DATA STRUCTURES AND ALGORITHMS

Hierarchical data structures:

AVL tree, Bayer tree, Heap

Summary of the previous lecture

- TREE is hierarchical (non linear) data structure
- Binary trees
  - Definitions
  - Full tree, complete tree
  - Enumeration (preorder, inorder, postorder)
- Binary search tree (BST)

AVL tree

The **AVL tree** (named for its inventors Adelson-Velskii and Landis published in their paper "**An algorithm for the organization of information**“ in 1962) should be viewed as a **BST** with the following additional property:

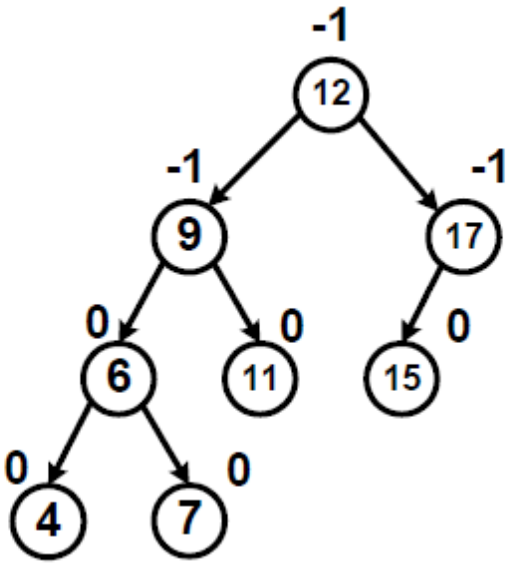
- For every node, the heights of its left and right subtrees differ by at most 1.

Difference of the subtrees height is named balanced factor.
A node with balance factor 1, 0, or -1 is considered balanced.

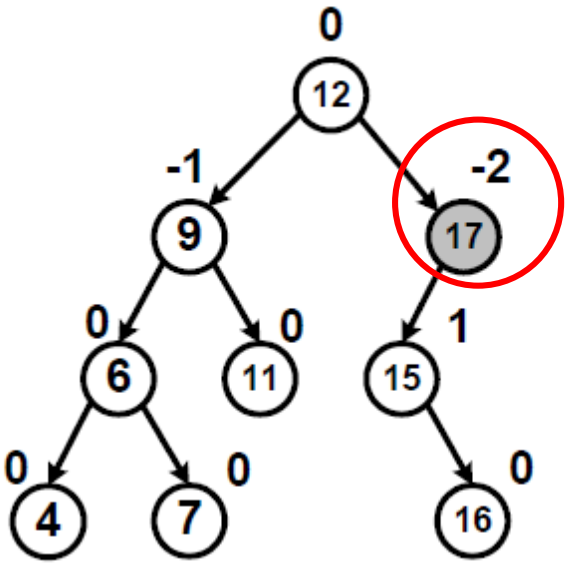
As long as the tree maintains this property, if the tree contains **n** nodes, then it has a depth of at most $\mathbf{log_2}\boldsymbol{n}$.

As a result, search for any node will cost $\mathbf{log_2 n}$, and if the updates can be done in time proportional to the depth of the node inserted or deleted, then updates will also cost $\mathbf{log_2 n}$, even in the worst case.

AVL tree



AVL tree

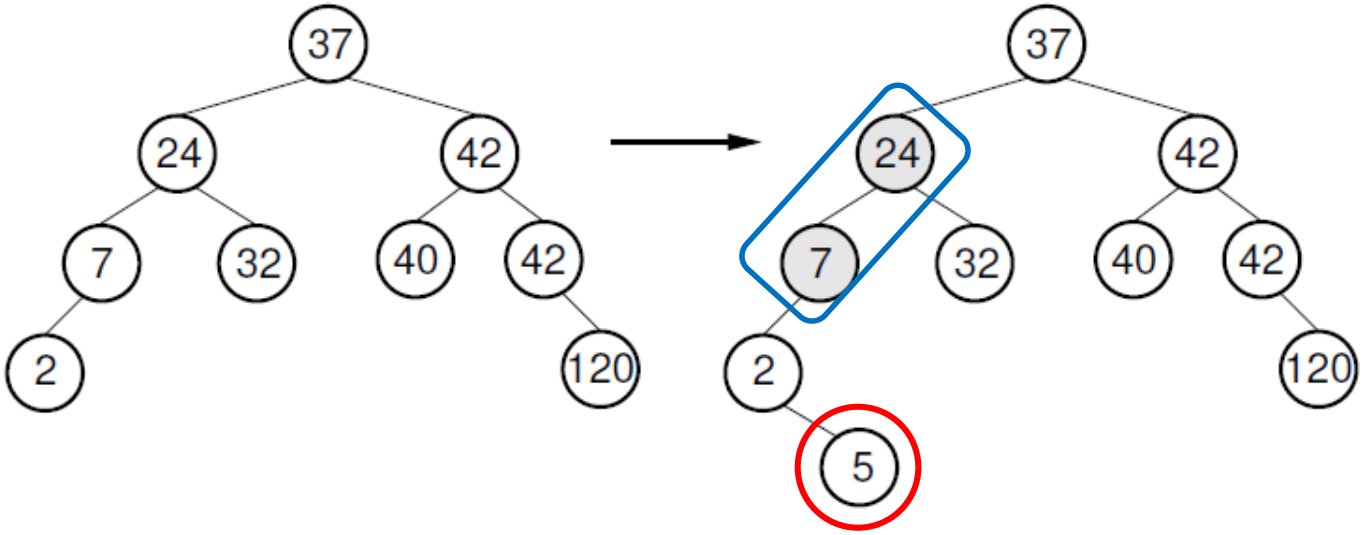


Not AVL tree

Realization of AVL tree element

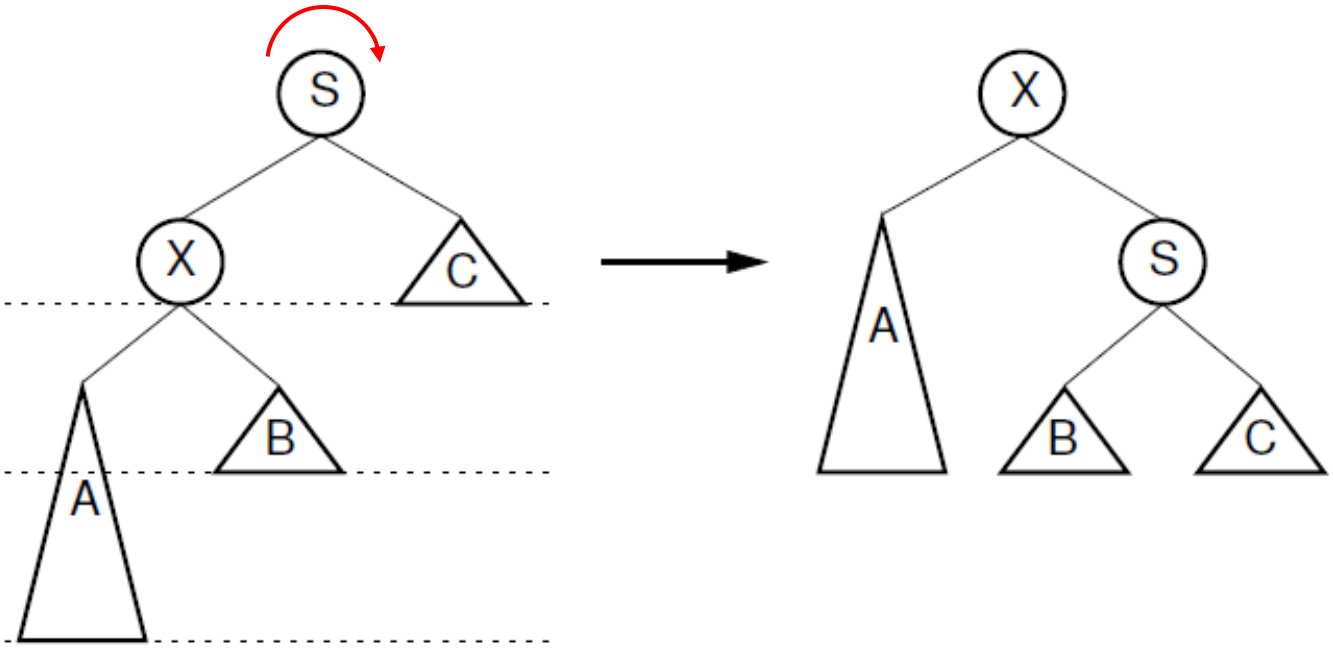
```
struct  AVLnode
{
    int data;
    AVLnode* left;
    AVLnode* right;
    int factor;              // balance factor
}
```

Adding a new node



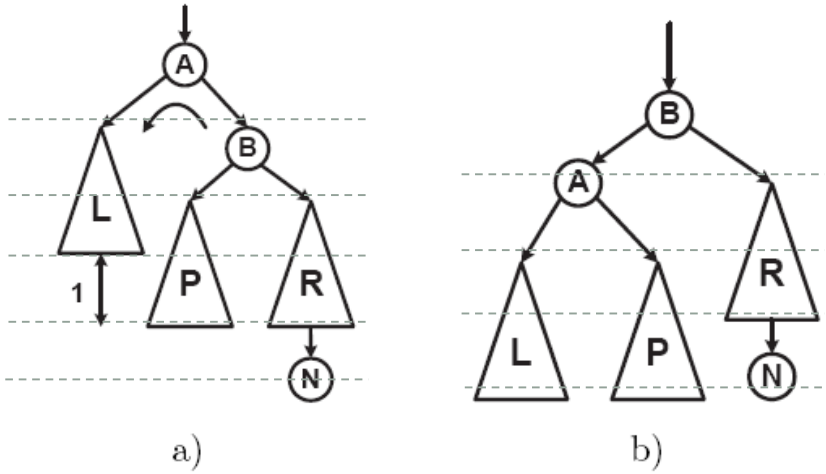
Insert operation violates the AVL tree balance property. Prior to the insert operation, all nodes of the tree are balanced (i.e., the depths of the left and right subtrees for every node differ by at most one). After inserting the node with value **5**, the nodes with values **7** and **24** are no longer balanced.

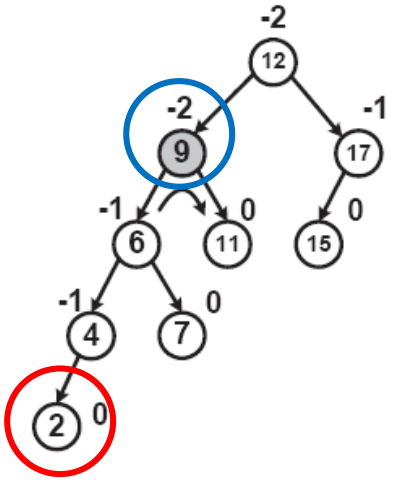
Single rotation (right)



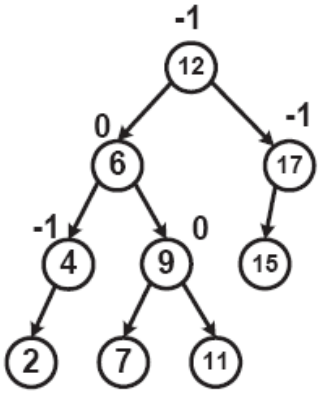
A single rotation in an AVL tree. This operation occurs when the excess node (in subtree A) is in the left child of the left child of the unbalanced node labeled **S**. The case where the excess node is in the right child of the right child of the unbalanced node is handled in the same way.

Single rotation (left)



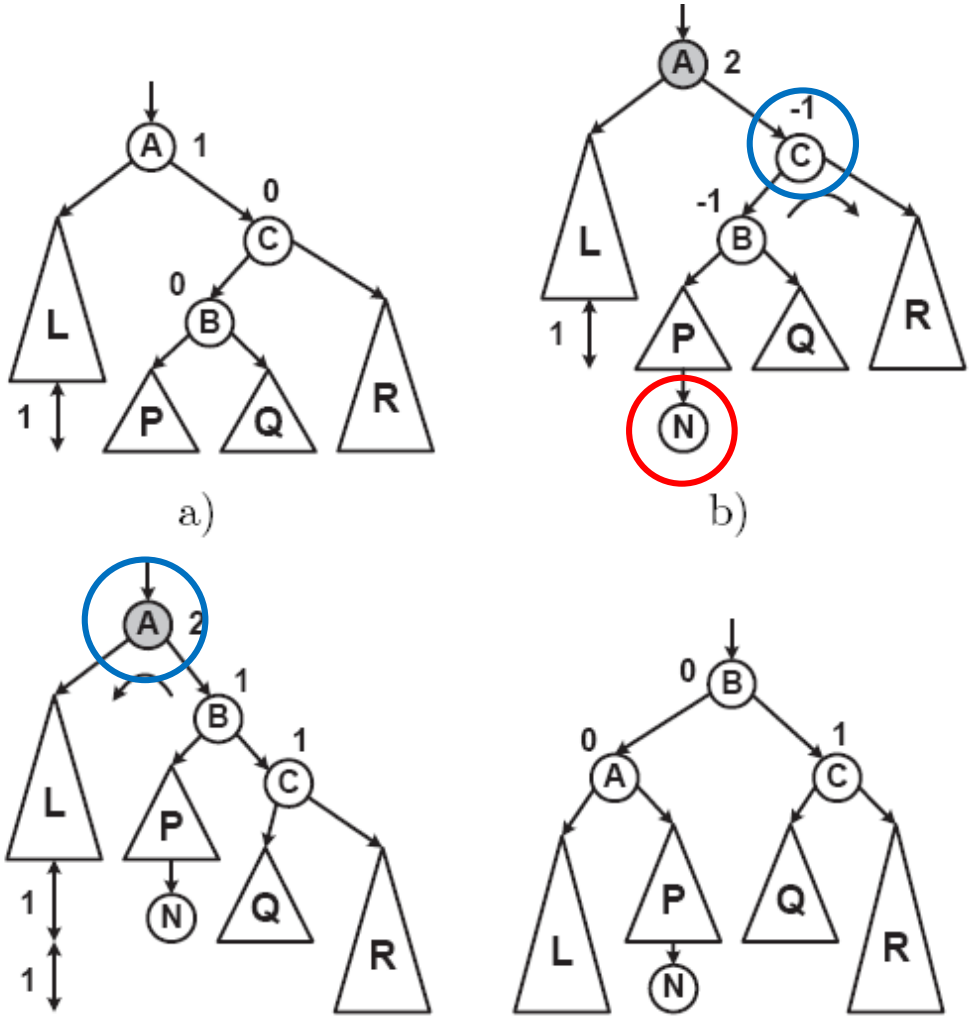


AVL tree after insertion new node 2

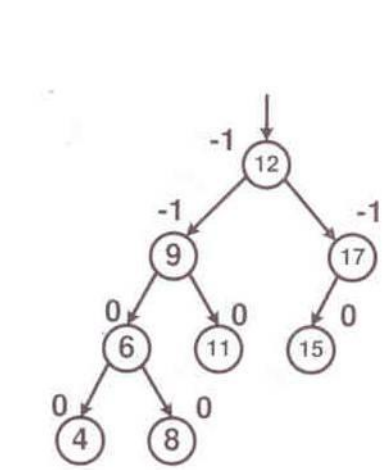


AVL tree after rotation

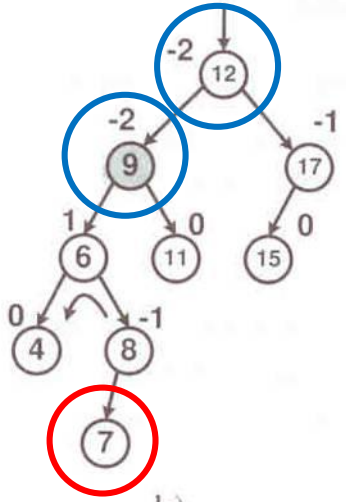
Two single rotations



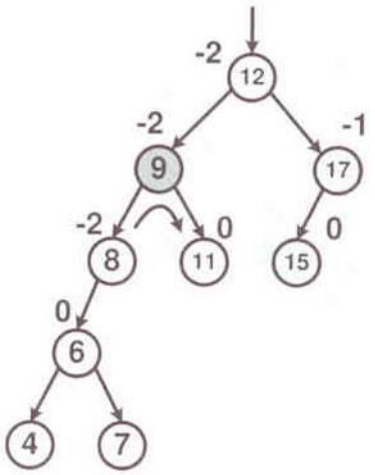
Two single rotations (example)

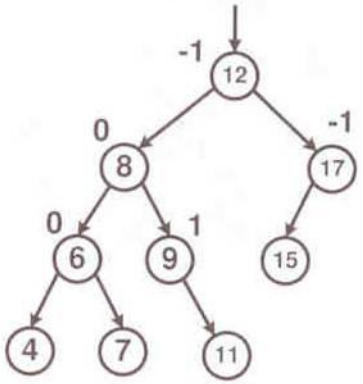


a)

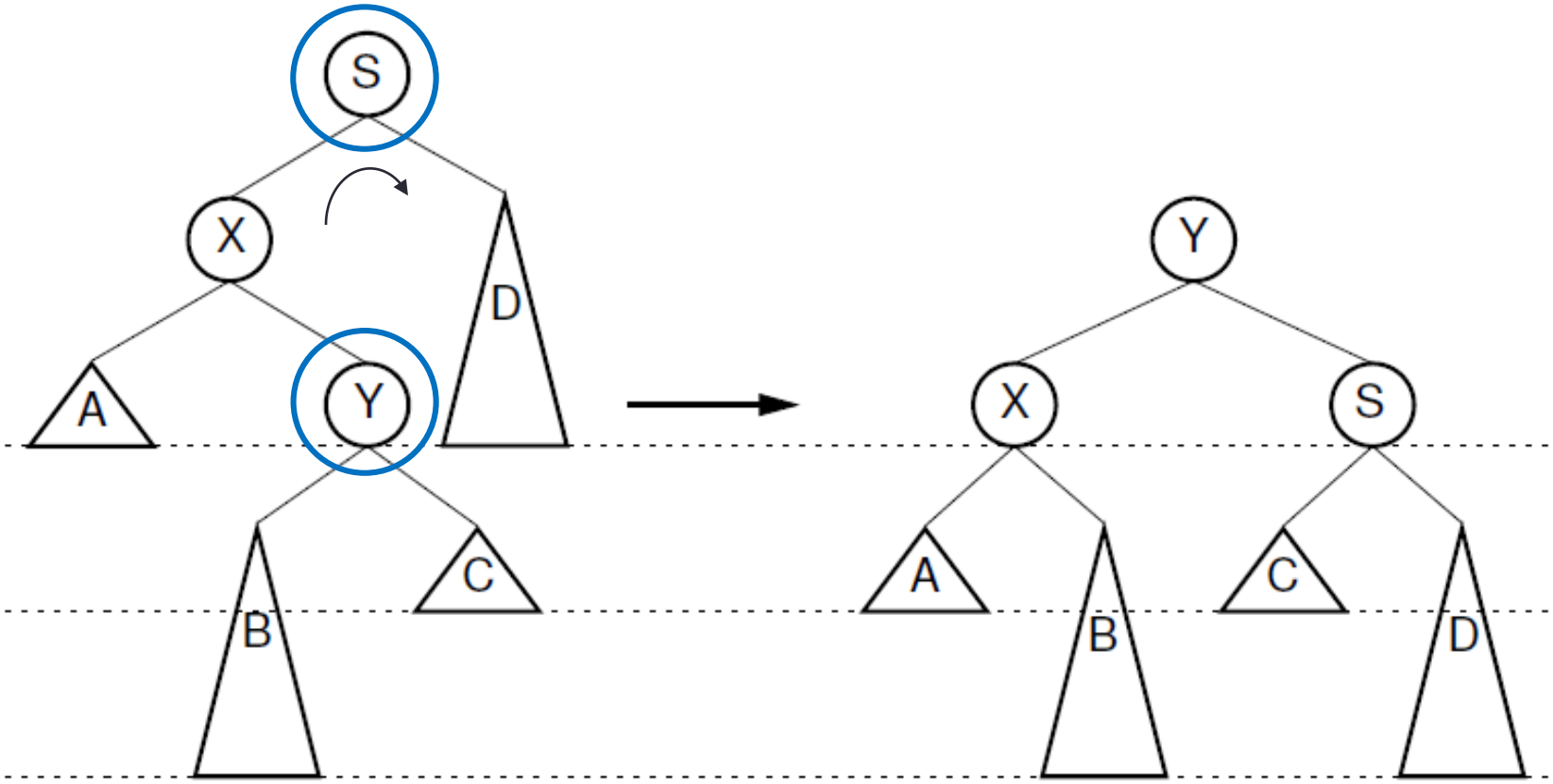


b)

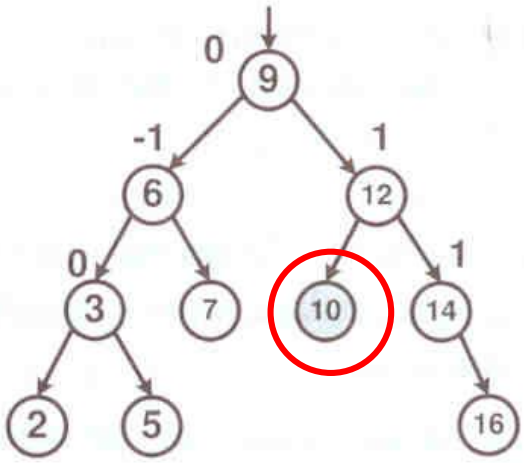




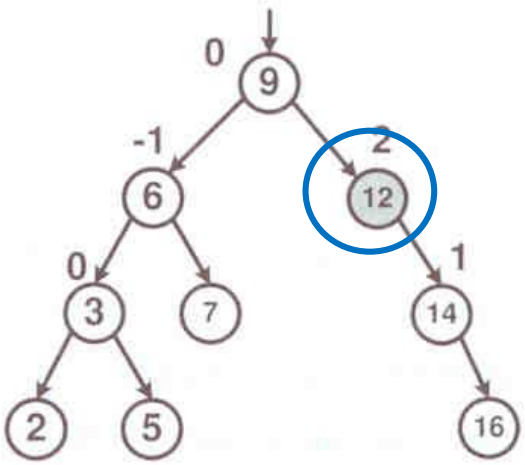
Double rotation in AVL tree



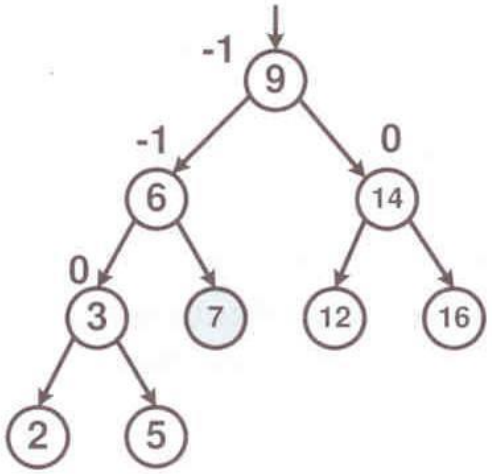
Removing single node



Initial AVL tree



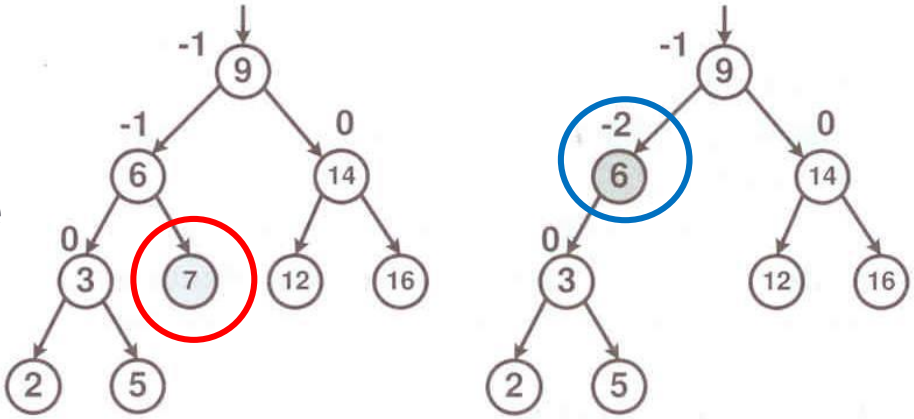
Removed leaf 10



Balanced AVL tree

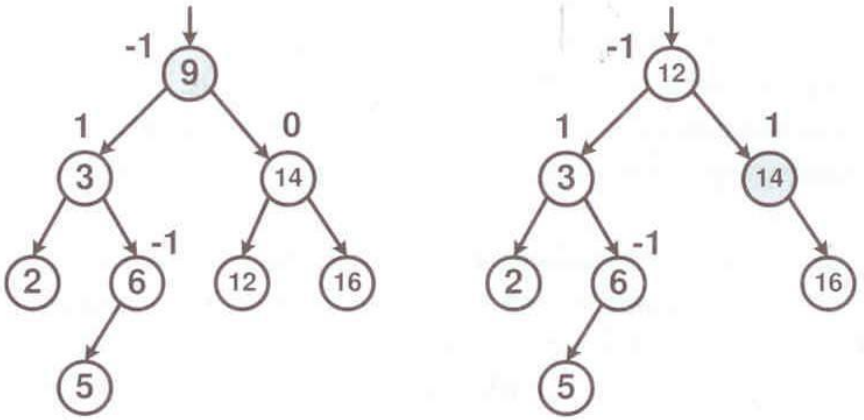
Removing single node

Balanced AVL tree



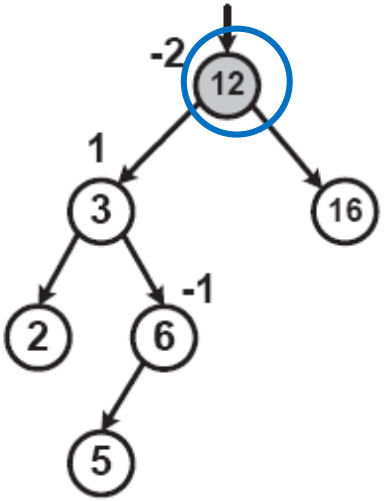
Removed node 7

Single rotation

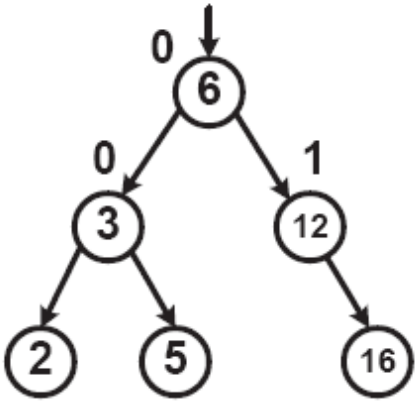


Removed node 9

Removing single node



Removed node 14



Balanced AVL tree

Bayer tree

Bayer tree (B-tree) is a tree data structure that keeps **data sorted** and allows searches, sequential access, insertions, and deletions in logarithmic amortized time.

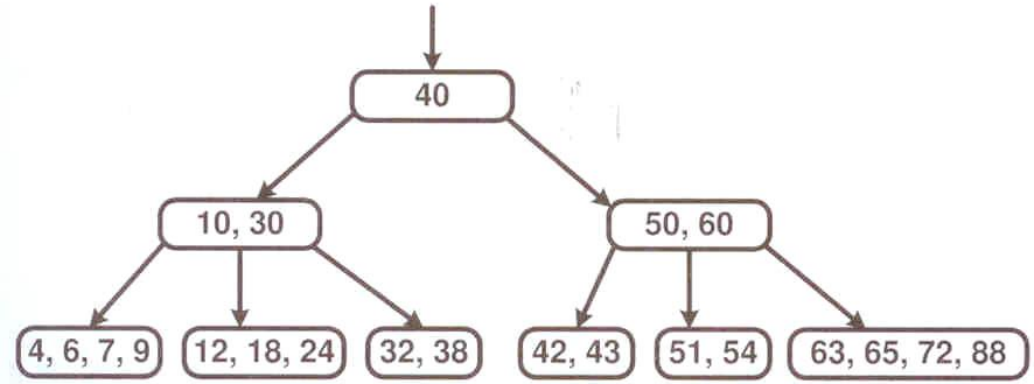
The B-tree is a generalization of a **binary search tree** in that a node can have more than two children.

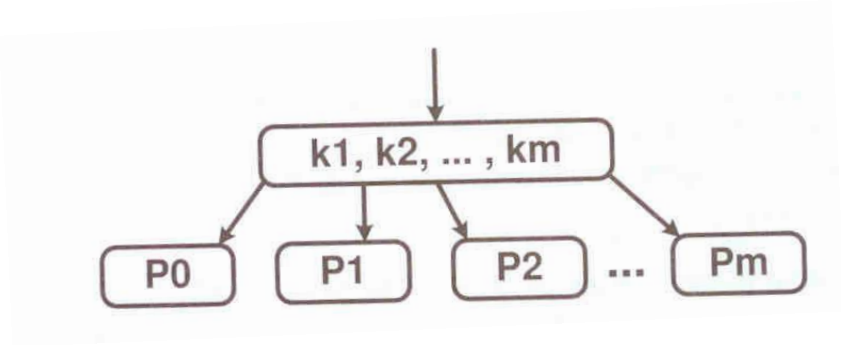
All leaves of the B-tree are at the same level of the tree.

B-tree is always balanced tree.

B-tree is optimized for systems that read and write large blocks of data. It is commonly used in databases and file systems.

B-tree example



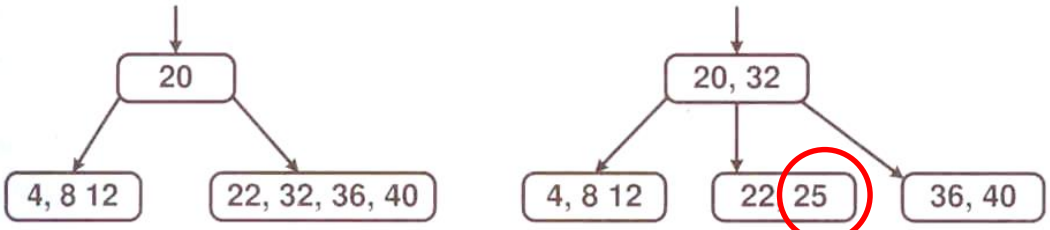


Property of B-tree:

P0 < k1 < P1 < k2 < P2 < … < Pm-1 < km < Pm, *where m <=2N*

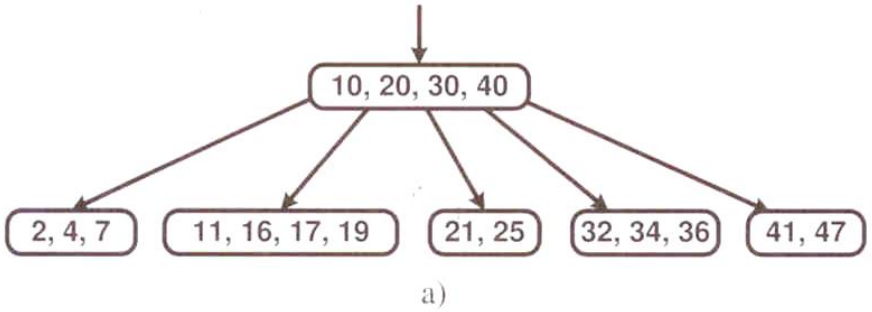
Adding a new value into B-tree

Initial B tree

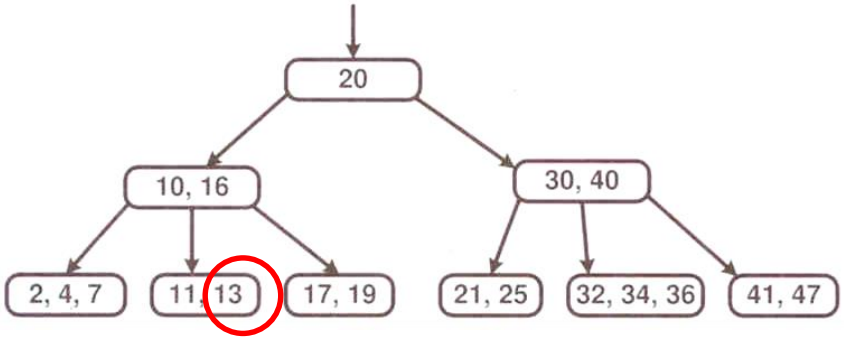


Modified B tree

Initial B tree



Modified B tree



Adding a new value into B-tree

The following algorithm is used to add new value:

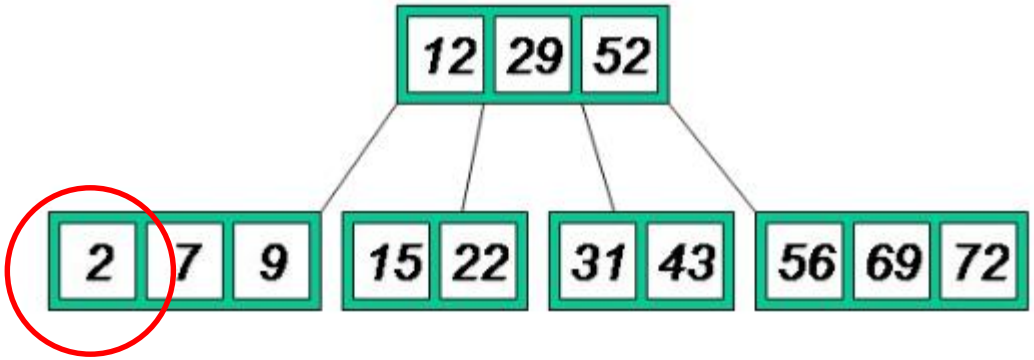
- Try to put new value into leaf node.
- If leaf is full (too many key values), then leaf is divided into two parts and middle key is moved to parent node.
- If parent is full, then it is divided into two parts and middle key is moved to parent node at the higher level.
- This procedure is repeated till root node.
- If root is full, then it is divided into two parts and middle key becoming a new root.

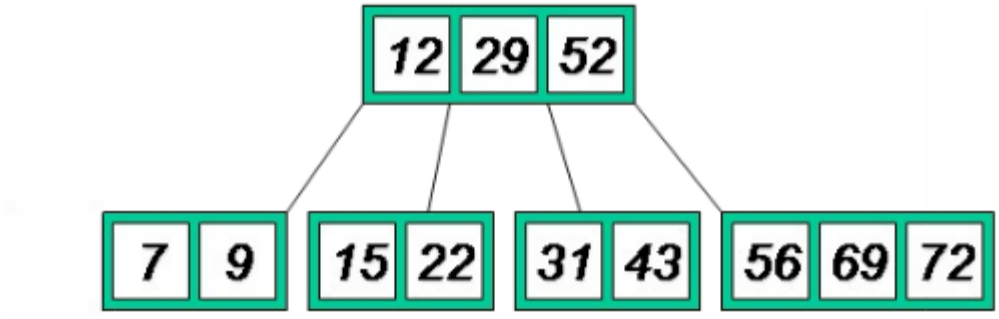
Deleting a key value

The following algorithm is used to delete a key value from the B-tree:

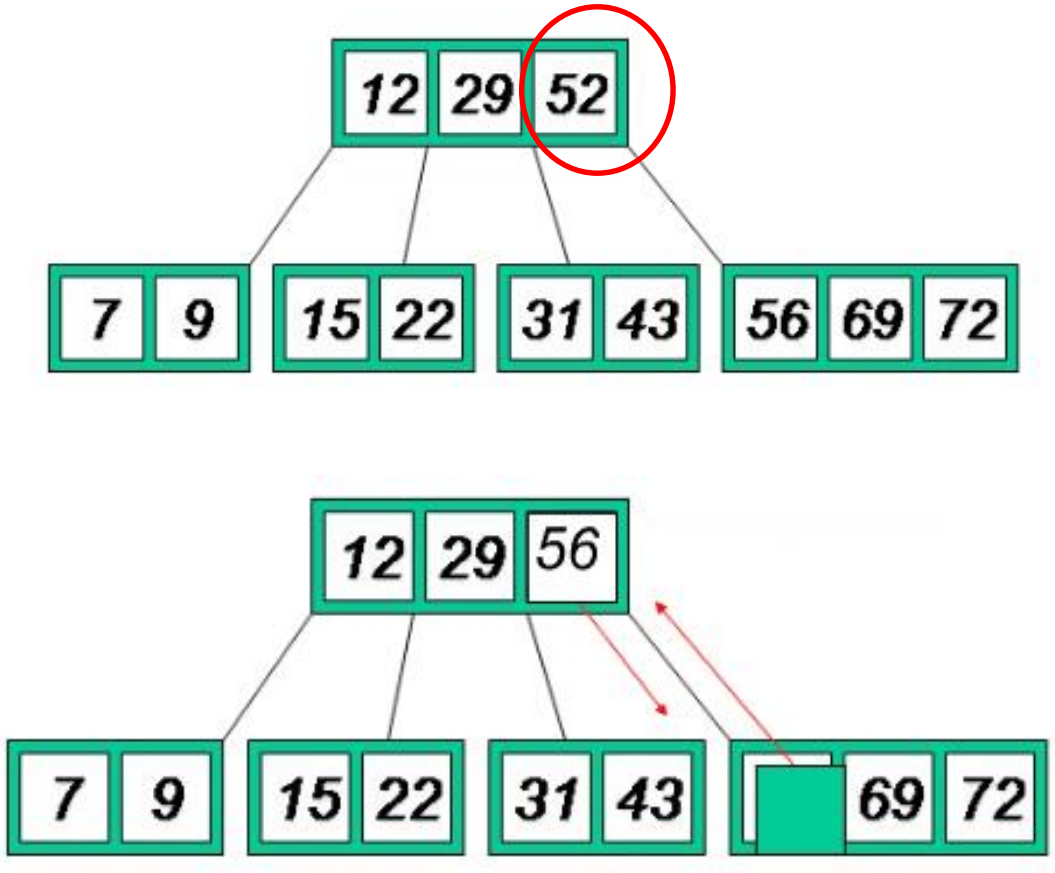
- If key value is deleted from leaf node and number of the keys is sufficient, then no more steps are made.
- If key value is deleted from internal node, then key value from the child node is move into the place of deleted key value.

Deleting a key value from the leaf

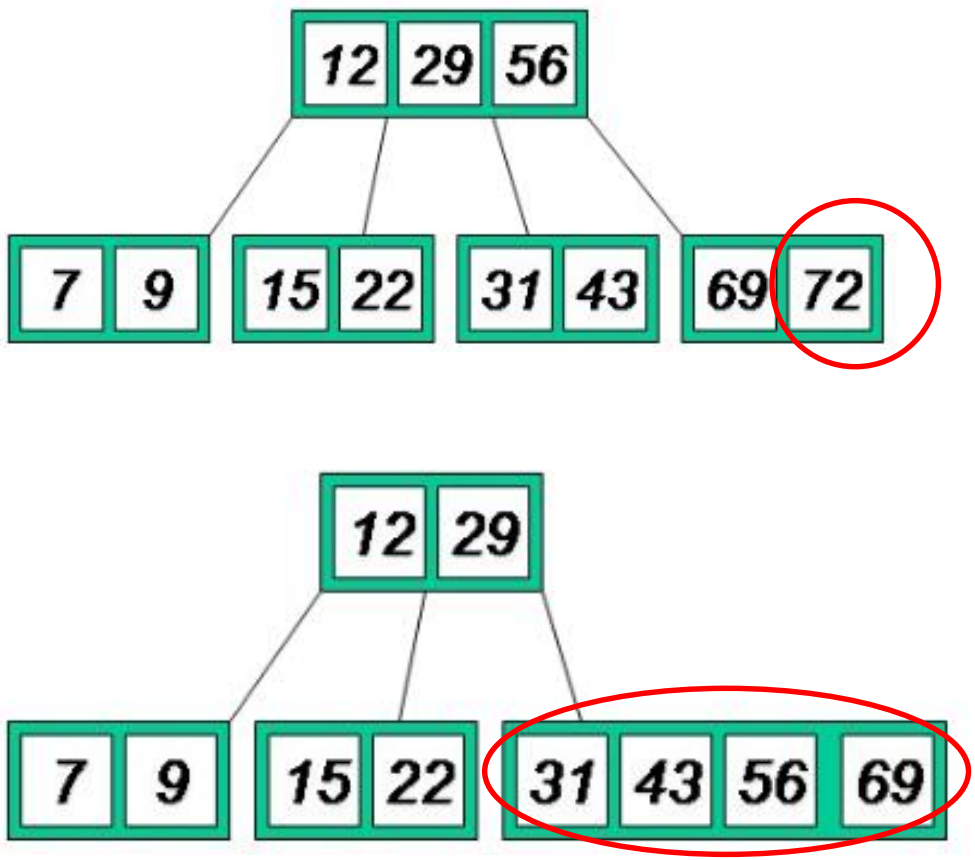




Deleting a key value from node



Deleting a key value from the leaf



Binary heap

A **binary heap** is a data structure created using a binary tree.

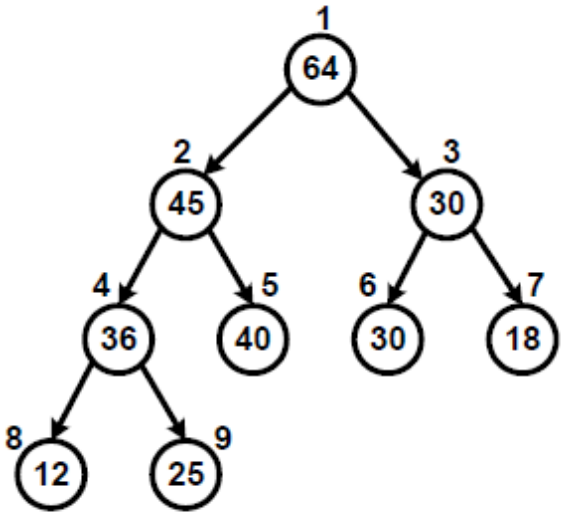
It can be seen as a binary tree with two additional constraints:

- The ***shape property***: the tree is an *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.

-The ***heap property***: each node is greater than or equal to each of its children according to some comparison predicate which is fixed for the entire data structure.

Largest node is ROOT.

Example



a)

1	2	3	4	5	6	7	8	9
64	45	30	36	40	30	18	12	25

Binary heap

Heap is full binary tree, so it convenient to store heap in a single array. Elements of array are heap nodes.

If heap node with index i is a parent node, then children indexes are $2i$ and $2i+1$.

Sequence of the elements e_1 ... e_n can be used for building heap and storing in the single array. Property of such array: indexes of leafs are from (N/2 + 1) till N.

Building a heap

Assume that we have the following set of elements:

$e_1\ e_2 \ldots e_N$.

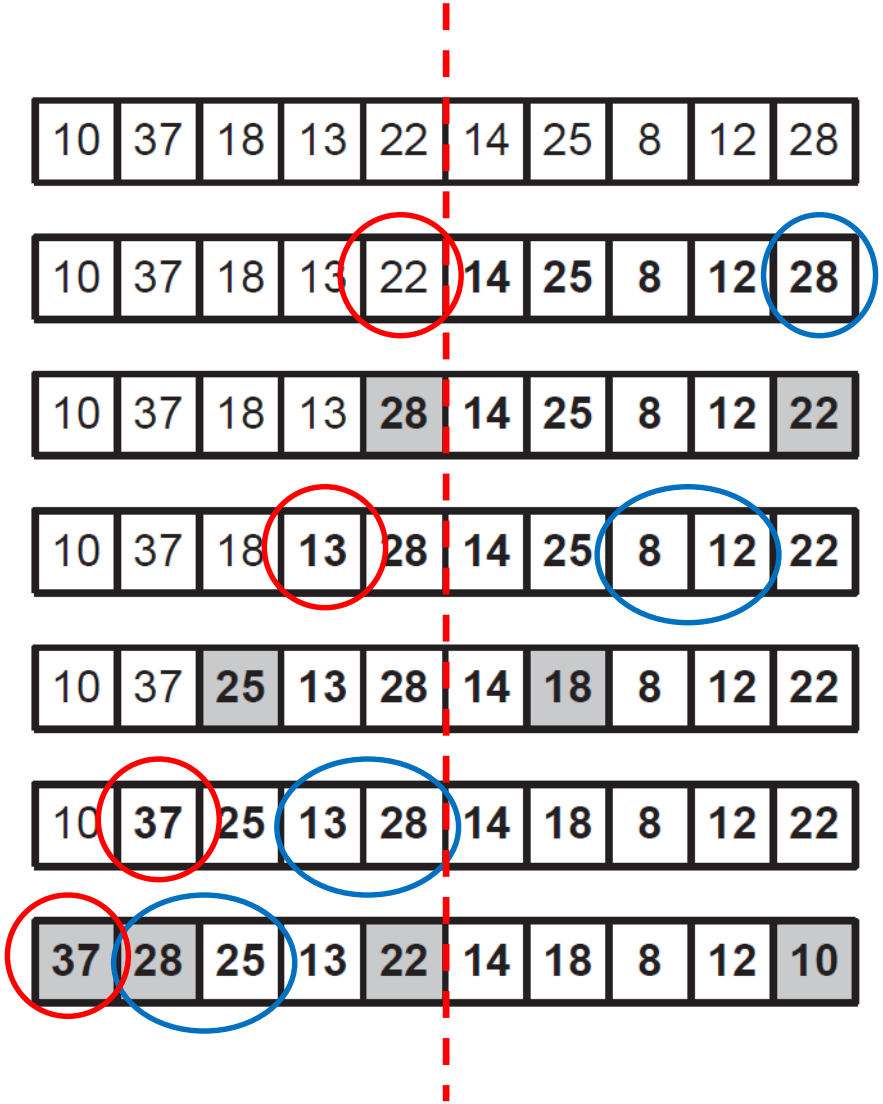
Algorithm for building a heap:

1. Put given elements into array in the same sequence as given in the initial set.
2. Make a loop for the nodes with index number from N/2 till 1 and check if the heap property is fulfilled. If heap property is not fulfilled, then change parent node this the largest child.

Running time of the heap building can be evaluated as follows:

$$L(N) = N \log N, \quad S(N) = \frac{1}{2} N \log N$$

Example


10 37 18 13 22 14 25 8 12 28
10 37 18 13 22 14 25 8 12 28
10 37 18 13 28 14 25 8 12 22
10 37 18 13 28 14 25 8 12 22
10 37 25 13 28 14 18 8 12 22
10 37 25 13 28 14 18 8 12 22
37 28 25 13 22 14 18 8 12 10

Heap algorithm

```
MakeHeap ()
begin
  (1)  for ( i=1; i ⩽ N; i++ ) do
  (2)       a_i = e_i;
       end do
  (3)  j = N/2;
  (4)  for ( i=j; i > 0; i −= 1 ) do
  (5)       HeapDownOrder ( i, N );
       end do
end MakeHeap
```

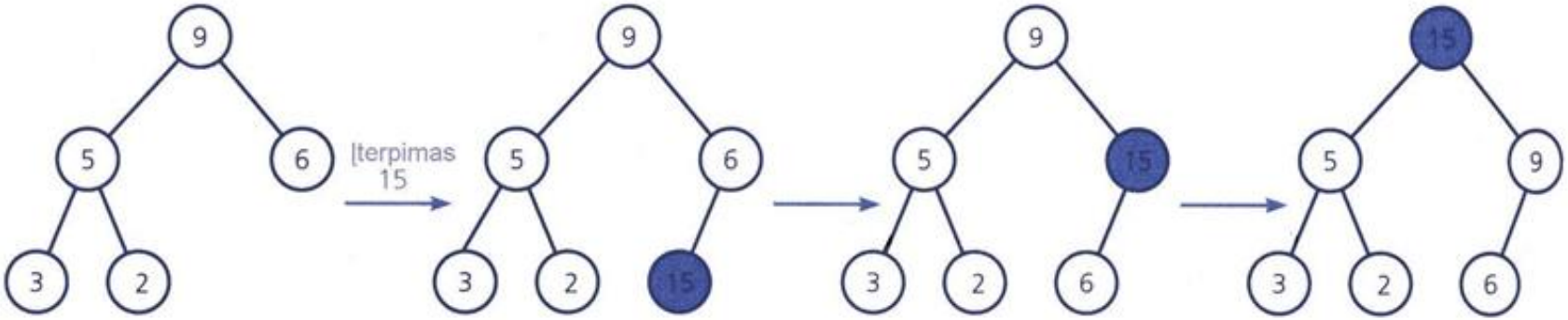
Heap algorithm

```
HeapDownOrder ( p, N )
begin
  (1)   i=p;    j = i+i;
  (2)   while   ( j ⩽ N ) do
  (3)         k = j;
  (4)         if  ( (j+1) ⩽ N ) then
  (5)           if  ( a_{j+1} > a_j ) k = j+1;
                end if
  (6)         if  ( a_i < a_k ) then
  (7)           swap (a_i, a_k);
  (8)           i = k;    j= i+i;
  (9)         else
  (10)          j = N+1;
                end if
        end do
end HeapDownOrder
```

Adding a new node

Adding a new element must be done as follows:

- new node must be added as a leaf and heap must be rebuilt.



Deleting a root

Heap must be rebuilt after deleting a root as follows:

- Last leaf is becoming a root.
- Heap must be rebuilt to correspond requirement of the heap.