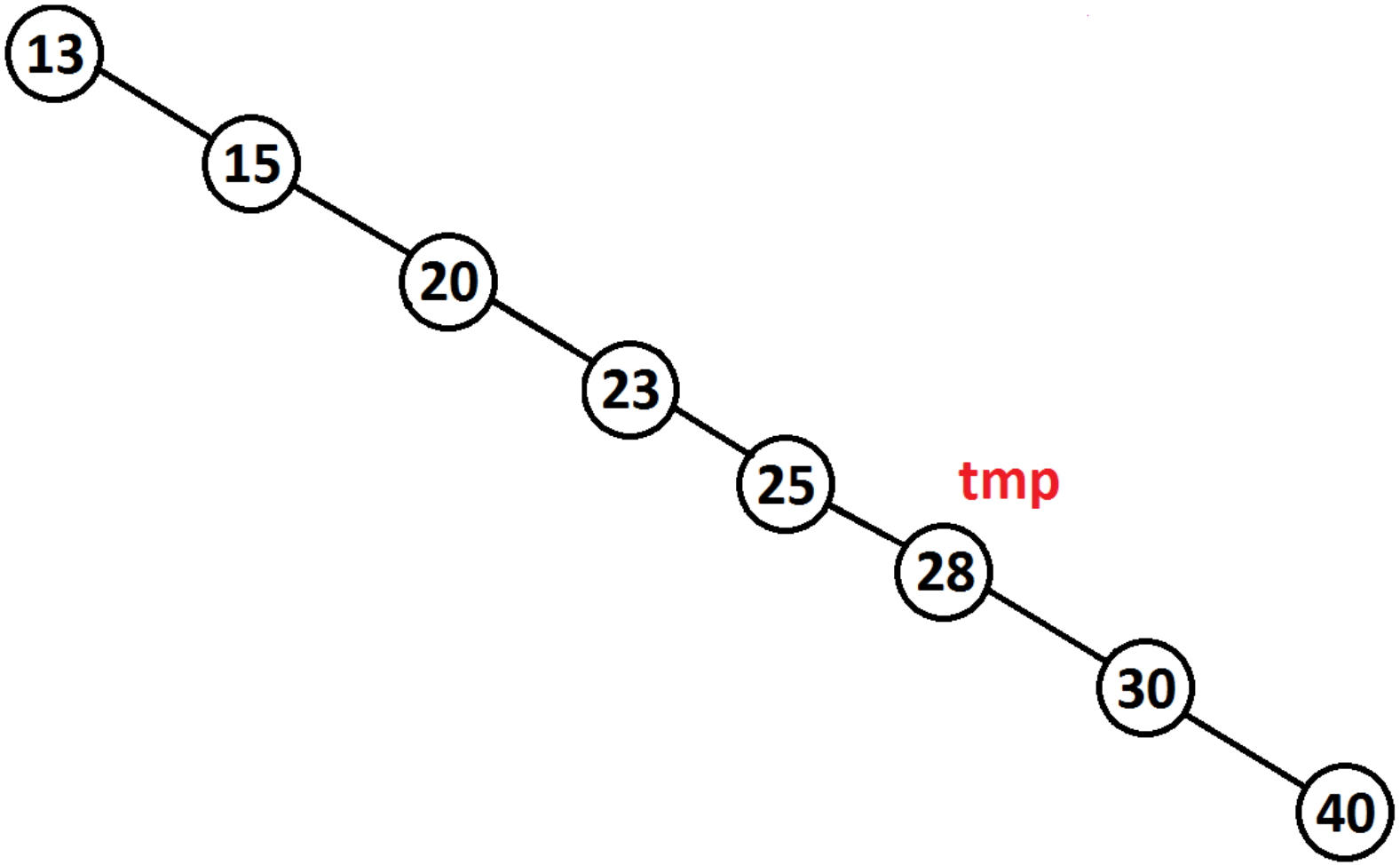

13
15
20
tmp
30
25
40
23
28

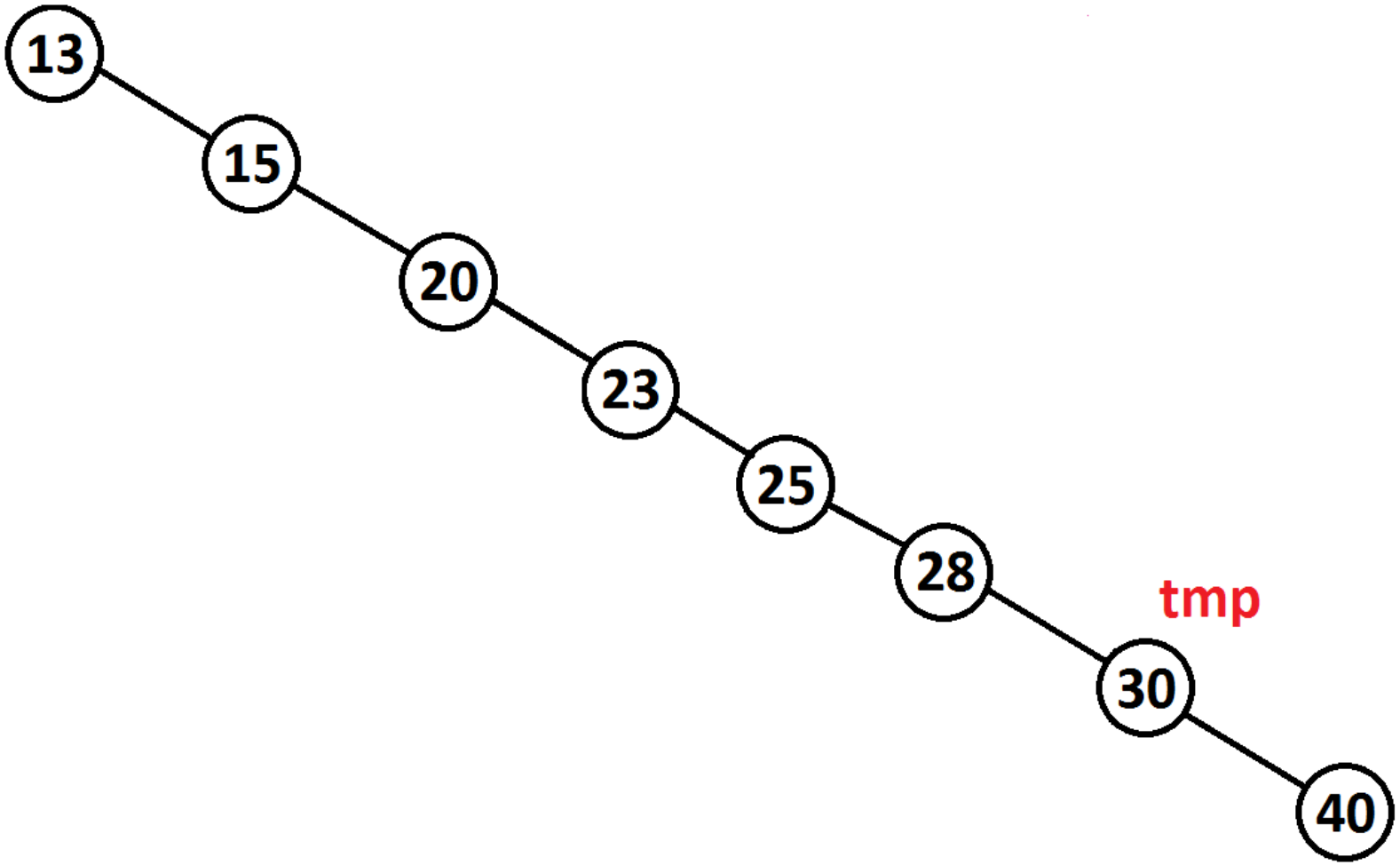

13
15
20
tmp
25
23
30
28
40

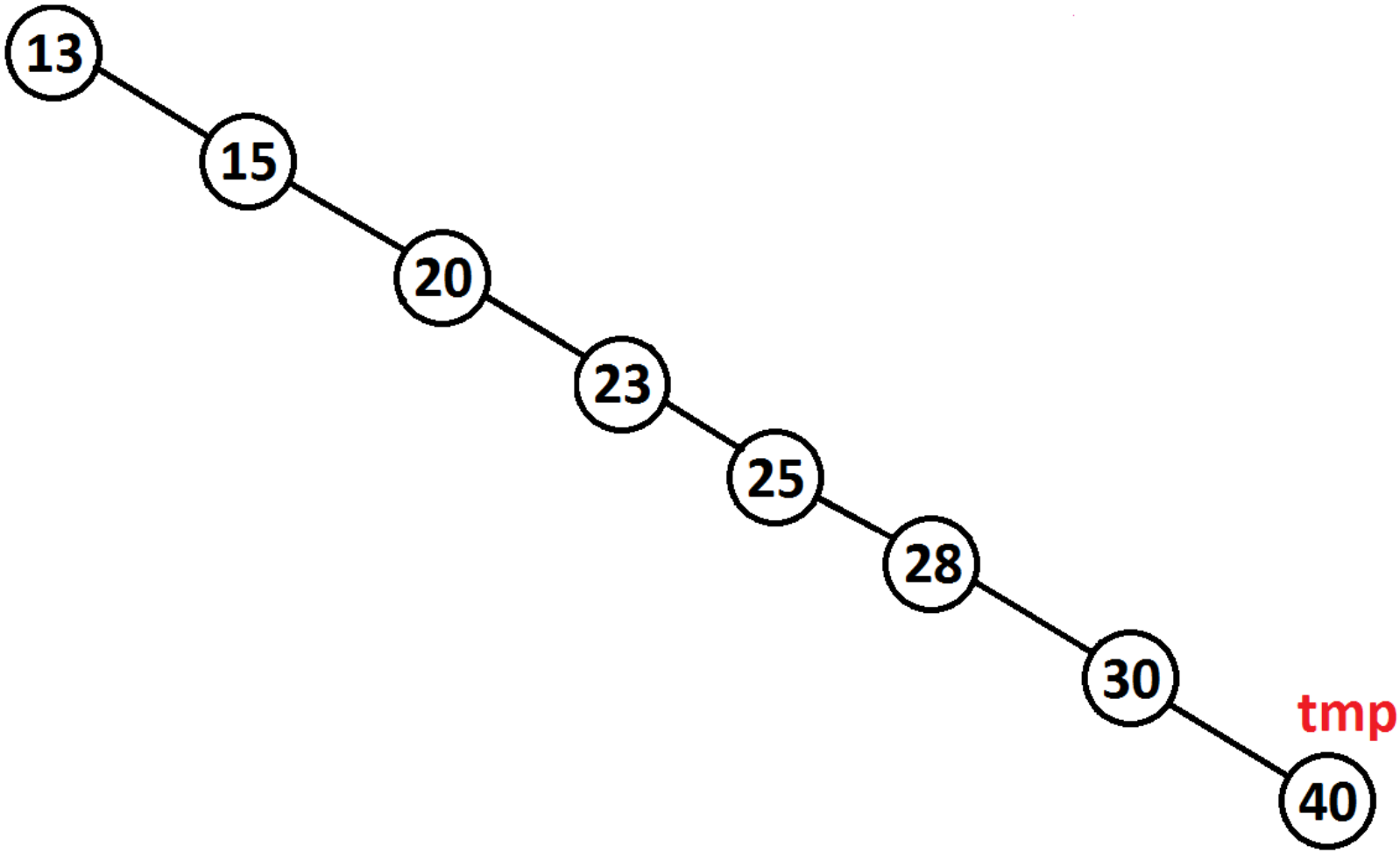

13
15
20
tmp
23
25
30
28
40

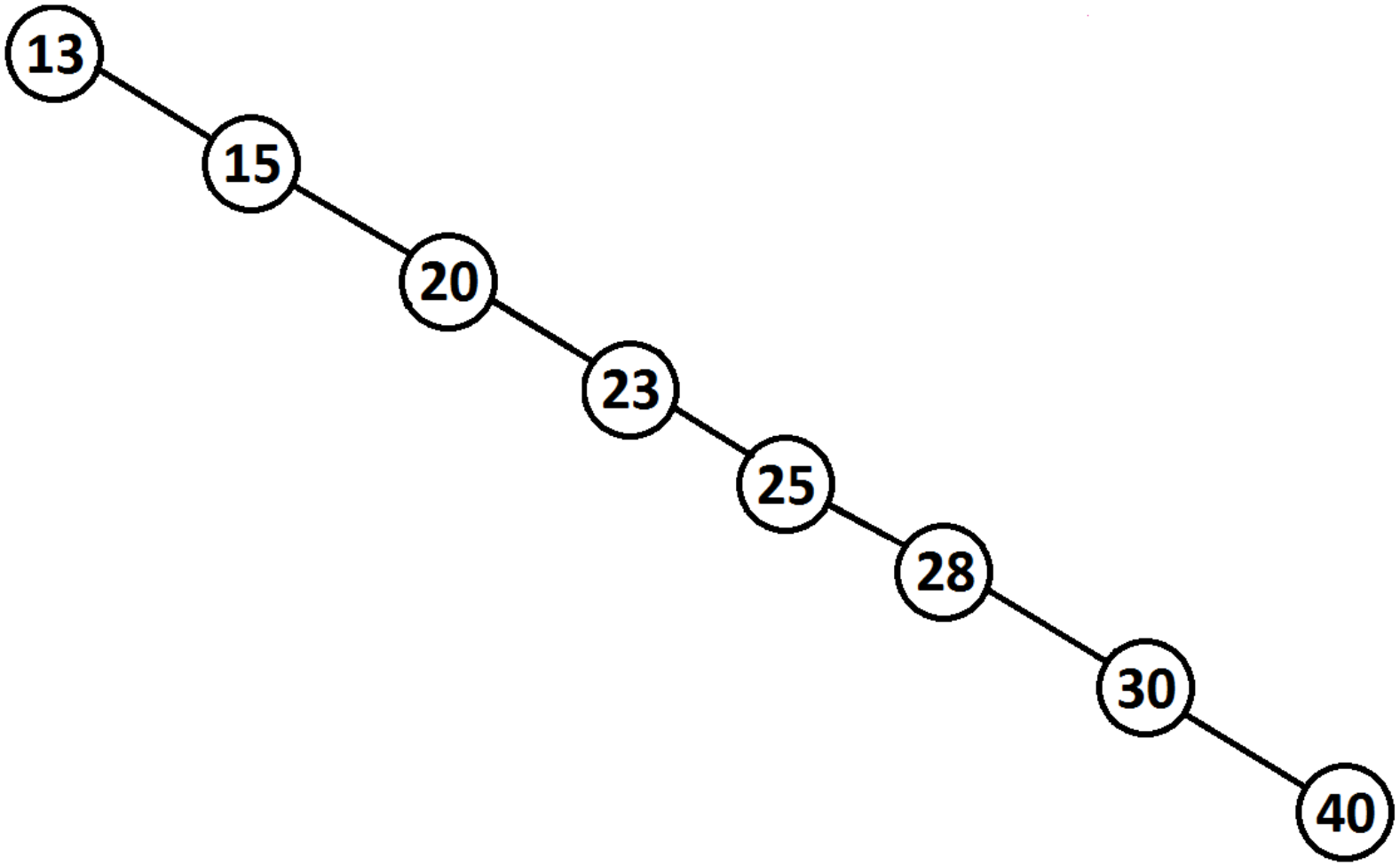

13
15
20
23
tmp
25
30
28
40


13
15
20
23
25
tmp
30
28
40


13
15
20
23
25
tmp
28
30
40


13
15
20
23
25
28
tmp
30
40


13
15
20
23
25
28
30
tmp
40



tmp =NULL

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

$m = \lfloor m/2 \rfloor$

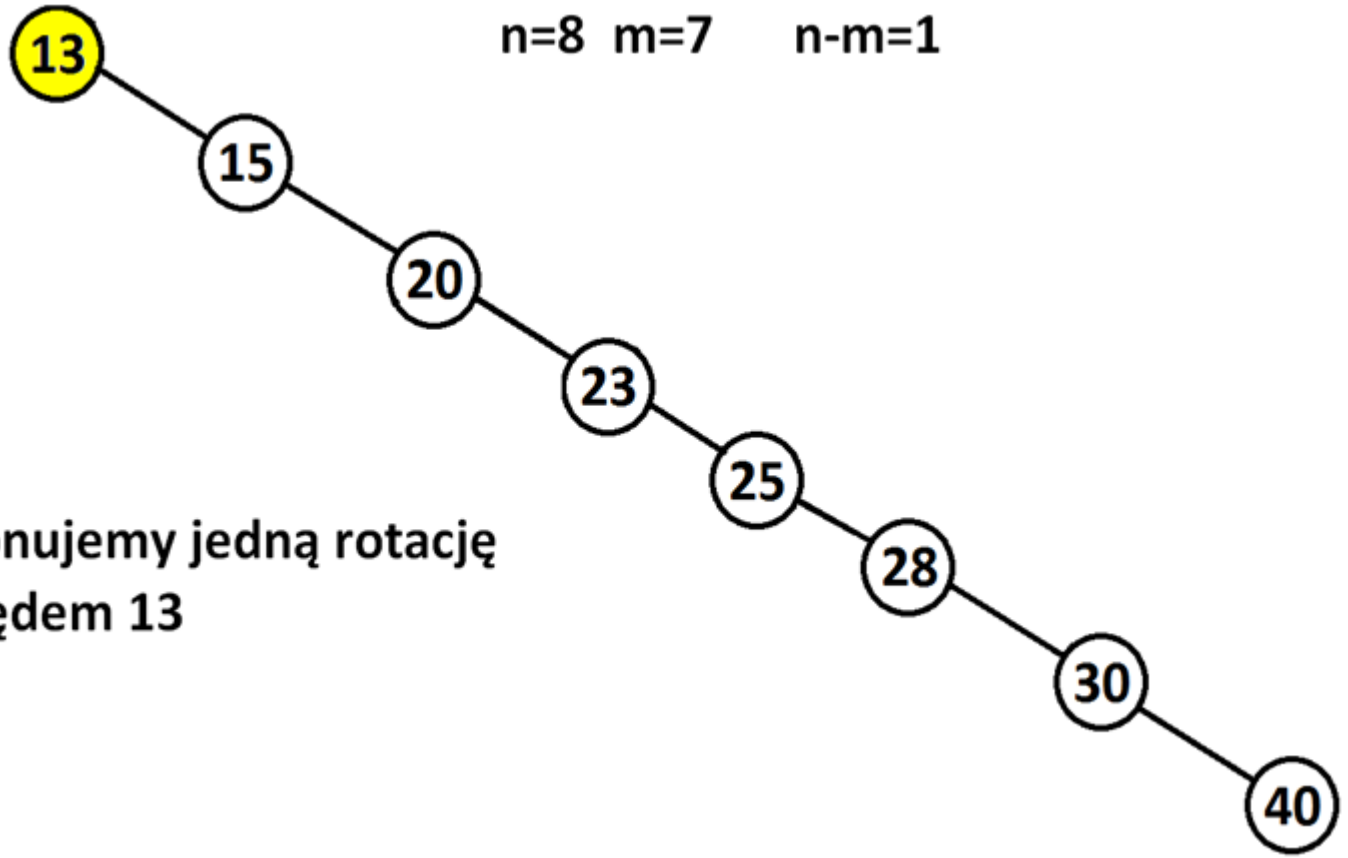
wykonaj m rotacji w lewo , startując od początkowego wierzchołka co drugi wierzchołek

m – ilość wierzchołków w drzewie w części zapełnionej

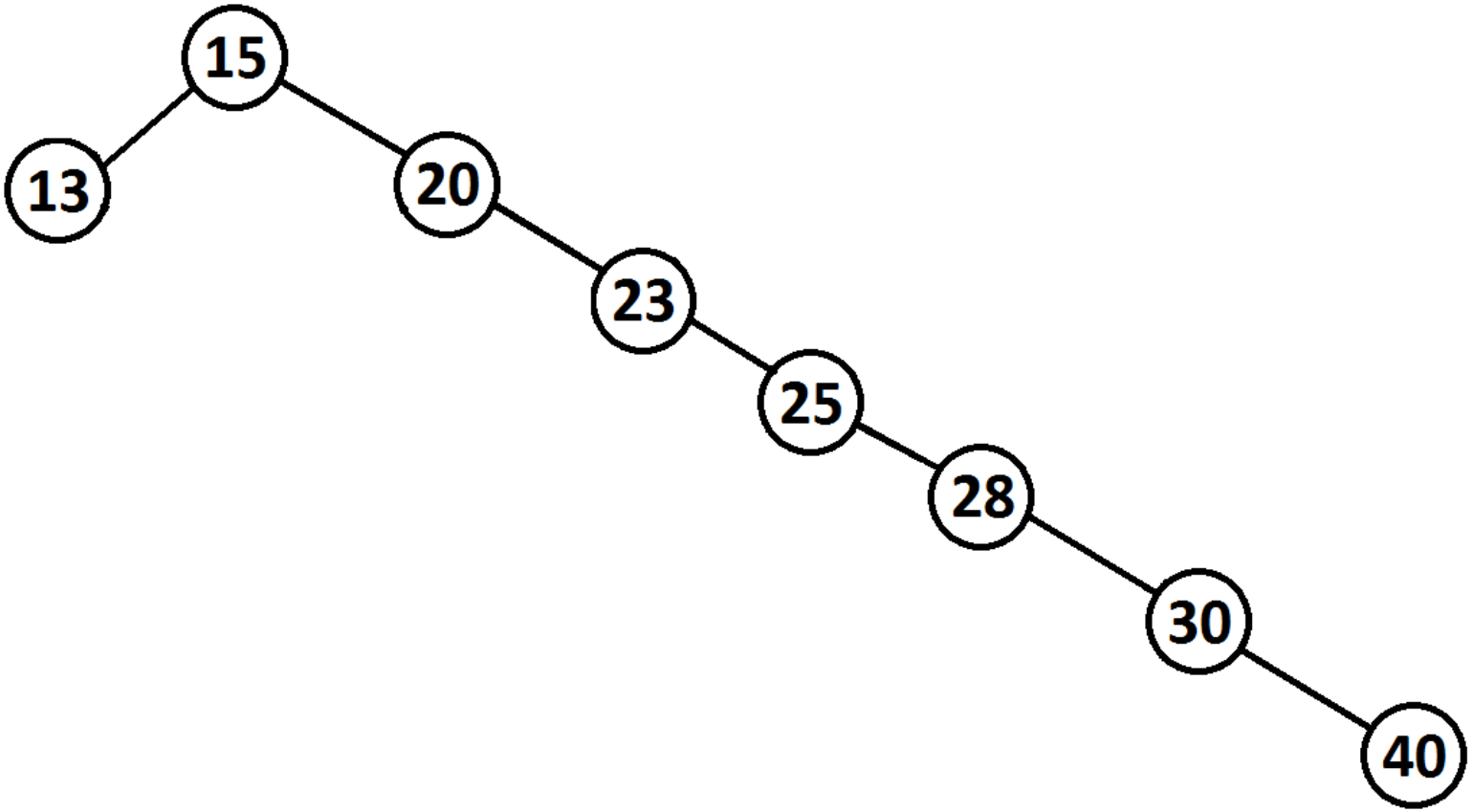
SDIZO-BST Algorytm DSW

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

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



po wykonaniu n-m rotacji



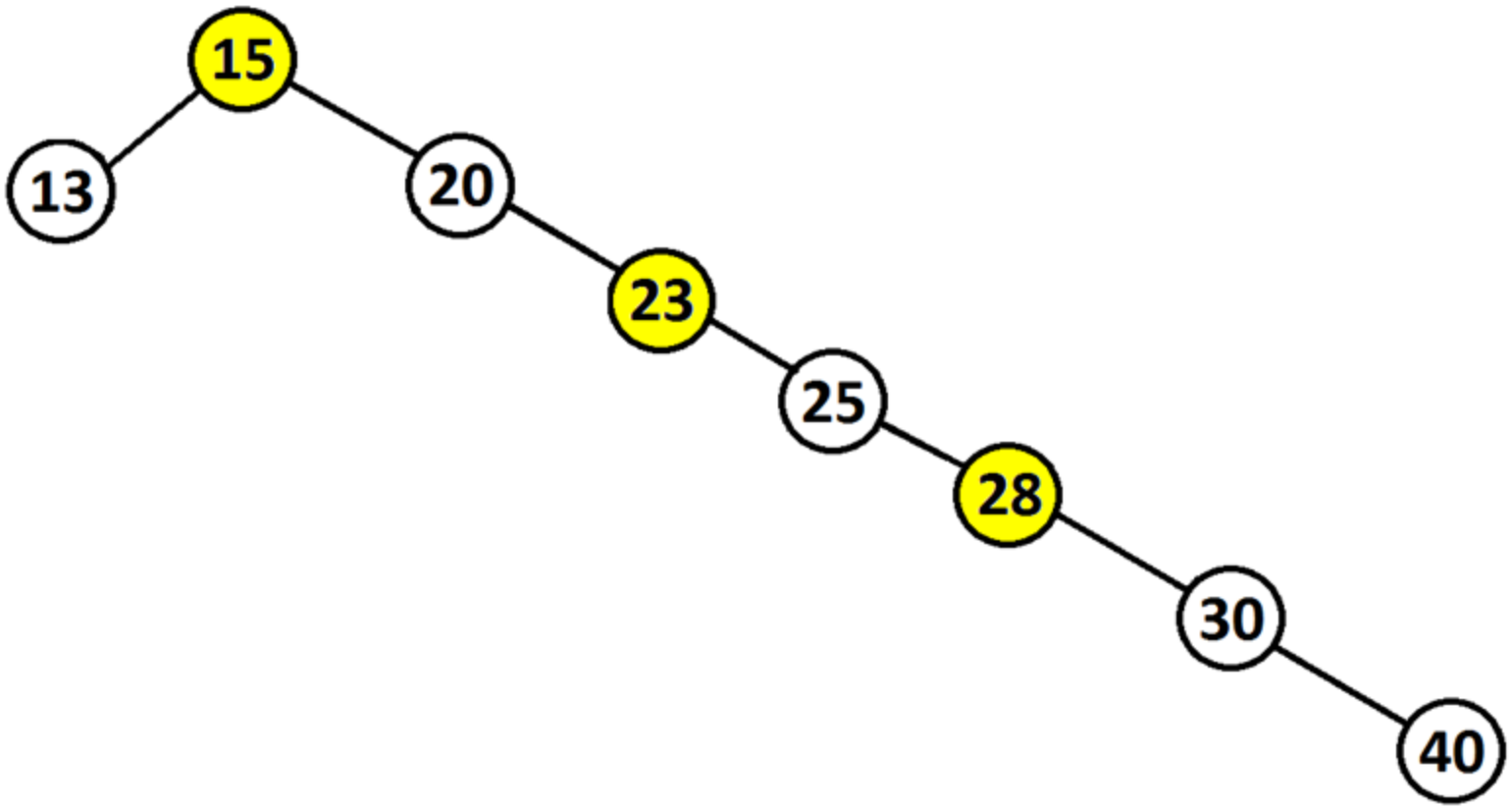
CreateBalancedTree (Root, n) //n-liczba węzłów

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

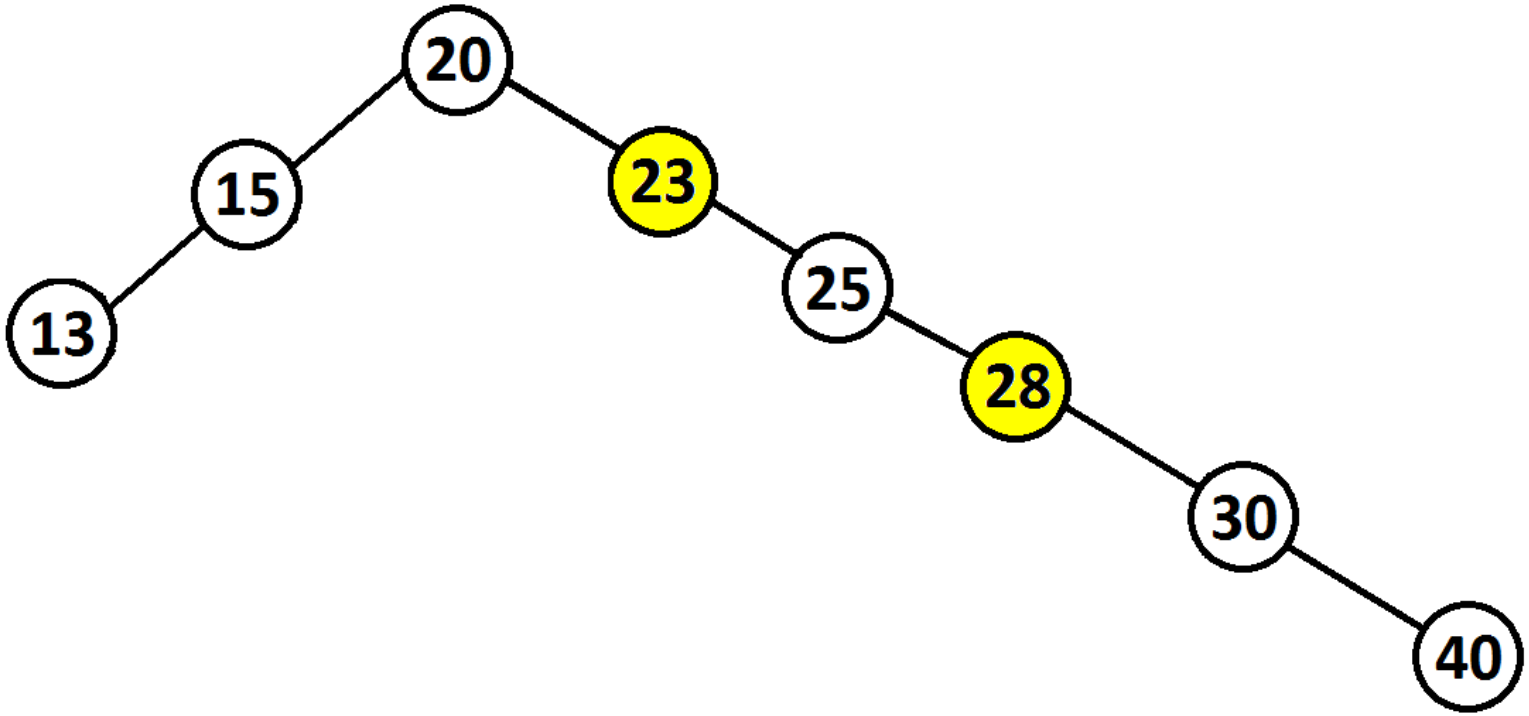
wykonaj **n-m** rotacji w lewo , startując od początkowego wierzchołka co drugi wierzchołek

```
while m > 1
        m = ⌊ m/2 ⌋
        wykonaj m rotacji w lewo , startując
        od początkowego wierzchołka co drugi wierzchołek
```

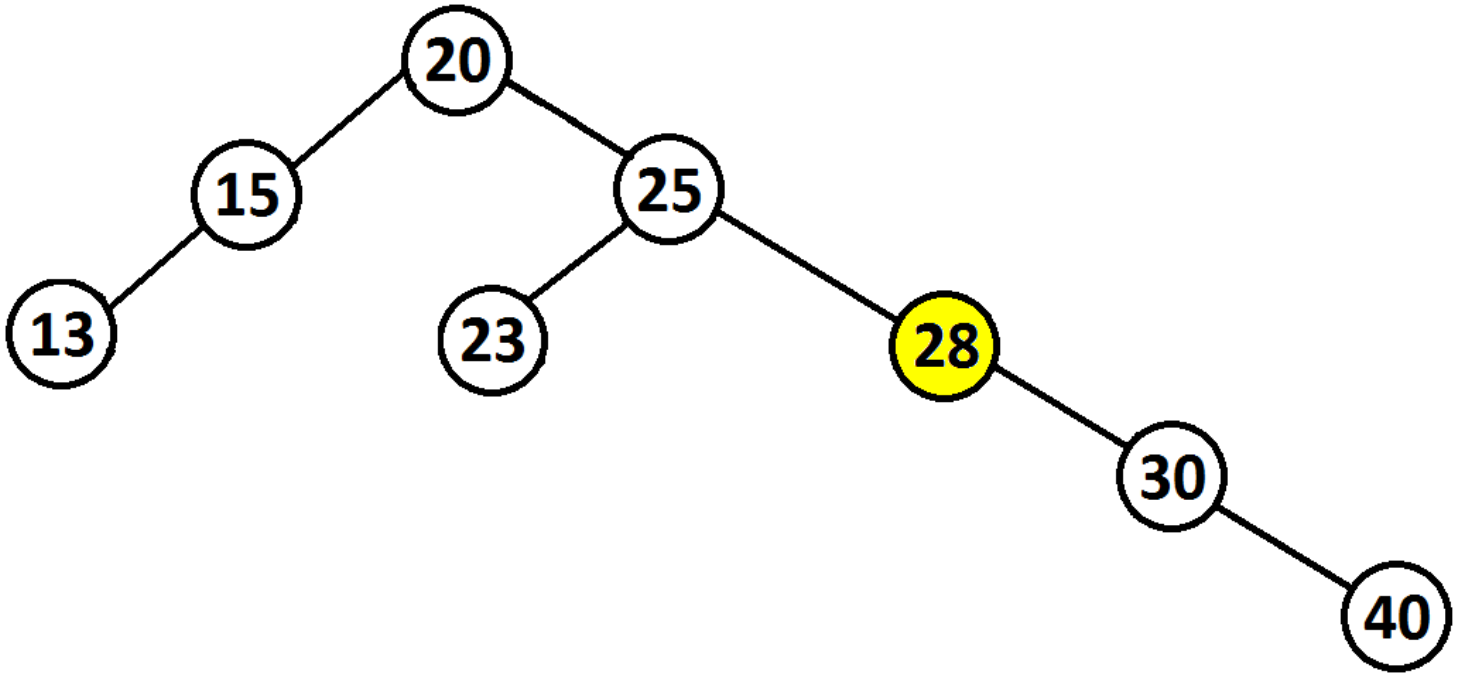
m=⌊m/2⌋ = ⌊7/2⌋ = 3 -> wykonujemy 3 rotacje

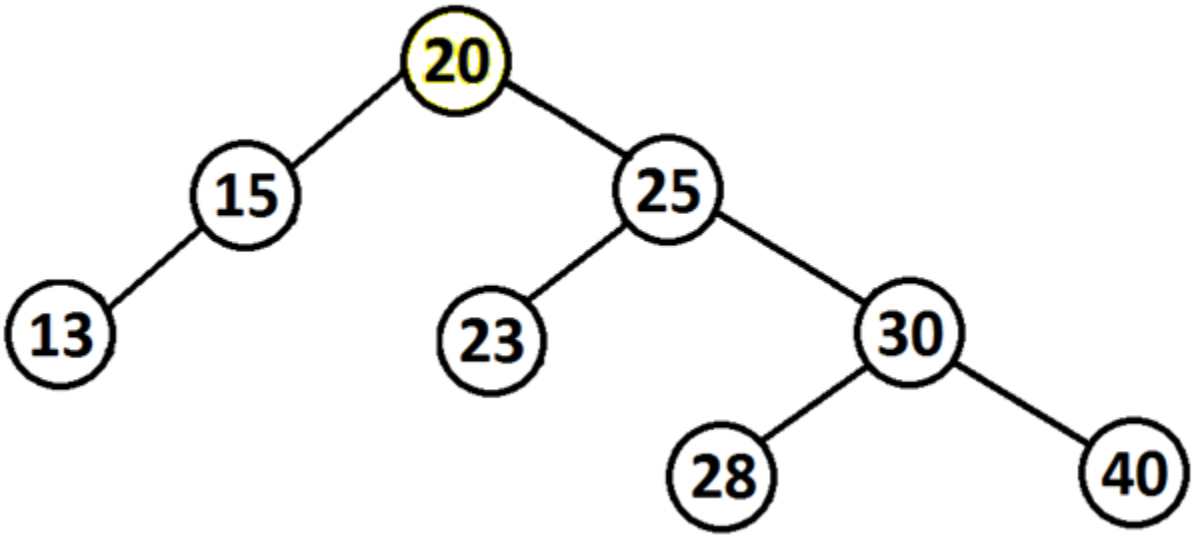


wykonujemy kolejna rotację względem 23



wykonujemy 3 rotację (względem 28)

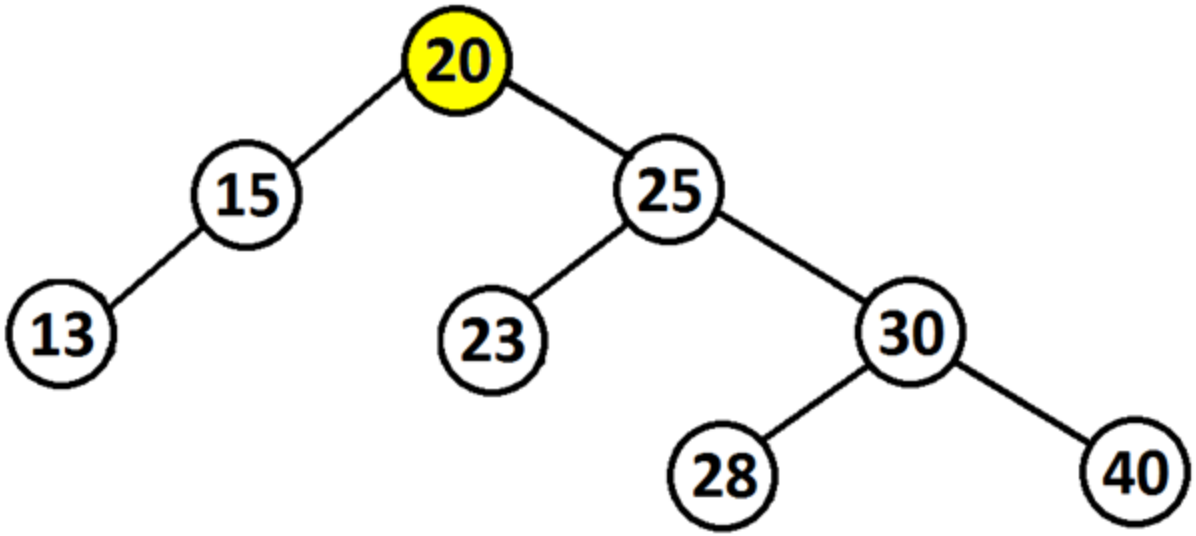



20
15
25
13
23
30
28
40

Algorytm DSW

```
while m > 1
        m = ⌊ m/2 ⌋
        wykonaj m rotacji w lewo , startując
        od początkowego wierzchołka co drugi wierzchołek
```

m=[m/2] = [3/2]=1 => wykonujemy 1 rotację



po wykonaniu wszystkich rotacji - drzewo zrównoważone