VE281

Data Structures and Algorithms

AVL Trees and Midterm Review

Review

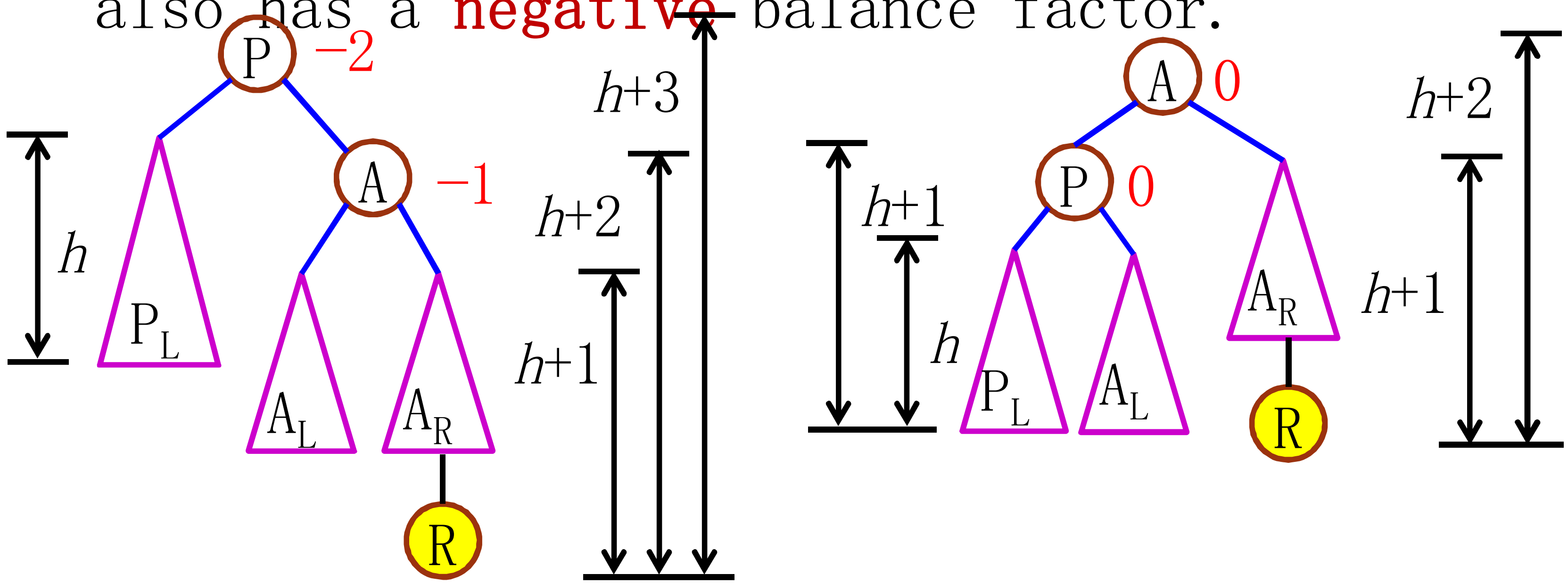
- Binary Search Trees
  - Binary search tree with **leftSize**
  - Rank search
  - Range search
- AVL Trees
  - Balance condition
  - Rotation
  - Balance factor
  - Left-left Insertion and left-left rotation

Outline

- AVL Trees
- Midterm Review

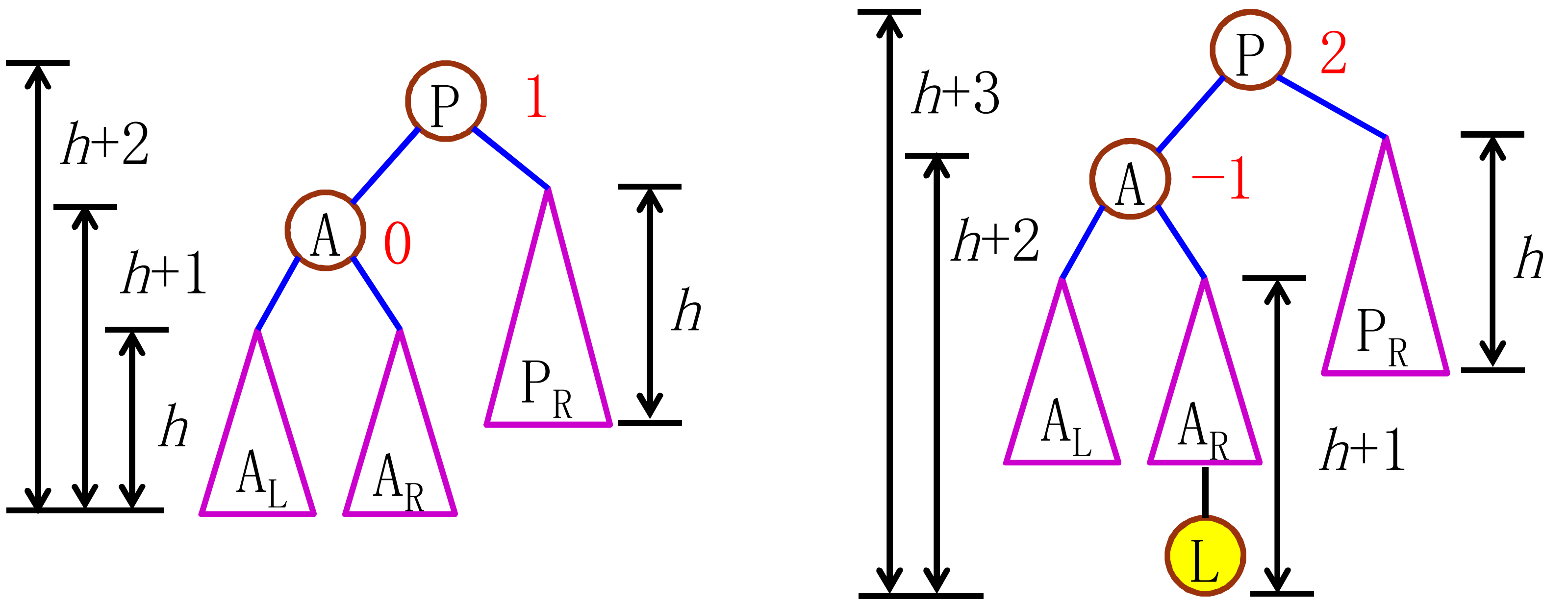
Right-Right (RR) Rotation

- Symmetric to left-left rotation.
- An RR rotation is called for when the node becomes unbalanced with a **negative** balance factor and the right subtree of the node also has a **negative** balance factor.



Breaking AVL Balance Condition

Left-Right Insertion

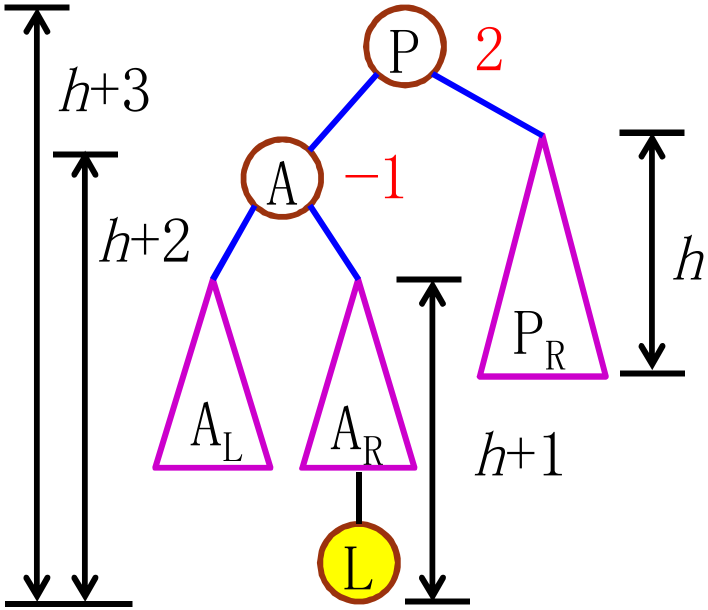


insert a new item

Left-right insertion: the first edge in the insertion path goes to the left and the second edge goes to the right.

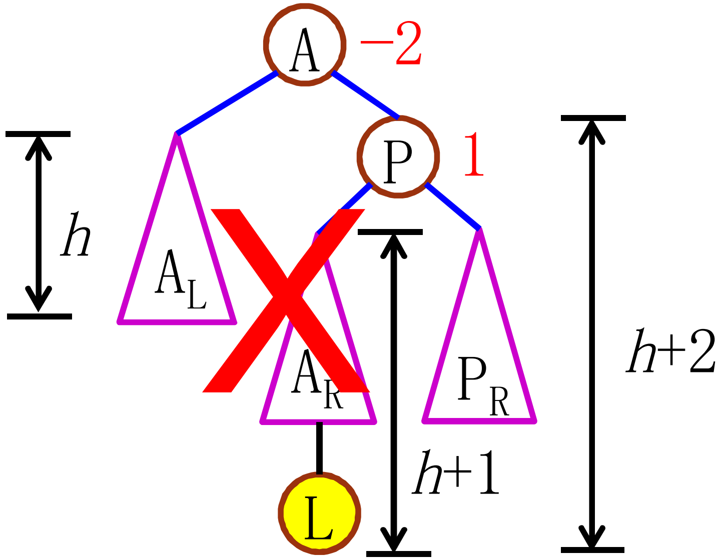
Restoring AVL Balance Condition

Left-Right Insertion

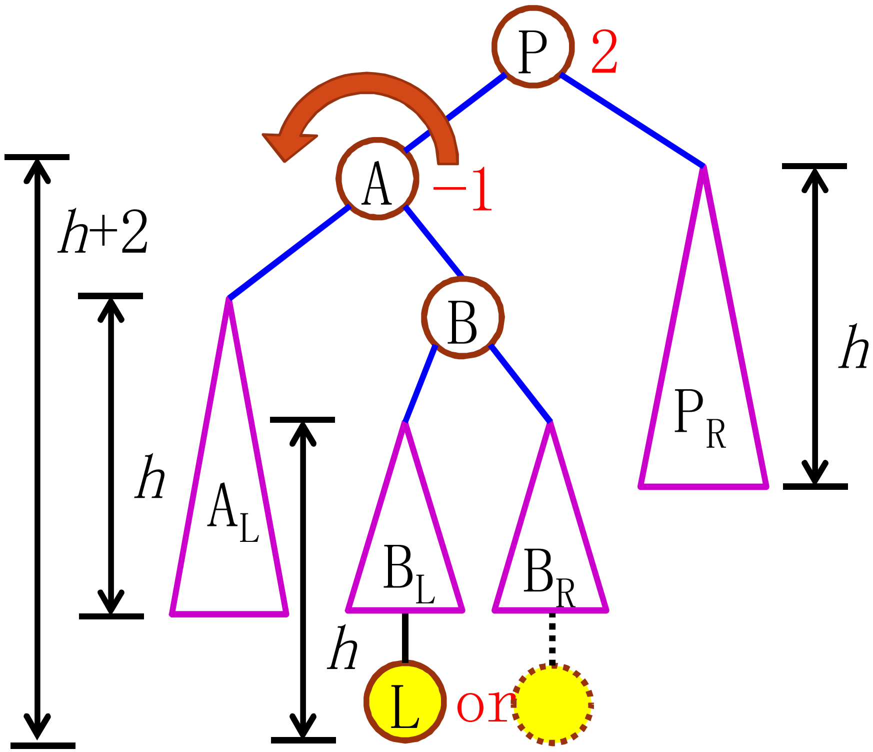


How to restore AVL balance?

A right rotation at node P does not work!



Left-Right (LR) Rotation

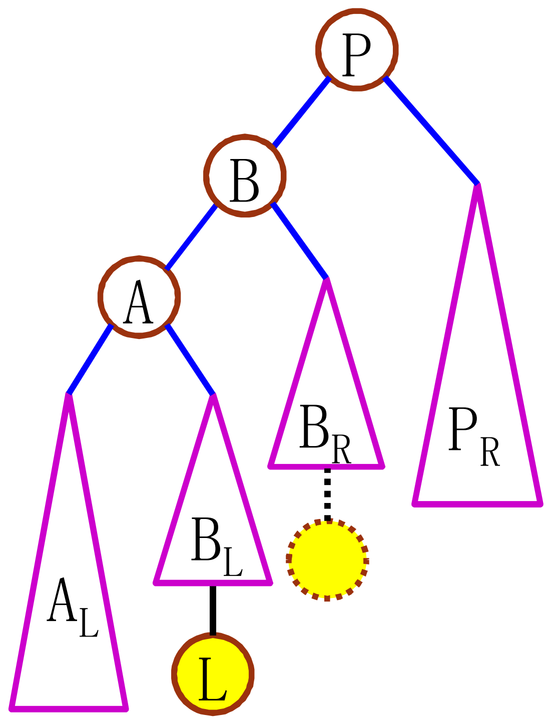


A **double rotation** to re-bal

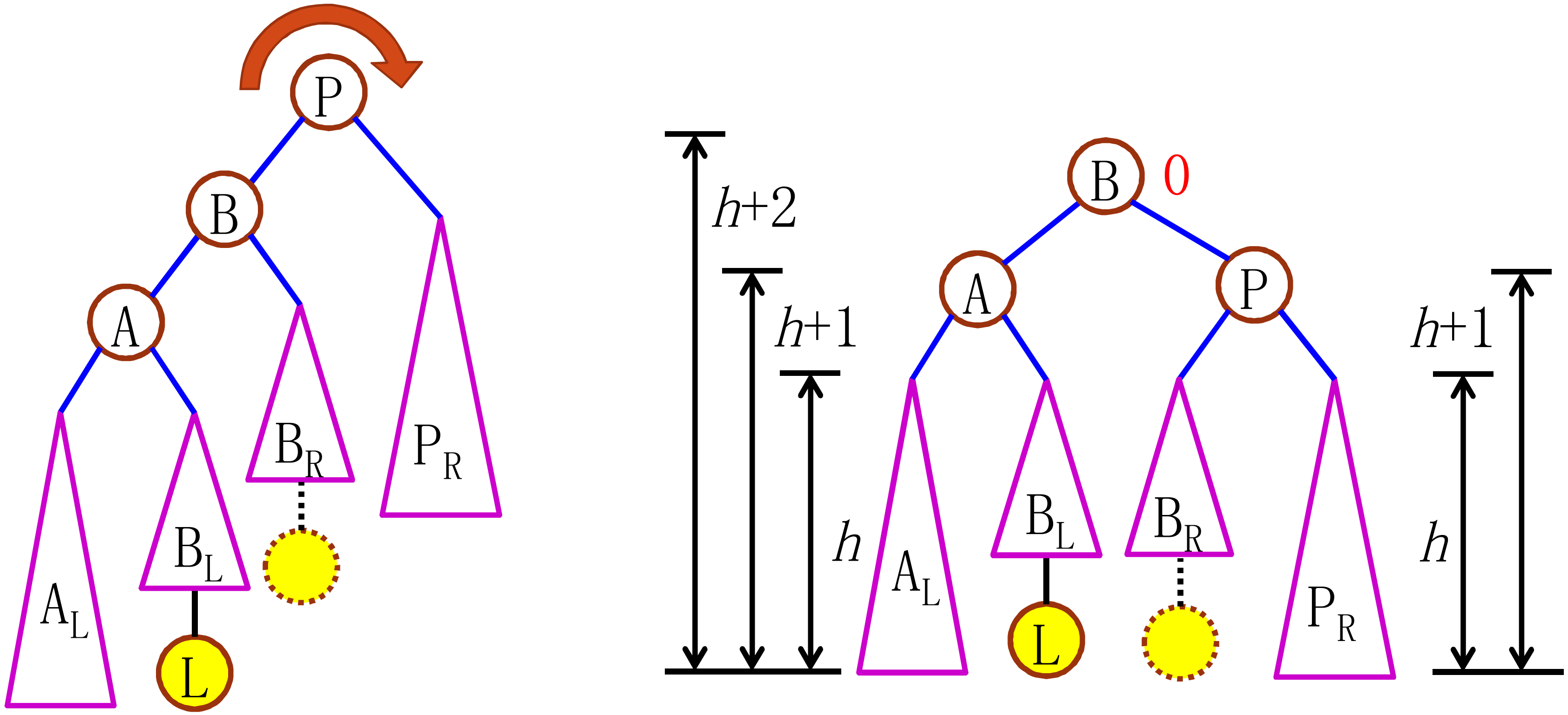
Do a **left** rotation on node

then a **right** rotation on no

(next slide).

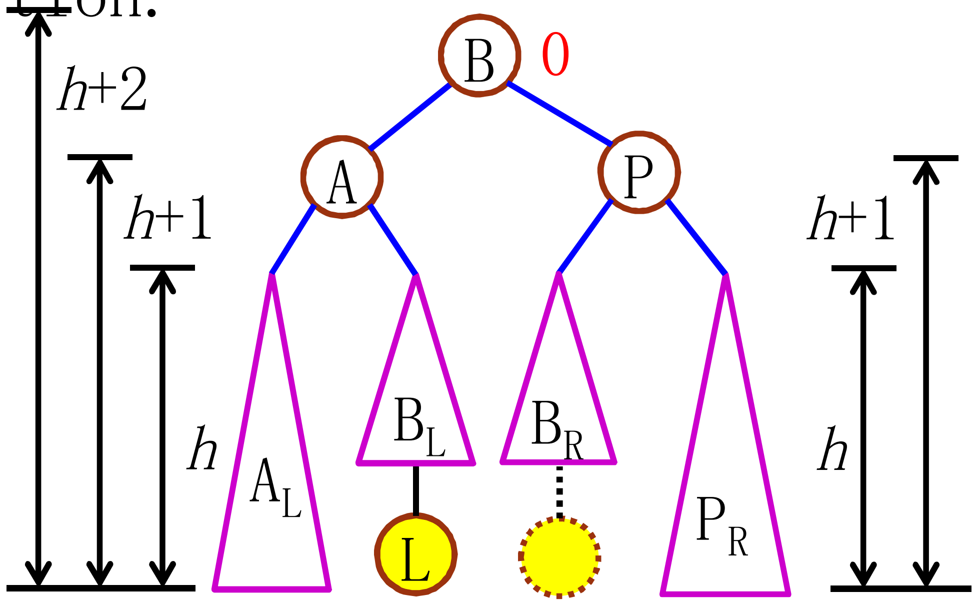


Left-Right (LR) Rotation



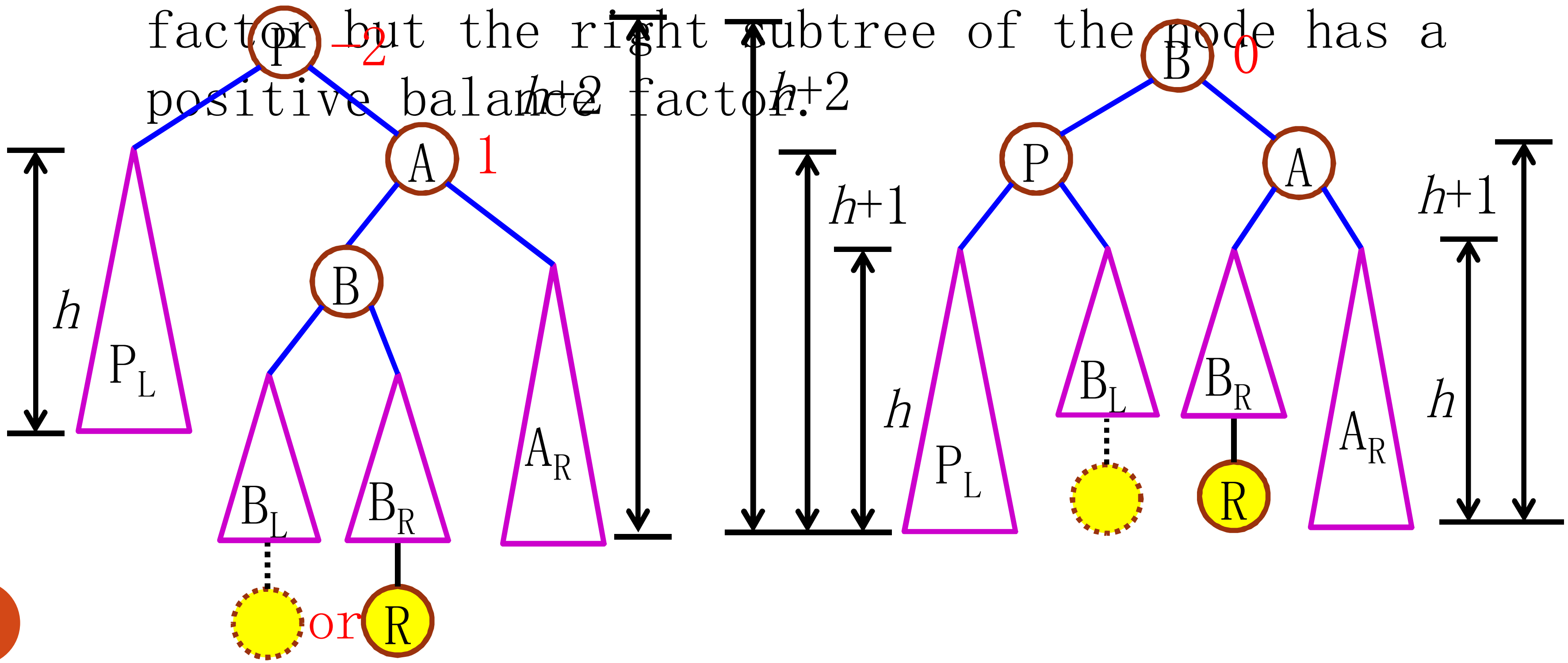
Properties of Left-Right Rotation

- The ordering property of BST is kept.
- Node B has a balance factor of 0.
- The height of the tree **after the rotation** is the same as the height of the tree before insertion.



Right-Left (RL) Rotation

- Symmetric to left-right rotation; also a double rotation.
- An RL rotation is called for when the node becomes unbalanced with a negative balance factor but the right subtree of the node has a positive balance factor.



Rotation Summary

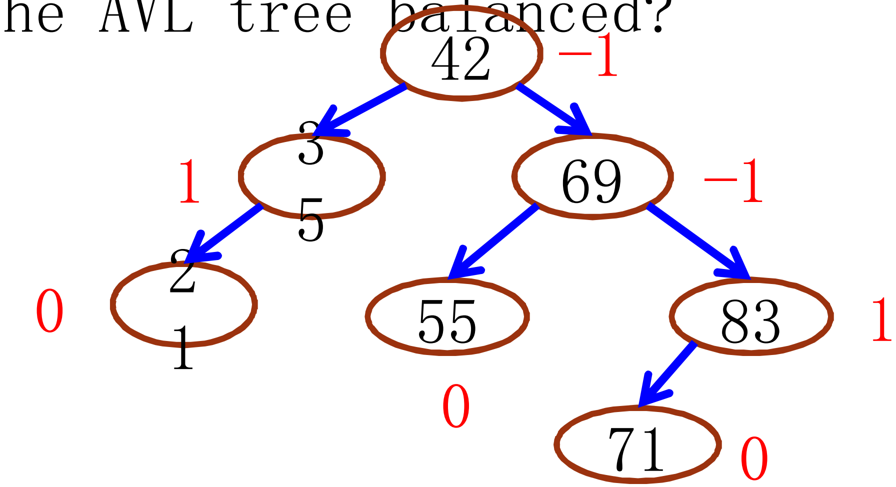
- When an AVL tree becomes unbalanced, there are four cases to consider depending on the **direction** of the first two edges on the insertion path from the **unbalanced node**:
  - Left-left
  - Right-right
  - Left-right
  - Right-left

LL Rotation
RR Rotation
single rotation

LR Rotation
RL Rotation
double rotation

Exercises

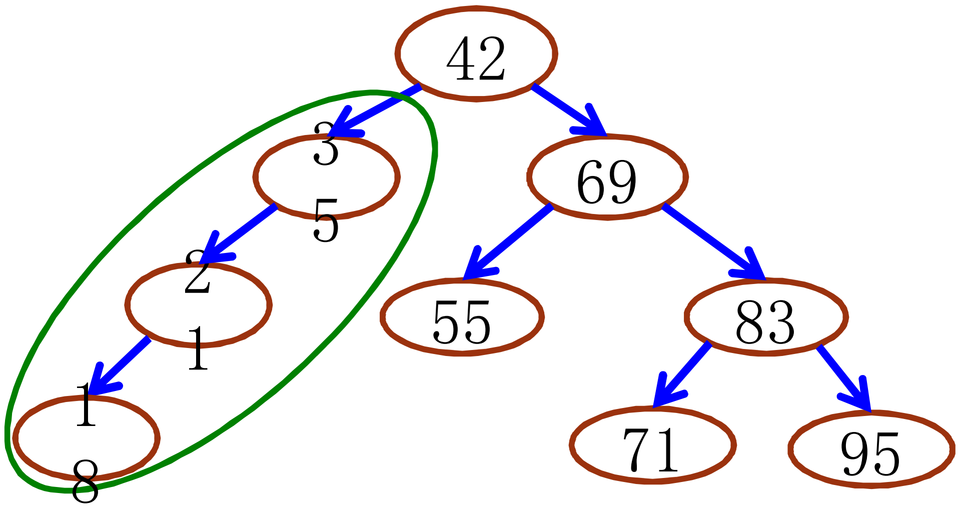
- Insert into an empty AVL tree: 42, 35, 69, 21, 55, 83, 71.
  - Compute the balance factors.
  - Is the AVL tree balanced?

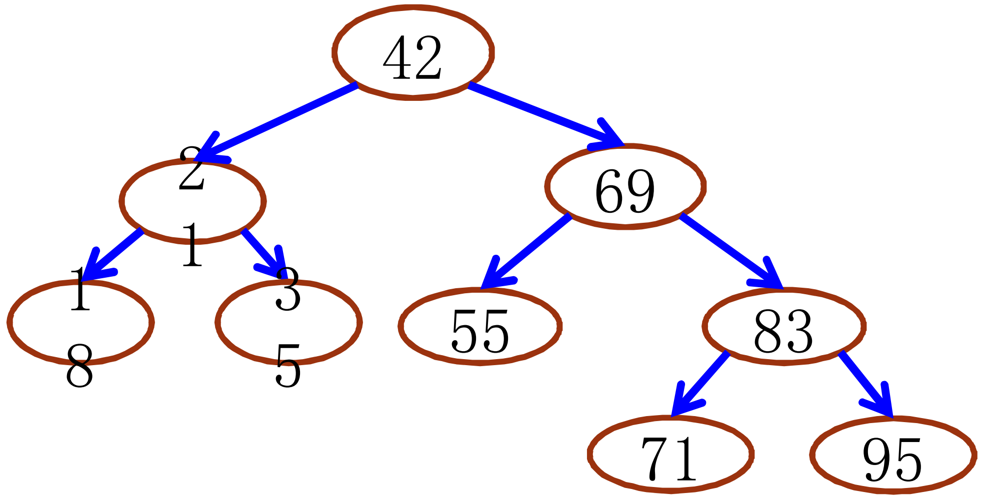


- Insert 95, 18, 75?

Exercises

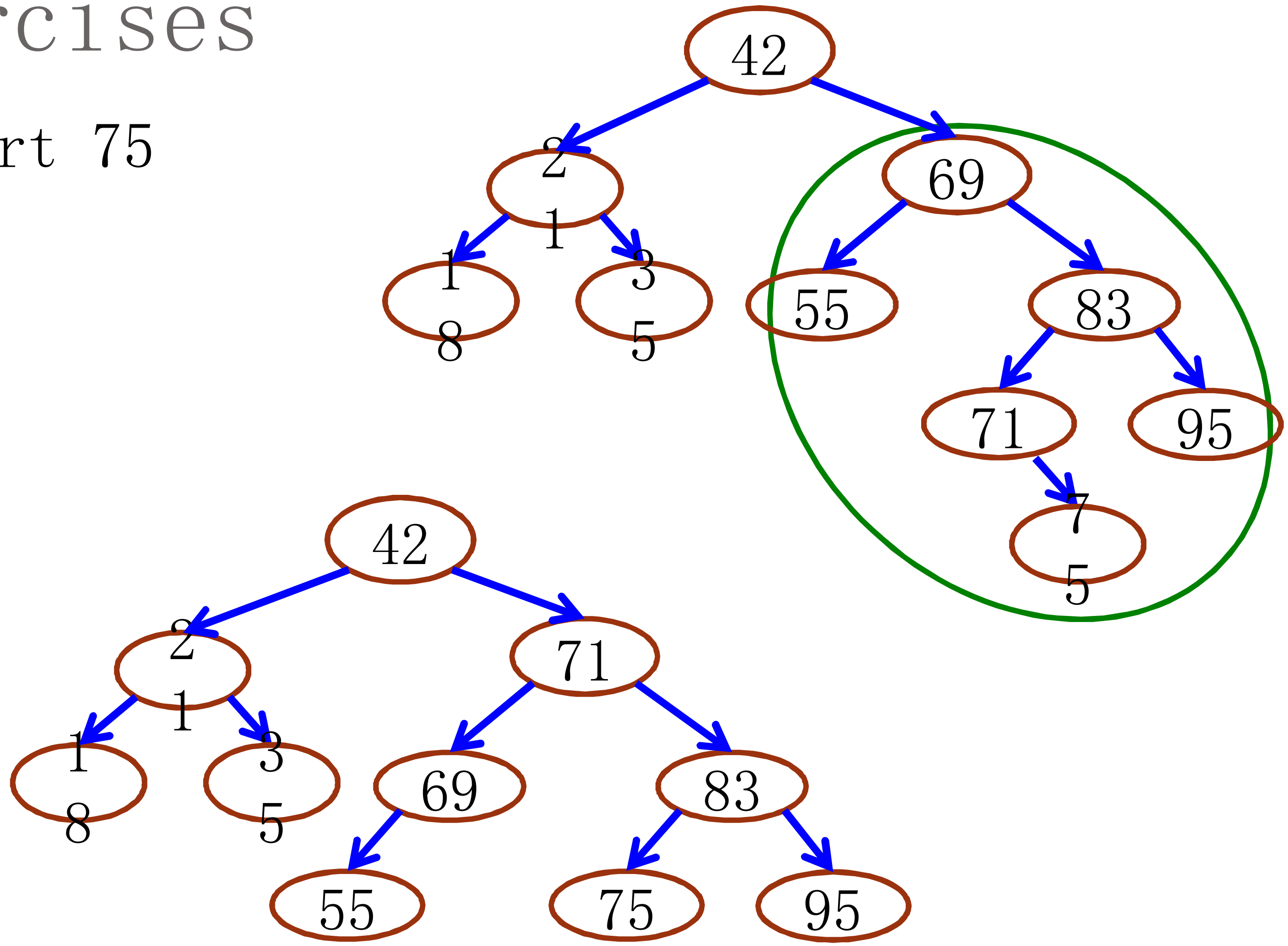
- Insert 95, 18





Exercises

- Insert 75



The Number of Rotations Required

- When an AVL tree **becomes unbalanced after an insertion**, **exactly one** single or double rotation is required to balance the tree.
  - Before the insertion, the tree is balanced.
  - Only nodes on the access path of the insertion can be unbalanced. All other nodes are balanced.
  - We rotate at the first unbalanced node from the leaf.
  - By the properties of rotation, the height of the node after rotation is the same as that before insertion.
  - All ancestors of that node on the access path should now be balanced.

AVL Trees

Supporting Data Members and Functions

```
struct node {
  Item item;
  int height;
  node *left;
  node *right;
};
```

```
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

Supporting Functions

```
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

Changes to Insertion

```
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);

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

Removal

- First remove node as with BST

- Then update the balance factors of those ancestors in the access path and rebalance as needed.

Midterm Review

Logistics of Midterm Exam

- 10:00 am - 11:40 am, Thursday, November 8th, 2012
- Location: Dong Xia Yuan 200
- Written exam with 8 questions in total.
- Closed book and closed notes.
- No electronic devices are allowed.
  - These include laptops and cell phones.
  - We will show a clock on the screen.
- Abide by the Honor Code!

Topics in Midterm Exam

- Asymptotic Algorithm Analysis
- Linked lists
- Stacks and queues
- Hashing and hash tables
- Trees and binary trees
- Binary tree traversal
- Binary search trees
- AVL trees

Asymptotic Algorithm Analysis

- Best, worst, and average cases

- Big-Oh, Big-Omega, and theta notations
  - Big-Oh: upper-bound
  - Big-Omega: lower-bound

- Common functions and their growth rates
  - $\log n, n, n \log n, n^k, 2^n, n!$

- Analyzing the time complexity of programs
  - Loop statement

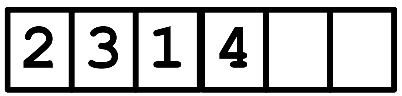
Linked Lists

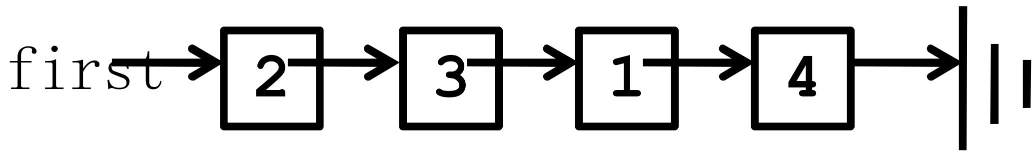
- Basic methods:
  insertFirst()
  removeFirst()

- Methods requiring linked list traversal:
  getSize()
  appendNode()
  removeNode()

Arrays versus Linked Lists

Worst Case Time Complexity





	Array	Linked List
Random access	$O(1)$ time	$O(n)$ time
insertFirst	$O(n)$ time	$O(1)$ time
removeFirst	$O(n)$ time	$O(1)$ time
appendNode	$O(1)$ time	$O(n)$ time
removeNode	$O(n)$ time	$O(n)$ time

Linked List Optimization

- **`appendNode()`**
  - Double-ended singly-linked list

- **`removeNode()`**
  - Double-ended doubly-linked list
  - How to reduce the time complexity to $O(1)$ with a singly-linked list?

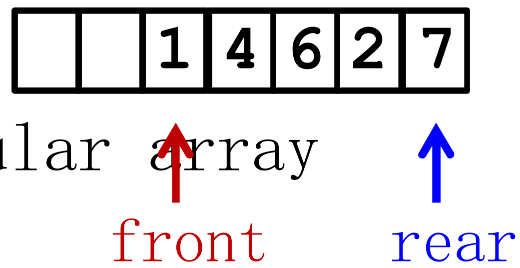
- Reverse a linked list

Stacks

- A "pile" of objects where new object is put on top of the pile and the top object is removed first.
  - LIFO access: last in, first out.
- Methods: **push**, **pop**, **size**, etc.
- Implementations: arrays versus linked lists
- Applications
  - Web browser's "back" feature
  - Parentheses matching

Queues

- A "line" of items in which the first item inserted into the queue is the first one out.
  - FIFO access: first in, first out
- Methods: **enqueue**, **dequeue**, **size**, etc.
- Implementations by linked lists
  - Double-ended singly-linked list



- Implementations by arrays: circular array
  - "circular" increment
    front = (front+1) % MAXSIZE;
  - Distinguish between an empty queue and a full queue

Queues

- Application: wire routing - lee’ s algorithm
- Deque: combination of stack and queue

Dictionary

- A collection of pairs, each containing a key and an element

  (key, element)

- Different pairs have different keys.

- Operations: search, insertion, and removal by keys.

- Implementations using arrays or linked lists.

Hashing and Hash Table

- Access table items by their keys in time that is relatively constant regardless of their locations.

- Main idea: use arithmetic operations, known as hash function, to transform keys into table locations.
  - The same key is always hashed to the same location.
  - Thus, **insert()** and **find()** are both directed to the same location in $O(1)$ time.

- Hash table: An array of buckets, where each bucket contains items as assigned by a hash

Hash Table Issues

- Choice of the hash function.
- Collision resolution scheme.
- Size of the hash table and rehashing.

Hash Functions

- Hash function (**h(key)**) maps key to buckets in two steps:

1. Convert key into an integer in case the key is not an integer.
   - A function **t(key)** which returns an integer value, known as hash code.
   - How to map strings to integers?
2. Compression map: Map an integer (hash code) into a home bucket.
   - A function **c(hashcode)** which gives an integer in the range [0, M-1], where M is the number of buckets in the table.
   - Compression mapping by modulo arithmetic: How to choose the divisor M?

Collision and Collision Resolution

- Collision occurs when the hash function maps two or more items—all having **different** search keys—into the **same** bucket.

- **Collision-resolution scheme**: assigns distinct locations in the hash table to items involved in a collision.
- Two major schemes:
  - Separate chaining: Each bucket keeps a **linked list** of all items whose home buckets are that bucket.
  - Open addressing: Probe with a sequence of hash functions h_0, h_1, h_2, . . .

Open Addressing

- Linear probing:
  $h_i(x) = (h(x) + i) \% M$
  - **insert**, **find**, **remove**
  - The problem of clustering

- Quadratic probing:
  $h_i(x) = (h(x) + i^2) \% M$

- Double hashing:
  $h_i(x) = (h(x) + i*g(x)) \% M$

Average Number of Comparisons

- Depends on the **load factor** $L = N/M$, where N is the number of items in the hash table and M is the size of the hash table.

- We analyze both the case of unsuccessful search ($U(L)$) and the case of successful search ($S(L)$).

Hash Table Size and Rehashing

- **Rehashing**: Create a larger table, scan the current table, and then insert items into new table using the new hash function.
  - We can approximately double the size of the current table.
  - The single operation of rehashing is time-consuming. However, it does not occur frequently.

- **Amortized analysis**: A method of analyzing algorithms that considers the entire sequence of operations of the program.
  - The cost is **averaged** over a sequence of operations.
  - Amortized analysis of rehashing: the average cost to insert $M + 1$ items is O(1).

Trees

- Root, leaf, subtree, parent, child, sibling, path, ancestors, and descendants.

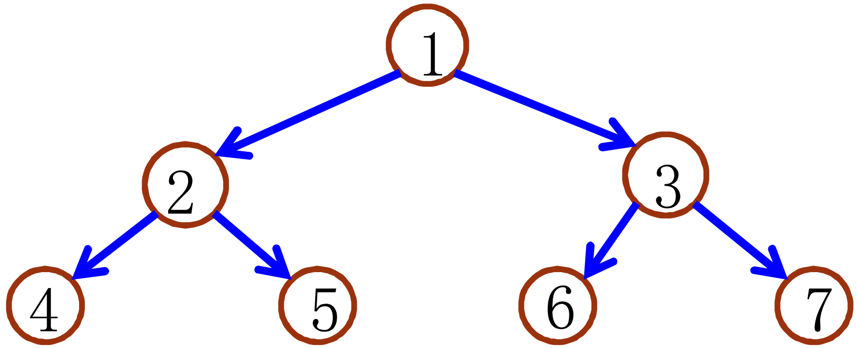
- Depth, height, level, degree of a node/tree

Binary Trees

- The relation between the number of nodes and the height

$$h + 1 \leq n \leq 2^{h+1} - 1$$

- **Proper**, **complete**, and **perfect** binary trees.
- Numbering nodes in a perfect binary tree.



- Representing a binary tree using linked structure.

```
struct node {
  Item item;
  node *left;
  node *right;
};
```

Binary Tree Traversal

- Depth-first traversal
  - Pre-order
  - Post-order
  - In-order
  - Implemented with recursion
  - Implemented with a stack

- Level order traversal
  - Implemented with a queue

Binary Search Trees

- A binary search tree (BST) is a binary tree with the following properties:
  - Each node is associated with a **key**. A key is a value than can be compared.
  - The key of any node is greater than the keys of all nodes in its left subtree and smaller than the keys of all nodes in its right tree.

- Operations
  - Search, insertion, and removal by keys.
  - Rank search
  - Range search

Binary Search Trees

Insertion and Removal

- Insertion
  - Insertion inserts the item **as a leaf** of the BST.
  - It inserts at a proper location in the BST, maintaining the BST properties.

- Removal
  - Node to be removed is a leaf: delete the node.
  - Node to be removed is a degree-one node: “bypass” the node from its parent to its child.
  - Node to be removed is a degree-two node: replace the node key with the largest key in the left subtree.

Average Case Time Complexity

- The average case time complexity for a **successful/unsuccessful** search with BST is $\Theta(\log n)$.

	Search	Insert/Remove
Linked List	$O(n)$	$O(n)$
Sorted Array	$O(\log n)$	$O(n)$
Hash Table (Separate Chaining)	$O(L)$	$O(L)$
BST	$O(\log n)$	$O(\log n)$

AVL Trees

- Motivation: the worst case time complexity for search/insertion/removal is still $O(\log n)$.

- Balance condition of AVL trees
  - 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**.

- Balance factor: $B_T = h_l - h_r$

Balance AVL Trees: Rotation

- When an AVL tree becomes unbalanced, there are four cases to consider depending on the **direction** of the first two edges on the insertion path from the **unbalanced node**:
  - Left-left
  - Right-right
  - Left-right
  - Right-left

LL Rotation
RR Rotation
single rotation

LR Rotation
RL Rotation
double rotation

Good Luck to Everyon

Questions?