VE281

Data Structures and Algorithms

Recitation Class

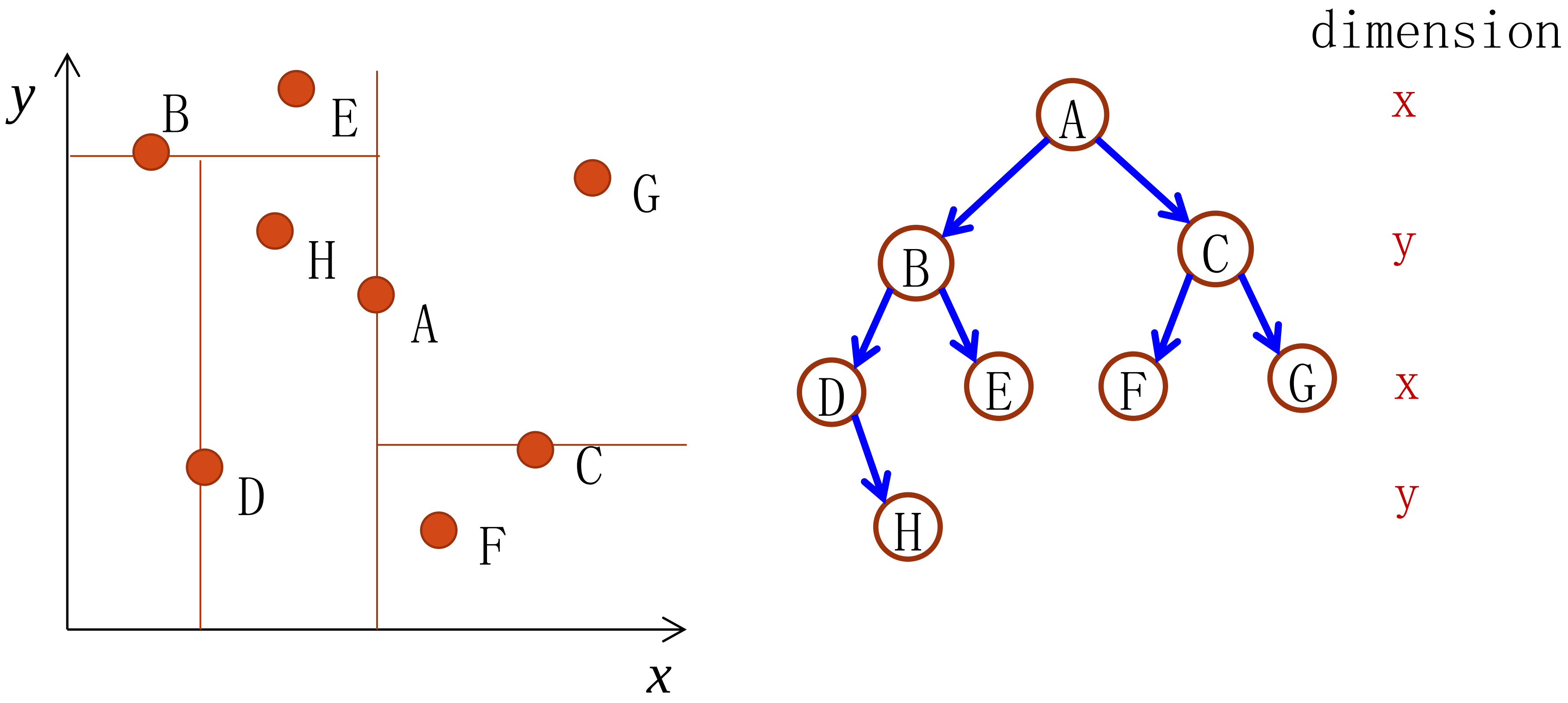
Nov. 19 2018

VE281 TA Group

k-d Trees

- A k-d tree is a binary search tree
- At each level, keys from a different search dimension is used as the discriminator
  - Nodes on the left subtree of a node have keys with value $<$ the node's key value along this dimension
  - Nodes on the right subtree have keys with value $\geq$ the node's key value along this dimension
- cycle through the dimensions as going down

k-d Trees



k-d Trees

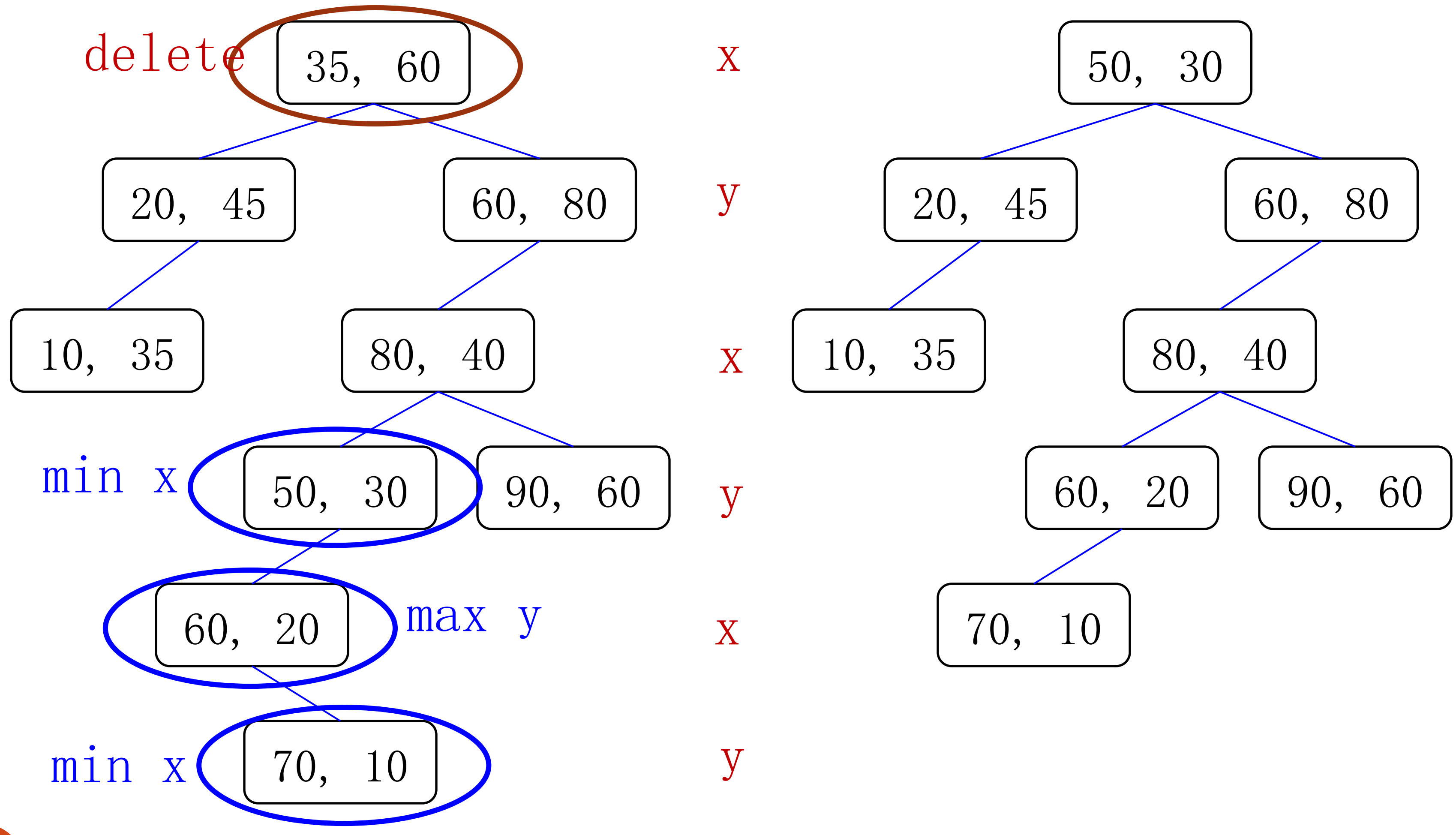
```
void insert(node *&root, Item item, int dim) {
  if(root == NULL) {
    root = new node(item);
    return;
  }
  if(item.key == root->item.key)
    return;
  if(item.key[dim] < root->item.key[dim])
    insert(root->left, item, (dim+1)%numDim);
  else
    insert(root->right, item, (dim+1)%numDim);
}
```

k-d Trees

```
node *search(node *root, Key k, int dim) {
  if(root == NULL) return NULL;
  if(k == root->item.key)
    return root;
  if(k[dim] < root->item.key[dim])
    return search(root->left, k, (dim+1)%numDim);
  else
    return search(root->right, k, (dim+1)%numDim);
}
```

Time complexity: $O(\log n)$

k-d Trees



delete
35, 60
20, 45
60, 80
10, 35
80, 40
min x
50, 30
90, 60
60, 20
max y
min x
70, 10
x
y
x
y
x
y
50, 30
20, 45
60, 80
10, 35
80, 40
60, 20
90, 60
70, 10

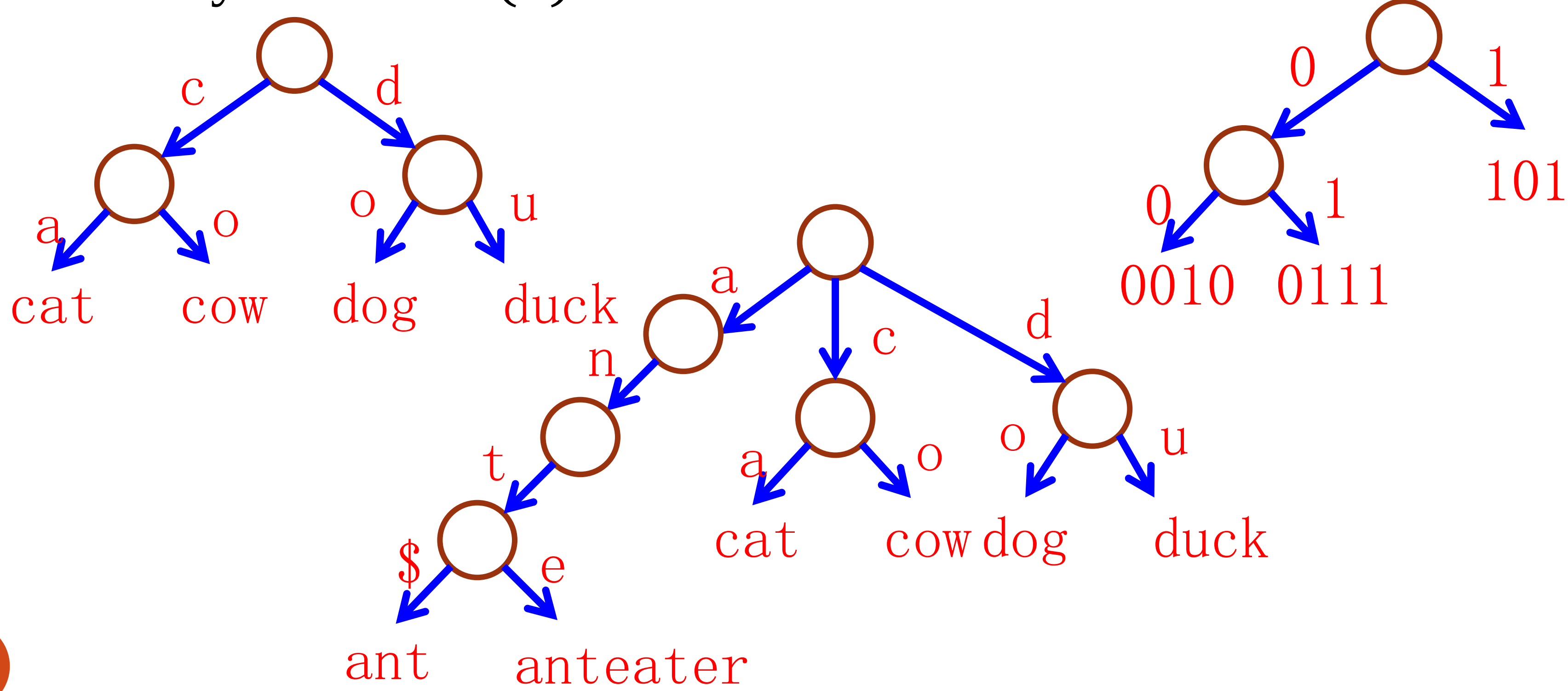
k-d Trees

```
node *findMin(node *root, int dimCmp, int dim) {
// dimCmp: dimension for comparison
  if(!root) return NULL;
  node *min =
    findMin(root->left, dimCmp, (dim+1)%numDim);
  if(dimCmp != dim) {
    rightMin =
     findMin(root->right, dimCmp, (dim+1)%numDim);
    min = minNode(min, rightMin, dimCmp);
  }
  return minNode(min, root, dimCmp);
}


void rangeSearch(node *root, int dim,
      Key searchRange[], Key treeRange[],
      List results)
```

Tries

- A trie is a tree that uses parts of the key, as opposed to the whole key, to perform search.
- In the worst case, inserting or finding a key that consists of k symbols is $O(k)$.



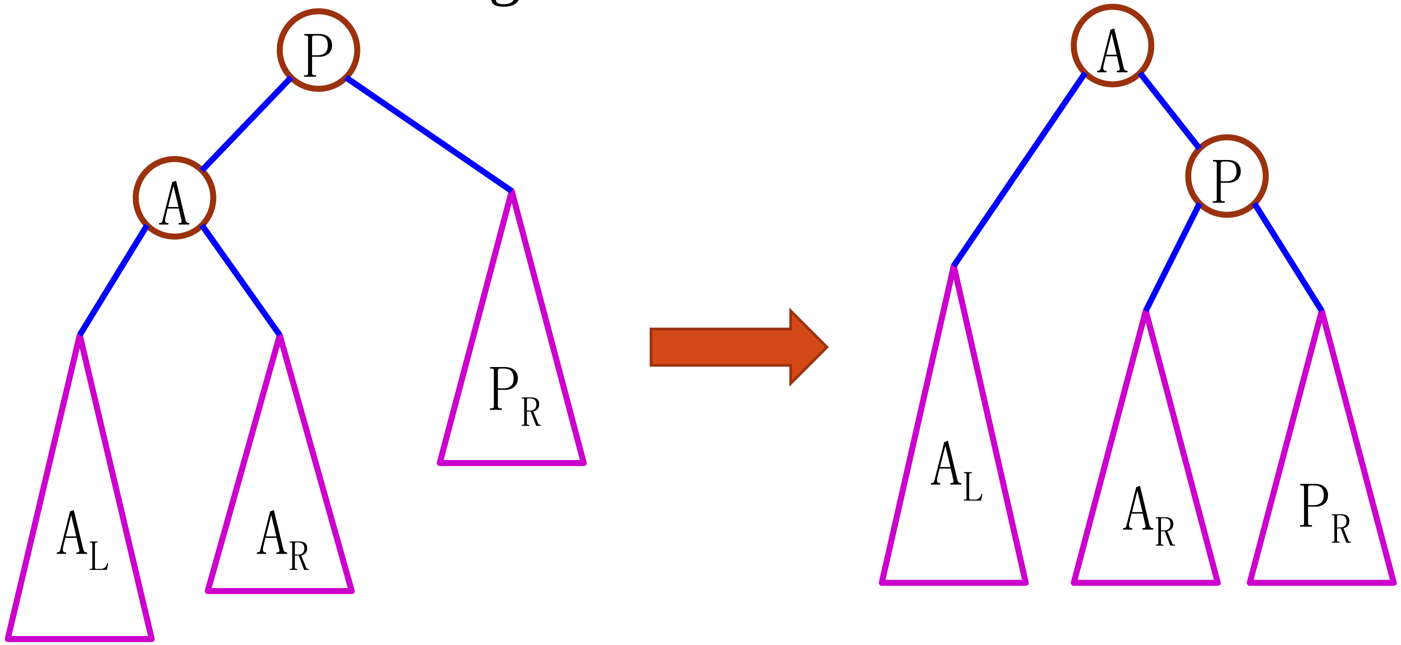
AVL Trees

- Balanced Search Tree
  1. Height of a tree of n nodes = O(log n).
  2. Balance condition can be maintained efficiently: O(log n) time to rebalance a tree.
- AVL trees' balance condition:
  - An empty tree is **AVL balanced**.
  - A non-empty binary tree is **AVL balanced** if
  1. Both its left and right subtrees are AVL balanced, and
  2. The height of left and right subtrees differ by **at most 1**.
- Height: $\log_2(n+1) - 1 \leq h \leq 1.44 \log_2(n+2)$
- Re-balance: after each insertion or removal

AVL Trees

- Right Rotation

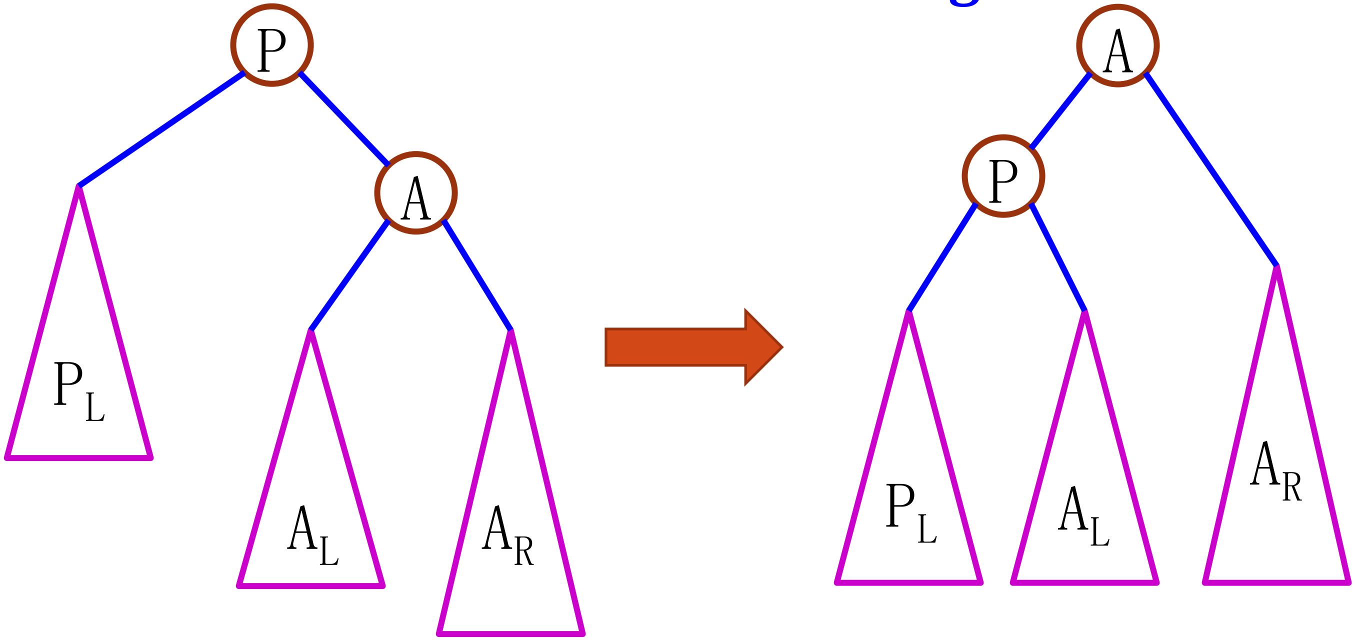
1. The right link of the **left child** becomes the left link of the **parent**.
2. **Parent** becomes right child of the **old left child**.



AVL Trees

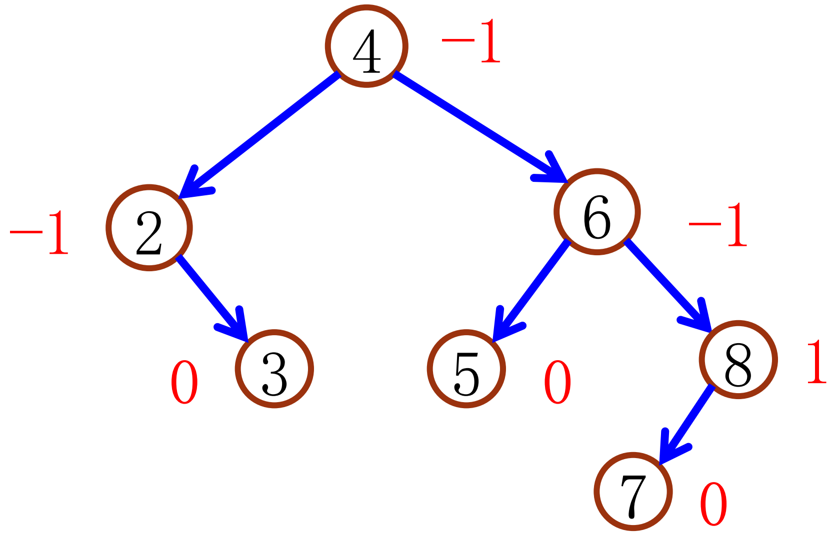
- Left Rotation

1. The left link of the **right child** becomes the right link of the **parent**.
2. **Parent** becomes left child of the **old right child**.



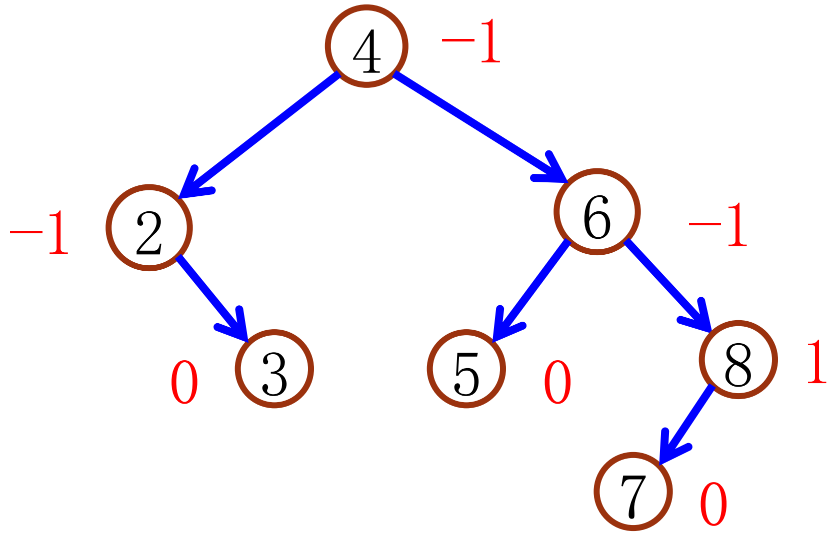
AVL Trees

- Balance Factor $B_T = h_l - h_r$
- Balance Condition $|B_T| \leq 1$.



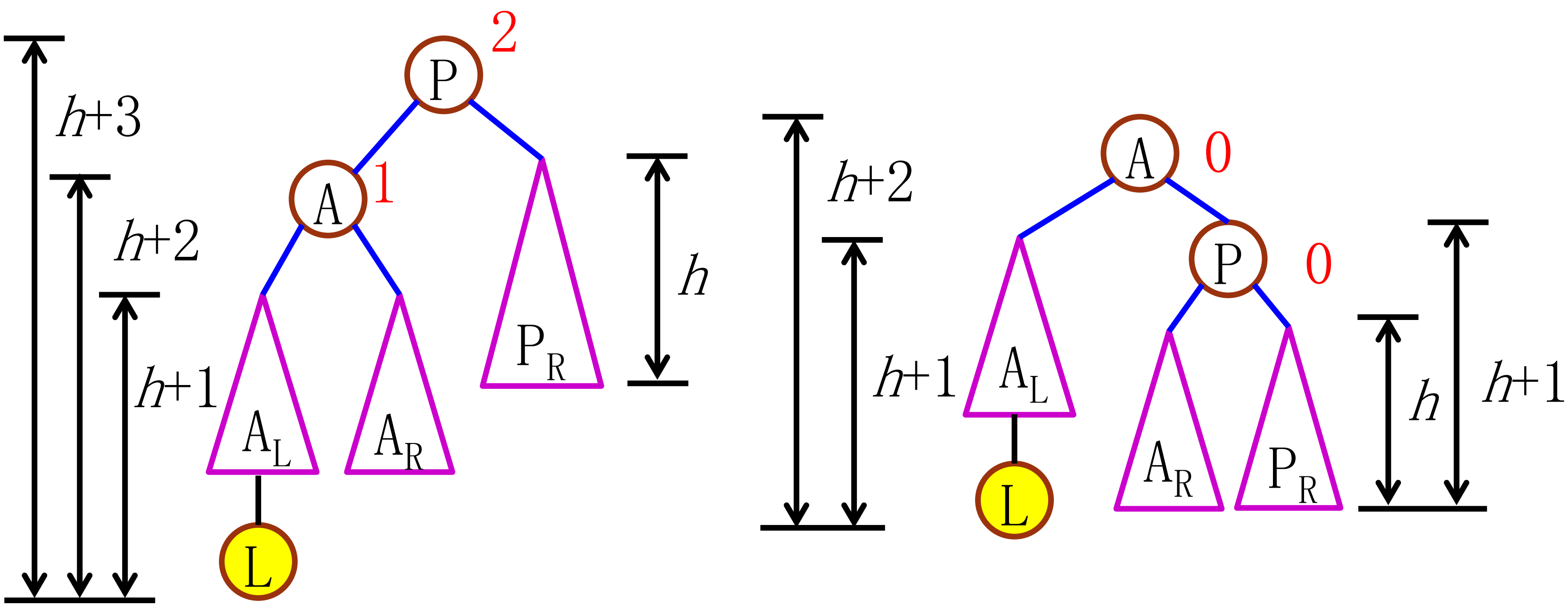
AVL Trees

- Balance Factor $B_T = h_l - h_r$
- Balance Condition $|B_T| \leq 1$.



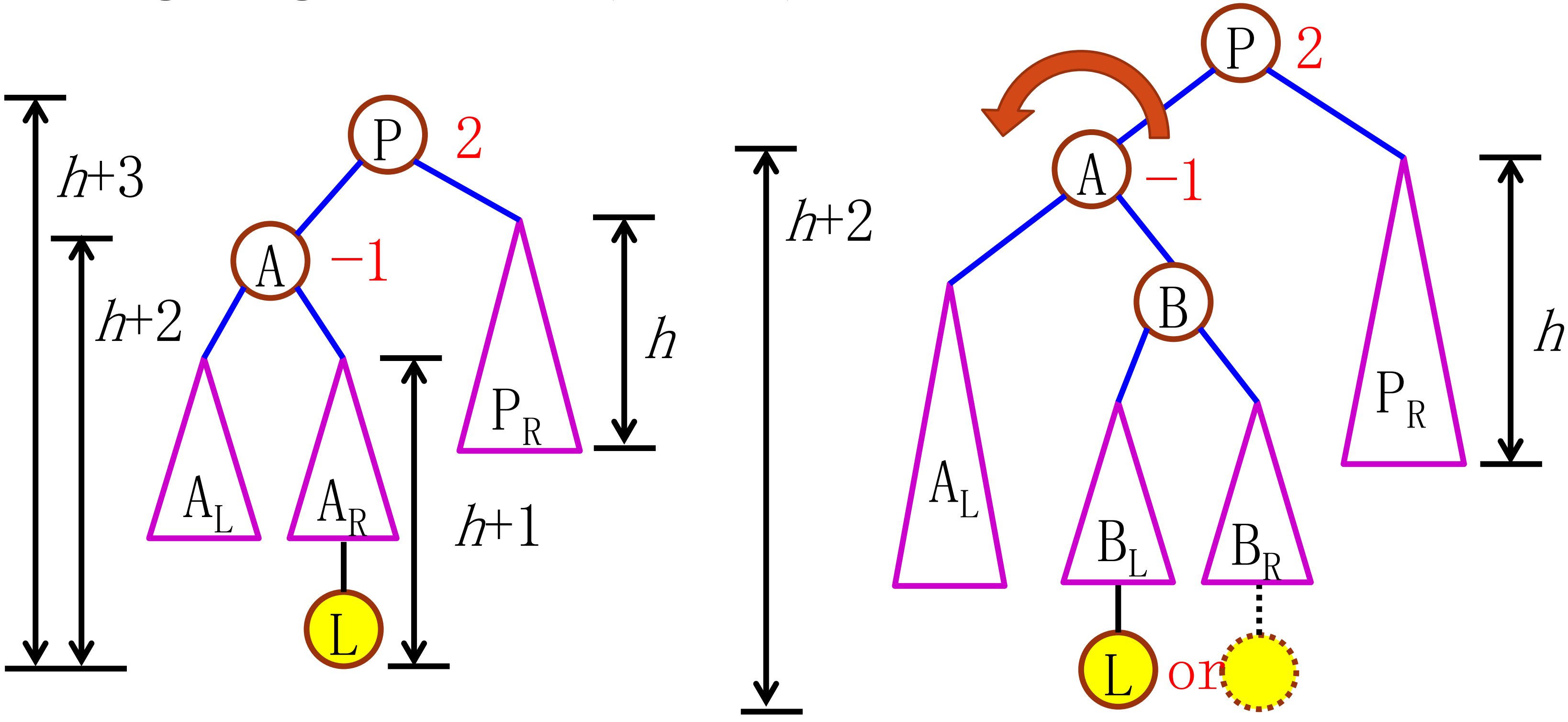
AVL Trees

- Left-left Insertion
- Right-right insertion(similar)



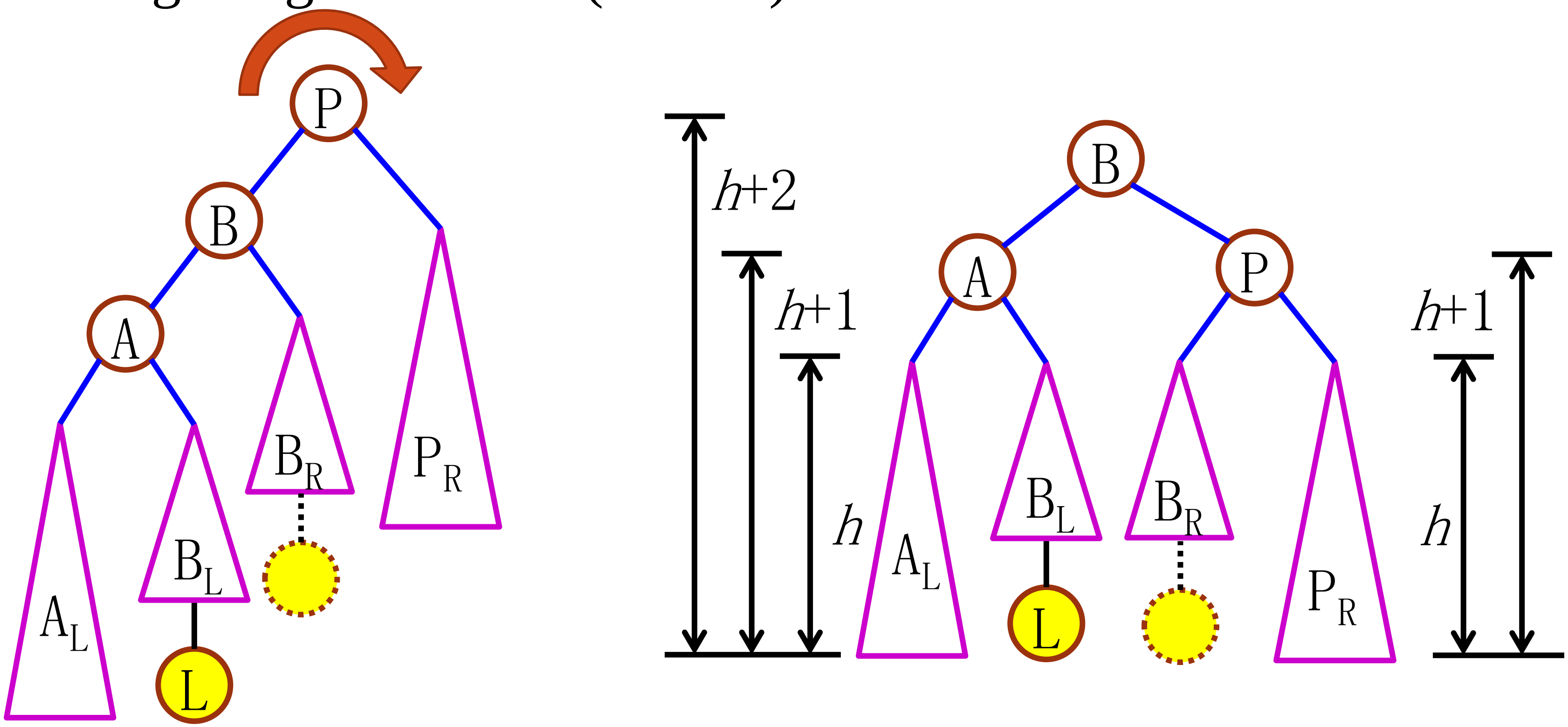
AVL Trees

- Left-right Insertion
- Right-right insertion(similar)



AVL Trees

- Left-right Insertion
- Right-right insertion(similar)



AVL Trees

```
int Height(node *n) {
  if(!n) return -1;
  return n->height;
}
void AdjustHeight(node *n) {
  if(!n) return;
  n->height = max( Height(n->left),
    Height(n->right) ) + 1;
}

int BalFactor(node *n) {
  if(!n) return 0;
  return (Height(n->left) - Height(n->right));
}
```

AVL Trees

```
void LLRotation(node *&n);
void RRRotation(node *&n);
void LRRotation(node *&n);
void RLRotation(node *&n);

void Balance(node *&n) {
  if(BalFactor(n) > 1) {
    if(BalFactor(n->left) > 0) LLRotation(n);
    else LRRotation(n);
  }
  else if(BalFactor(n) < -1) {
    if(BalFactor(n->right) < 0) RRRotation(n);
    else RLRotation(n);
  }
}
```

AVL Trees

```
void insert(node *&root, Item item)
{
  if(root == NULL) {
    root = new node(item);
    return;
  }
  if(item.key < root->item.key)
    insert(root->left, item);
  else if(item.key > root->item.key)
    insert(root->right, item);

  Balance(root);
  AdjustHeight(root);
}
```