Balanced Search Tree

AVL *Tree*

Outline

Data Structures

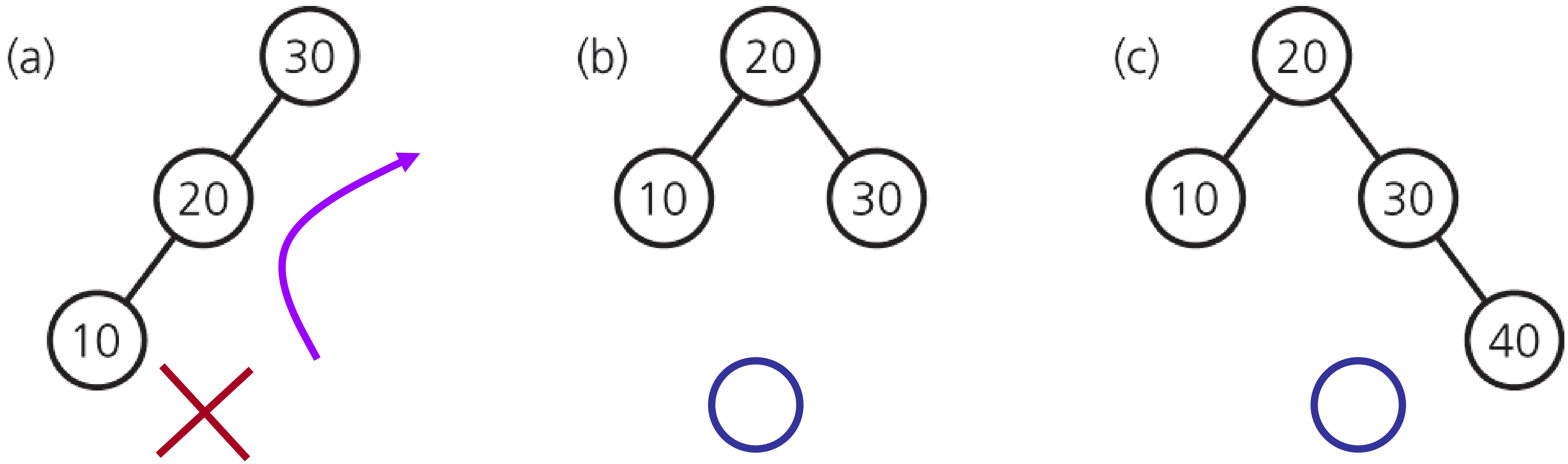
- 2-3 tree
- 2-3-4 tree
- AVL tree
  - [Adelson-Velskii & Landis, 1962]
- Red-black tree
  - [Rudolf Bayer, 1972]… B-tree

AVL Tree

- **An AVL tree**
  - A balanced binary search tree
  - Can be searched almost as efficiently as a minimum-height binary search tree
  - Maintains the tree height close to the minimum
    - **Requires far less work than would be necessary to keep the height exactly equal to the minimum**

AVL Tree: *Main Idea*

- **After each insertion or deletion**
  1. Check whether the tree is still balanced
  2. If the tree is unbalanced, rotate to restore the balance

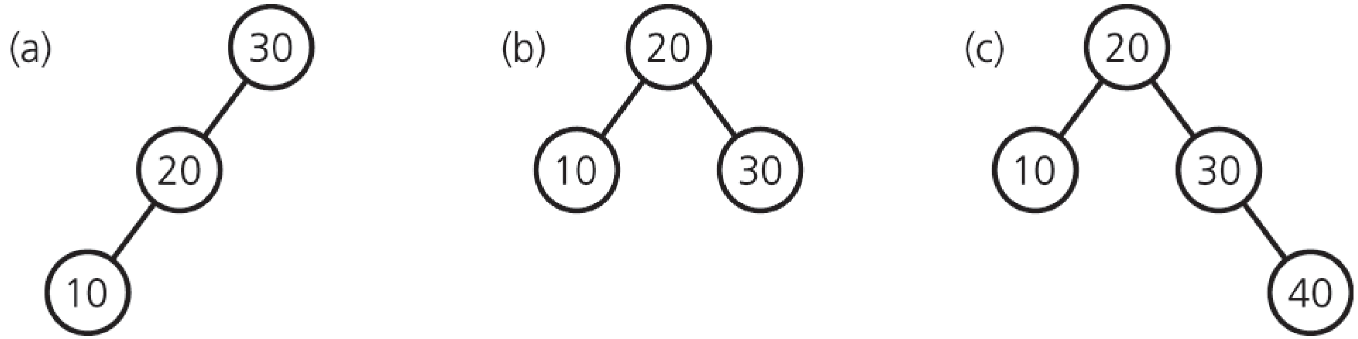


AVL Tree: *Main Idea*

- **After each insertion or deletion**
  1. Check whether the tree is still balanced
- **Balance Factor (BF)**

  BF(a node) = h(left subtree) – h(right subtree)

  – The heights of the left and right subtrees of any node in a binary search tree differ by no more than **1**.

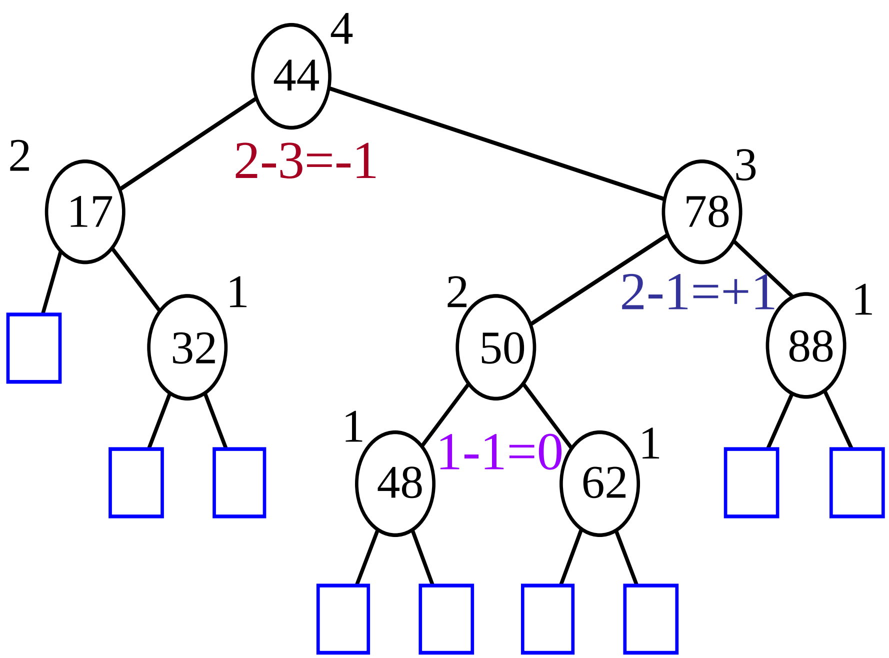


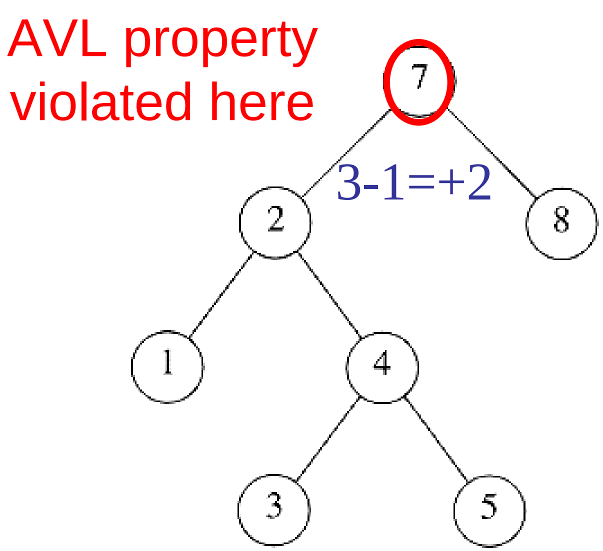
AVL Tree: *Property*

1. Check whether the tree is still balanced

BF(a node) = h(left subtree) – h(right subtree)

– The heights of the left and right subtrees of any node in a binary search tree differ by at most 1.





Rotations

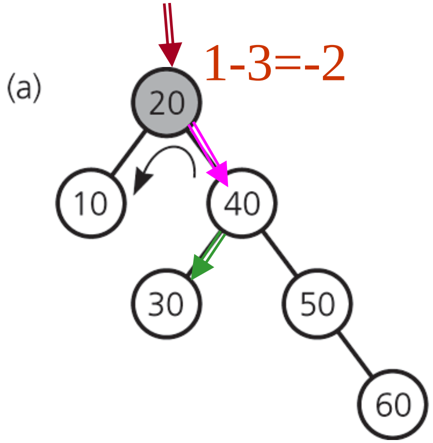
- Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of some subtree by 1
- Thus, if the AVL tree property is violated at a node *x*, it means that the heights of *left(x)* ad *right(x)* differ by exactly 2.
- Rotations will be applied to *x* to *restore* the AVL tree property.

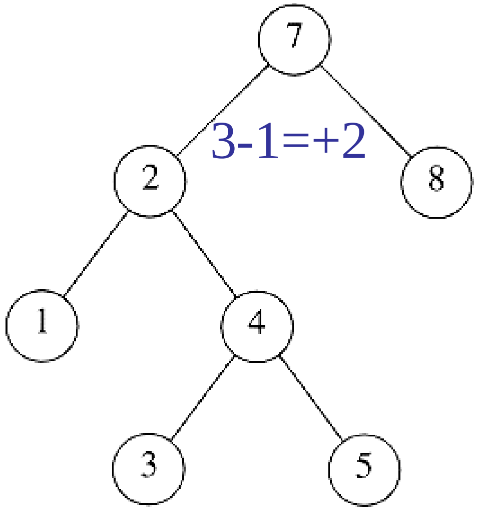
AVL Tree: *Actions*

Data Structures

- **After each insertion or deletion**
  1. Check whether the tree is still balanced
  2. If the tree is unbalanced, rotate to restore the balance

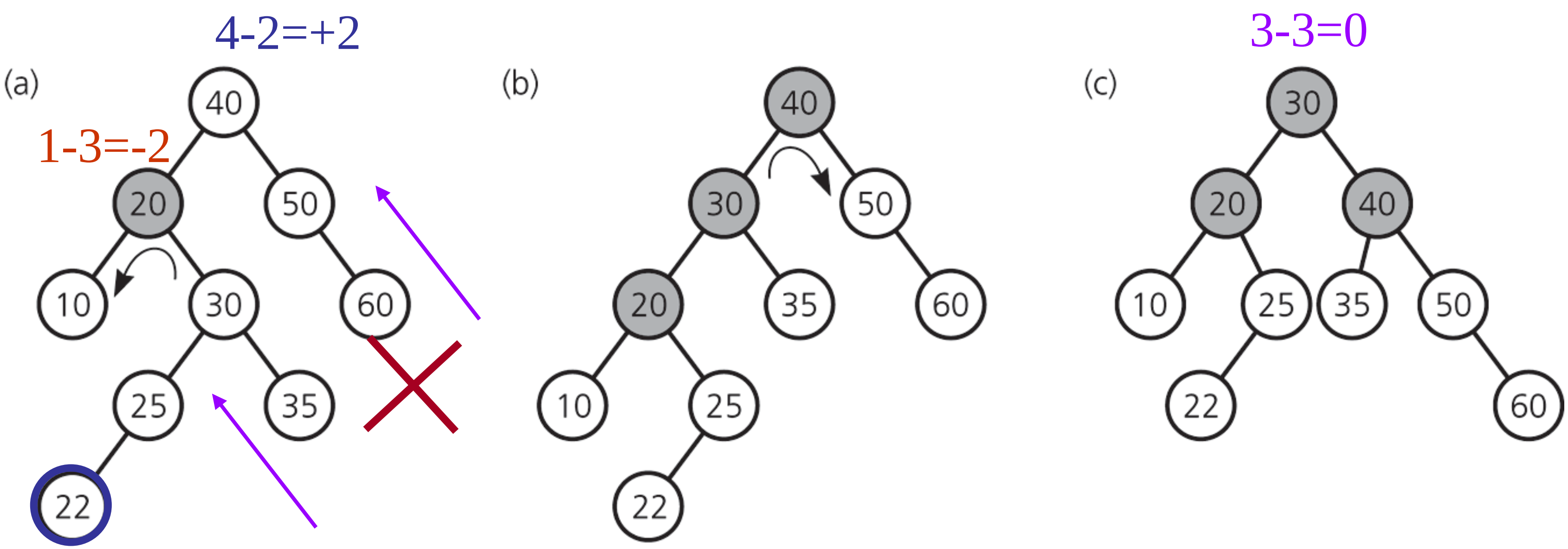
Rotations to restore the *balance* property
- Single rotation
- Double rotation





AVL Tree: *Actions*

1. If the tree is unbalanced, rotate to restore the balance
   - Single rotation
   - Double rotation



Which node to be rotated first?

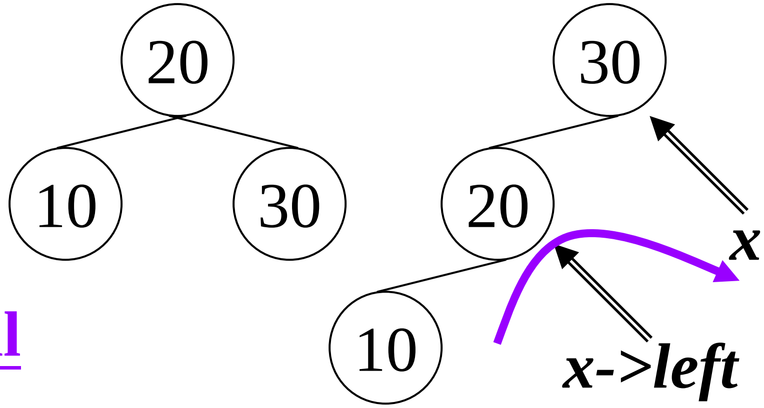
AVL Tree: *Insertion*

Data Structures

1. **Insert the new key as a new leaf just as in a *binary search tree***
2. **Trace the path *from the new leaf towards the root*.**
   - For each node *x* encountered, check if the heights of *left(x)* and *right(x)* differ by at most 1.
     - **If NOT, restructure by either a single rotation or a double rotation**
3. **Once we perform a rotation at a node *x*, the insertion is done!**
   - We won't need to perform any rotation at any ancestor of *x*.

AVL Tree: *Single Rotations*

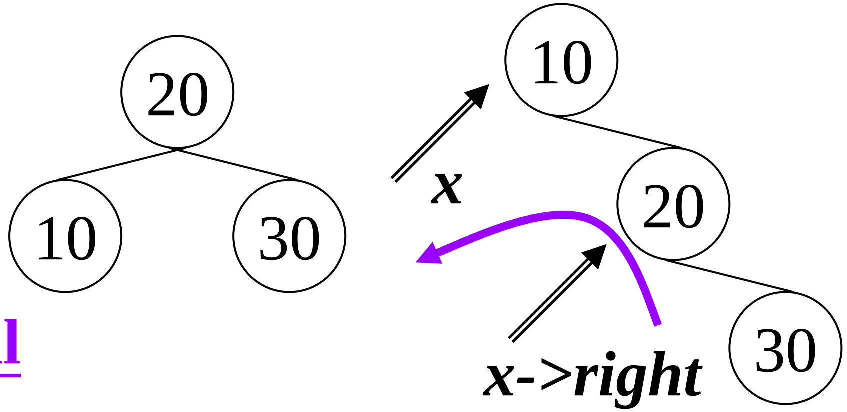
- **Let *x* be the node at which *x->left* and *x->right* differ by more than 1; Assume that the height of *x* is 3**
  - Height of *x->left* is **2** (i.e. height of *x->right* is 0)
    1. **Height of *x->left->left*: 1 ⇒ single rotation with the left child (LL)**
       - **BF(x) = +2**
       - **BF(x->left) = +1 or 0**



Tree height: shorter or equal

AVL Tree: *Single Rotations*

- Let *x* be the node at which *x->left* and *x->right* differ **by more than 1**; Assume that the height of *x* is 3
  - Height of *x->right* is **2** (i.e. height of *x->left* is 0)
    2. **Height of *x->right->right*: 1 ⇒ single rotation with the right child (RR)**
       - **BF(x) = -2**
       - **BF(x->right) = -1 or 0**



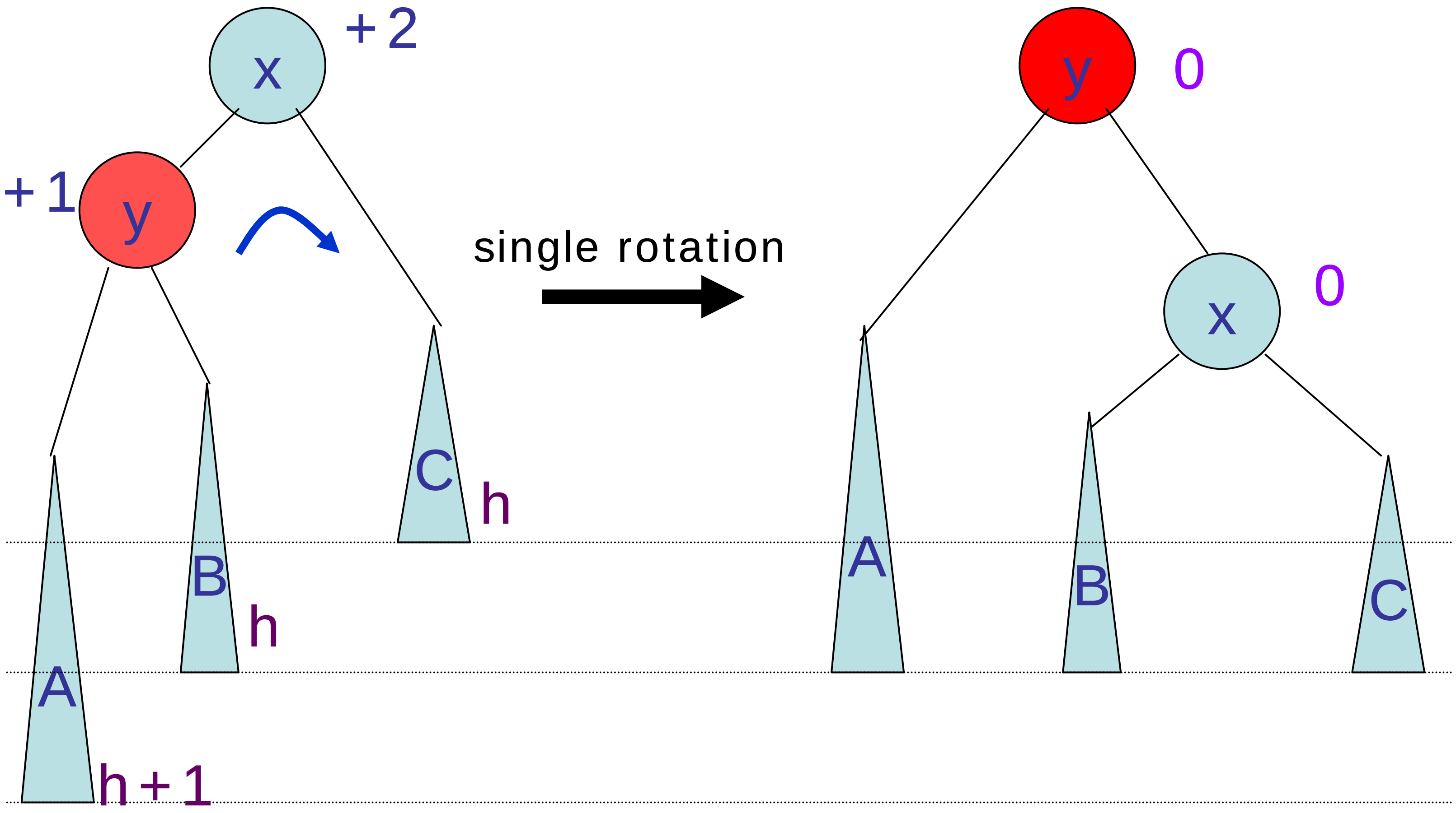
Tree height: shorter or equal

AVL Tree: *Single Rotations*

- Let *x* be the node at which *x->left* and *x->right* differ by more than 1; Assume that the height of *x* is h+3
  - Heights of two subtrees: **h+2**, h
    1. Height of *x->left->left*: h+1, *x->left->right*: h or h+1 ⇒ single rotation with the left child (LL)
       - BF(x) = +2 BF(x->left) = +1 or 0
    2. Height of *x->right->right*: h+1, *x->right->left*: h or h+1 ⇒ single rotation with the right child (RR)
       - BF(x) = -2 BF(x->right) = -1 or 0

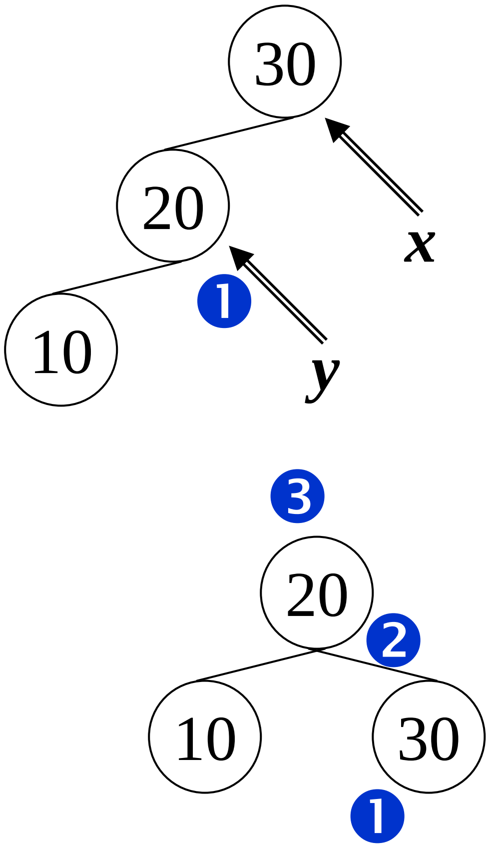
Q&A: What are the conditions for single rotations?

Single Rotations: *LL*

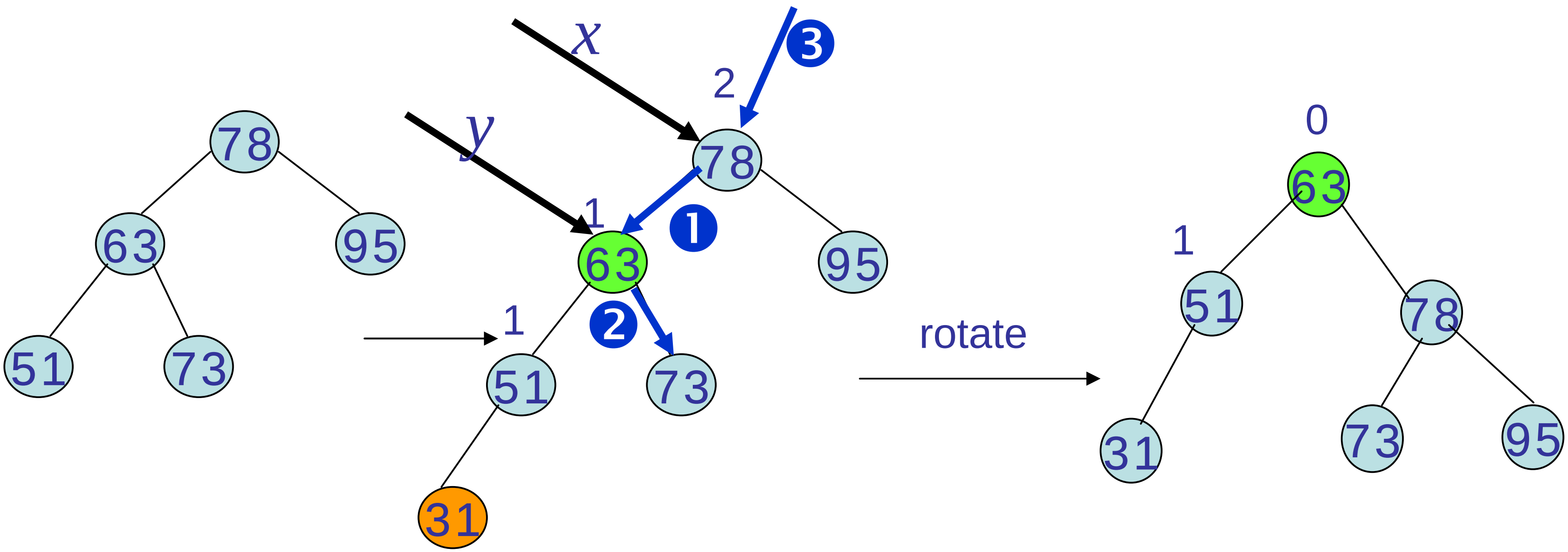


LL Rotation: *Pseudocode*

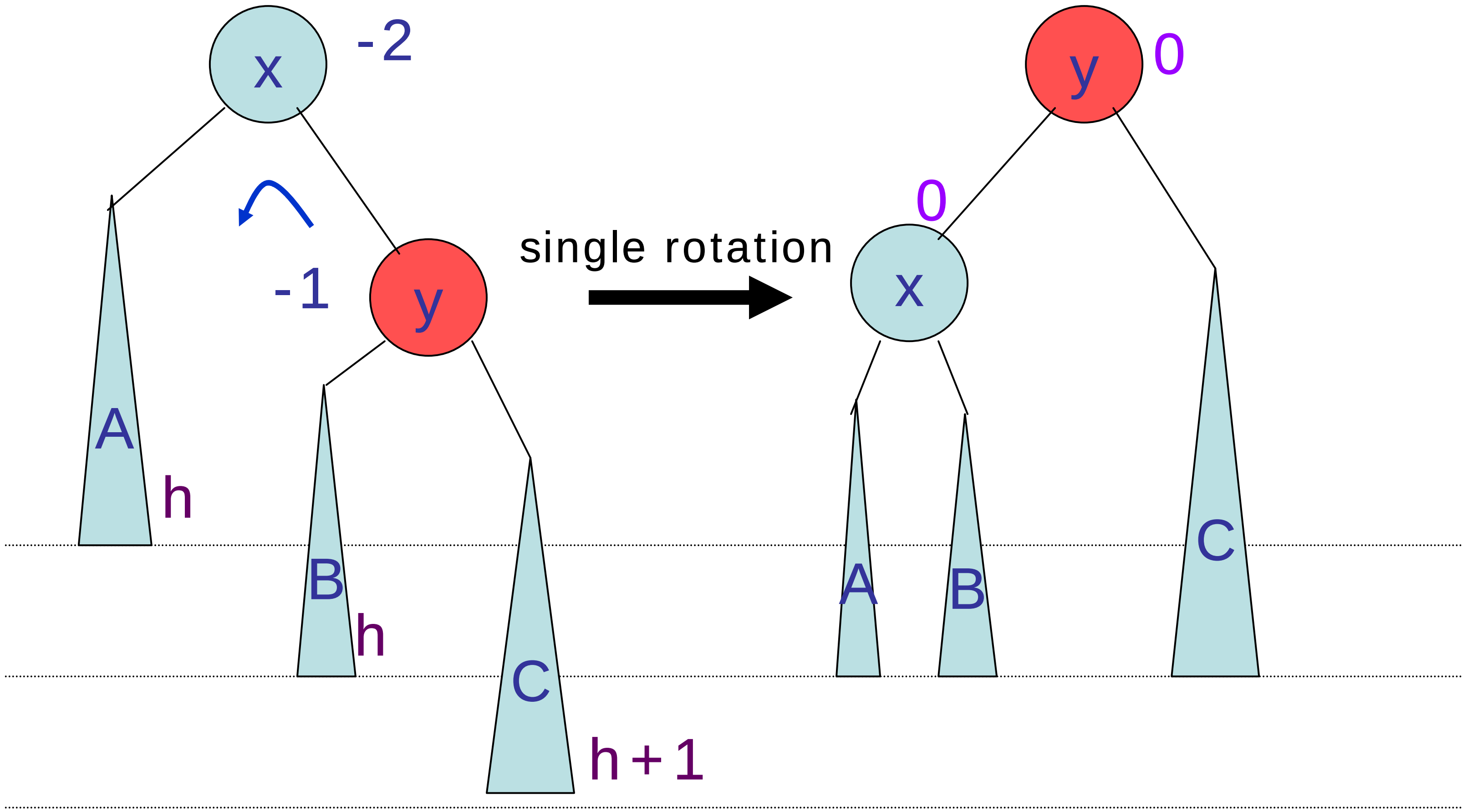

```
// rotate x with its left child
nodeType rotateLL(nodeType x)
{
   nodeType y = x->left;
   x->left = y->right;      ➊
   y->right = x;            ➋
   return y;                ➌
}
```



LL Rotation: *Example*



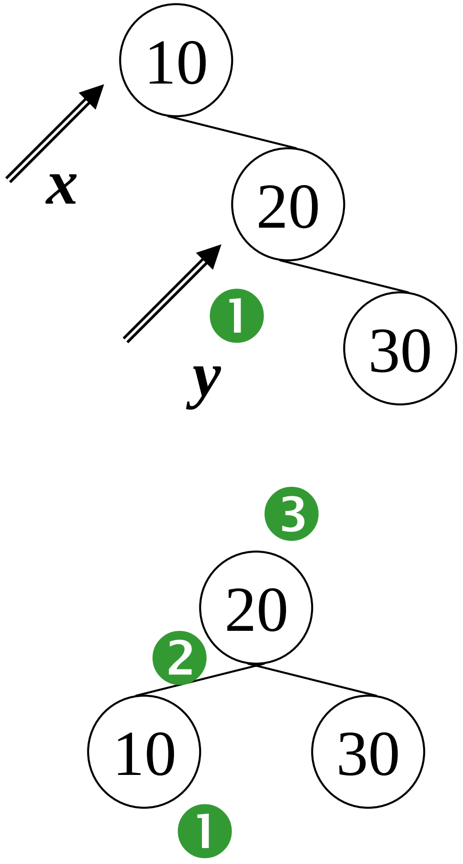
Single Rotations: *RR*



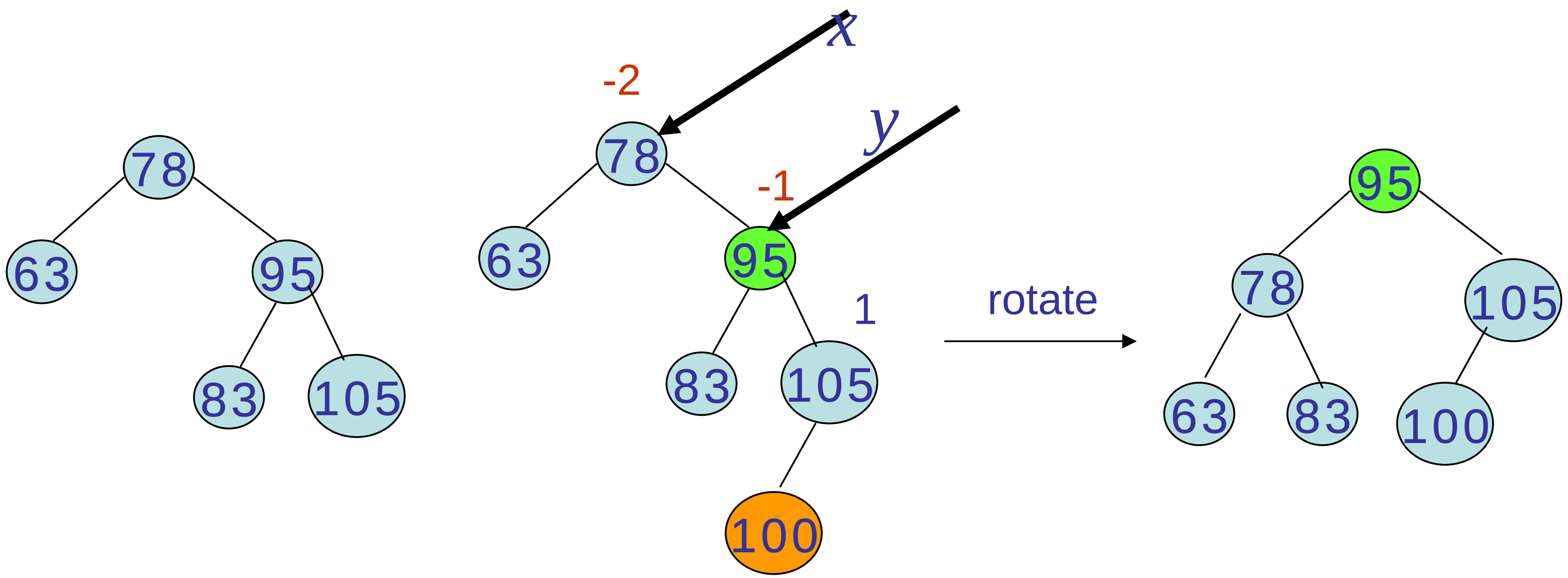
RR Rotation: *Pseudocode*

Data Structures

```
// rotate x with its right child
nodeType rotateRR(nodeType x)
{
    nodeType y = x->right;
    x->right = y->left;      ❶
    y->left = x;             ❷
    return y;                ❸
}
```



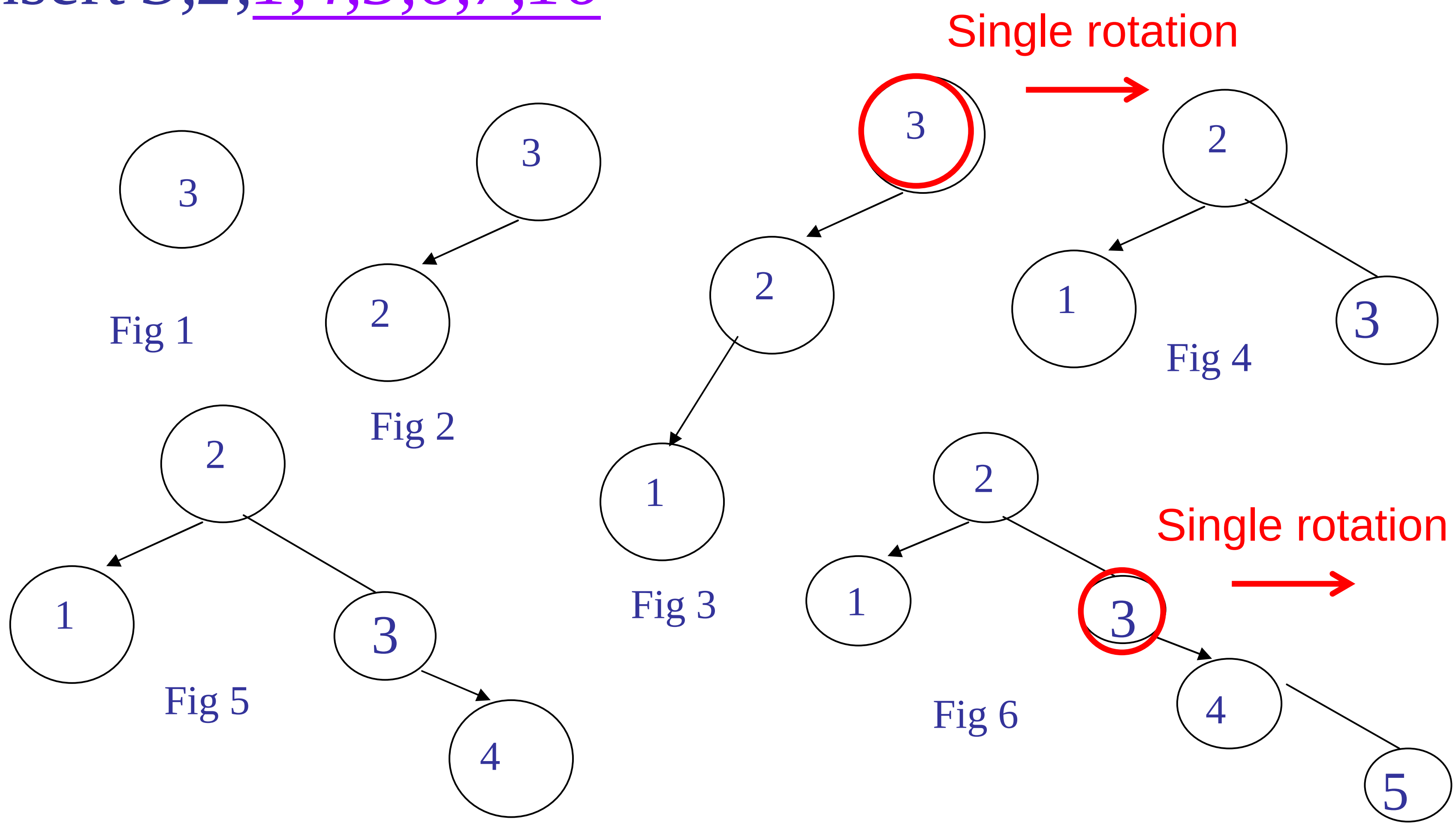
RR Rotation: *Example*



Try it!

Data Structures

Insert 3,2,*1,4,5,6,7,16*



Try it!

Insert *3,2,1,4,<u>5,6,7,16</u>*

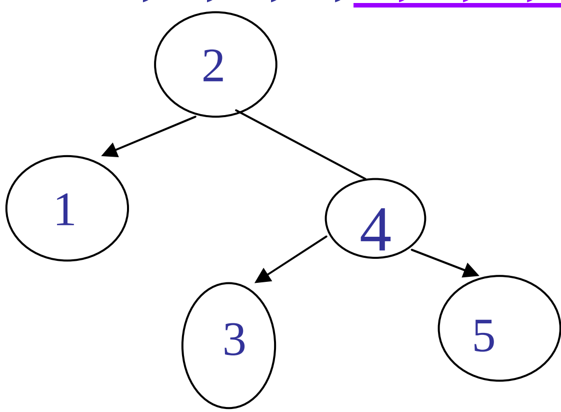


Fig 7

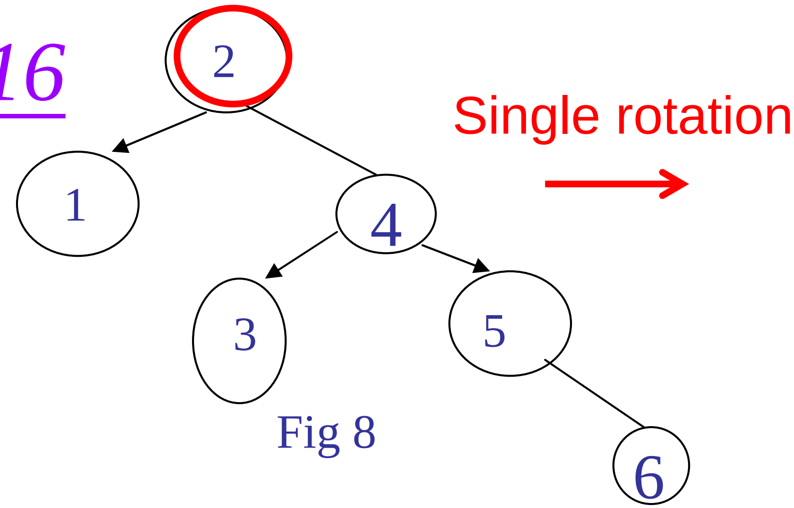


Fig 8

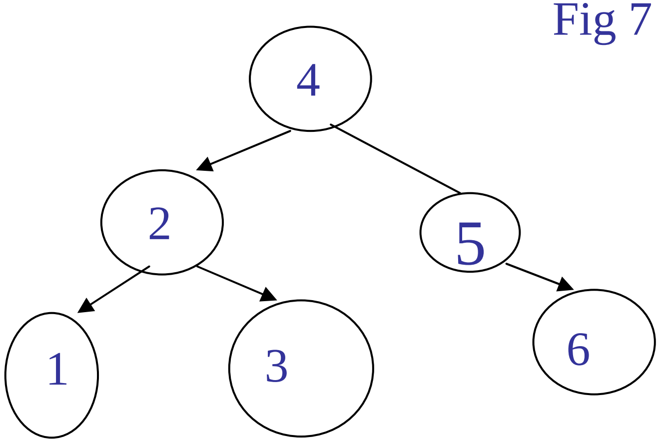


Fig 9

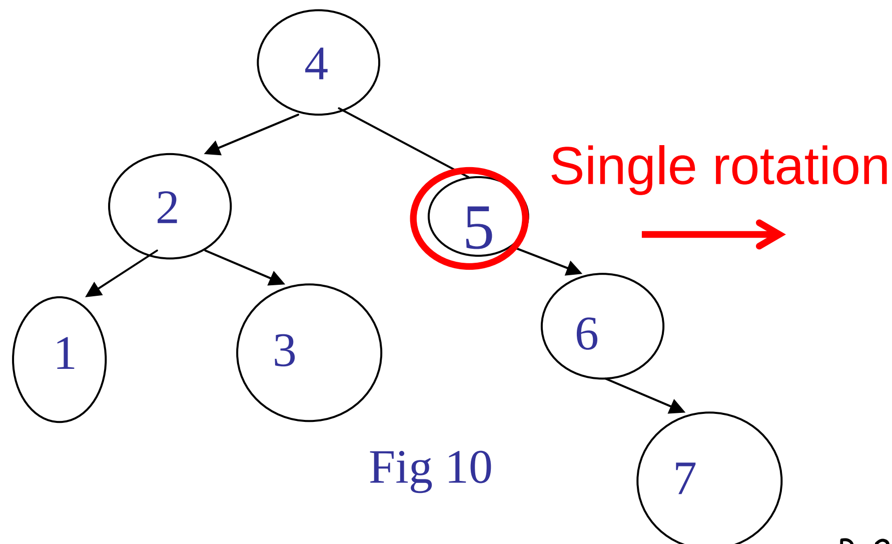


Fig 10

Try it!

Insert *3,2,1,4,5,6,7,16*

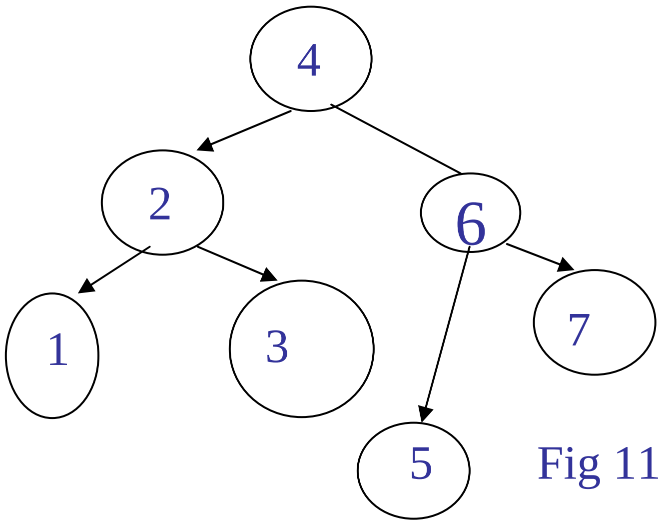


Fig 11

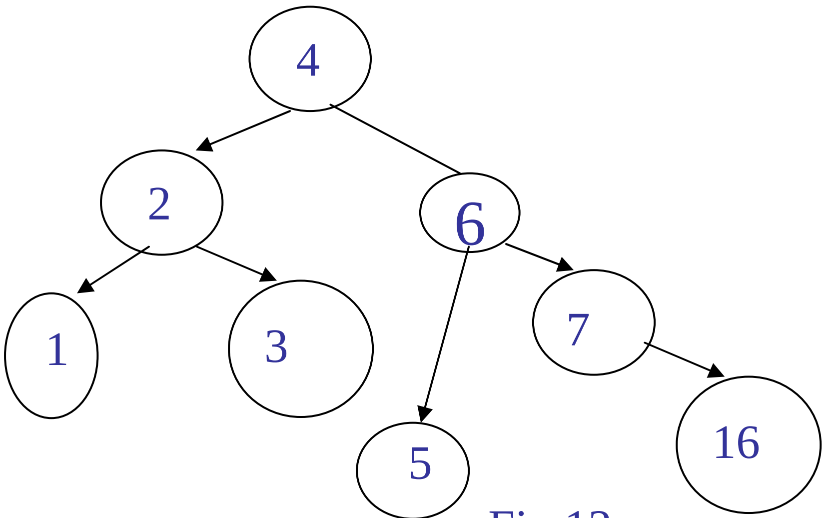


Fig 12