Tree & Binary Trees(4)

College of Computer Science, CQU

outline

- Binary Search Trees (BST) Definition
- Binary Search Tree Implementation
- AVL Tree

A Taxonomy of Trees

- **General Trees – any number of children / node**
- **Binary Trees – max 2 children / node**
  - Heaps – parent < (>) children
  - Binary Search Trees

BST：Motivation

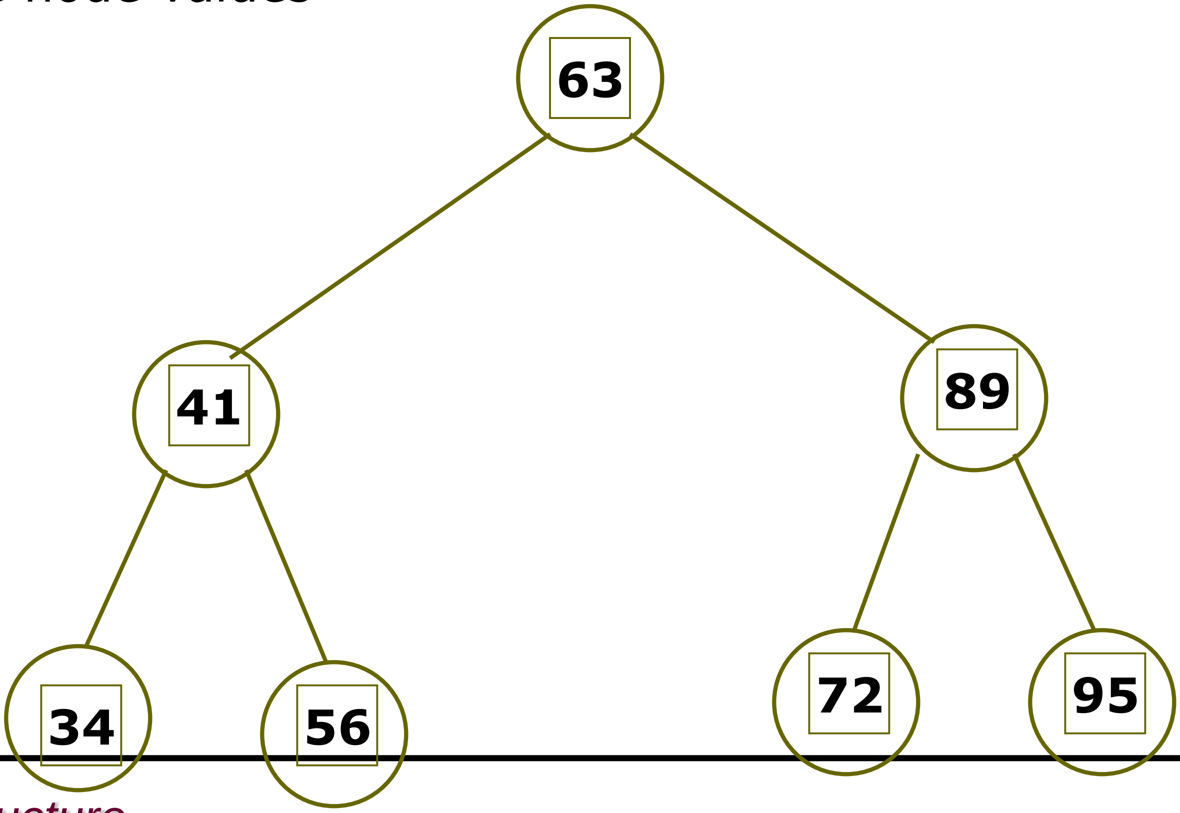
- **Binary search For sorted array search**
  - search： $\Theta(\log n)$， fast
  - insertion ： $\Theta(n)$ on average， slow
    - once the proper location for the new record in the sorted list has been found, many records might be shifted to make room for the new record.
- **Is there some way to organize a collection of records so that inserting records and searching for records can both be done quickly?**

Binary Search Trees

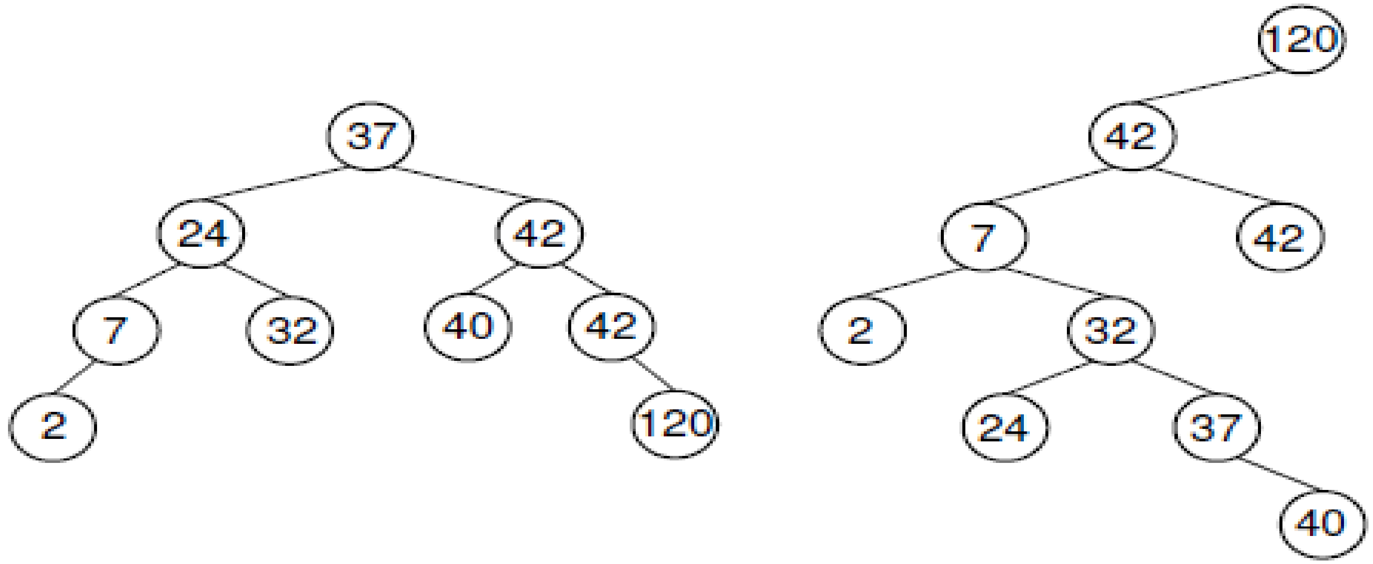
- **Binary search tree (BST)**
  - A node has a key value K
  - All nodes stored in the left subtree of the node have key values **less than** *K*. All nodes stored in the right subtree of the node have key values **greater than or equal to** *K*.
  - The left and right subtrees are also binary search trees.
- **if the BST nodes are printed using an inorder traversal, the resulting enumeration will be in sorted order from lowest to highest.**

Organization Rule for BST

- the values in all nodes in the left subtree of a node are less than the node value
- the values in all nodes in the right subtree of a node are greater than the node values



BST Example



•**Two Binary Search Trees for a collection of values. Tree (a) results if values are inserted in the order 37, 24, 42, 7, 2, 40, 42, 32, 120. Tree (b) results if the same values are inserted in the order 120, 42, 42, 7, 2, 32, 37, 24, 40.**

Binary Search Trees

- **Binary Search Trees (BST) are a type of Binary Trees with a special organization of data.**
- **This data organization leads to $\Theta(\log n)$ complexity for searches, insertions and deletions in certain types of the BST (balanced trees).**
  - O(h) in general

BST Operations: Search

Searching in the BST

`findhelp(root,key)`

• implements the binary search based on comparison of the keys in the tree

• the keys in the BST must be comparable (e.g integers, string, etc.)

The search starts at the root. It probes down, comparing the values in each node with the target, till it finds the first key equal to the target. Returns the value of this node or **`null`** if there is none.

Search in BST - Pseudocode

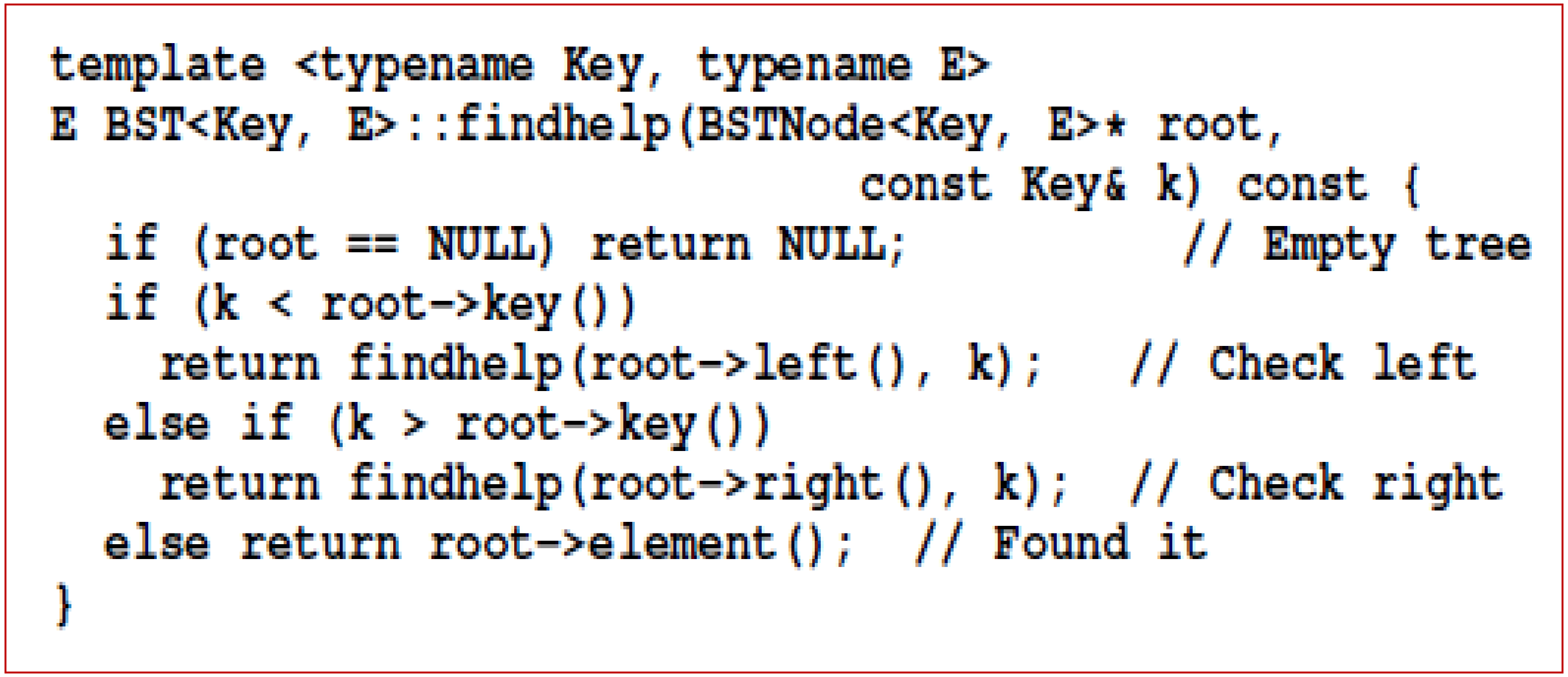
```
if the tree is empty
        return NULL

else if  the key in the node equals the target
        return the node value

else if  the key in the node is greater than the
target return the result of searching the left
subtree

else if  the key in the node is smaller than the
target return the result of searching the right
subtree
```

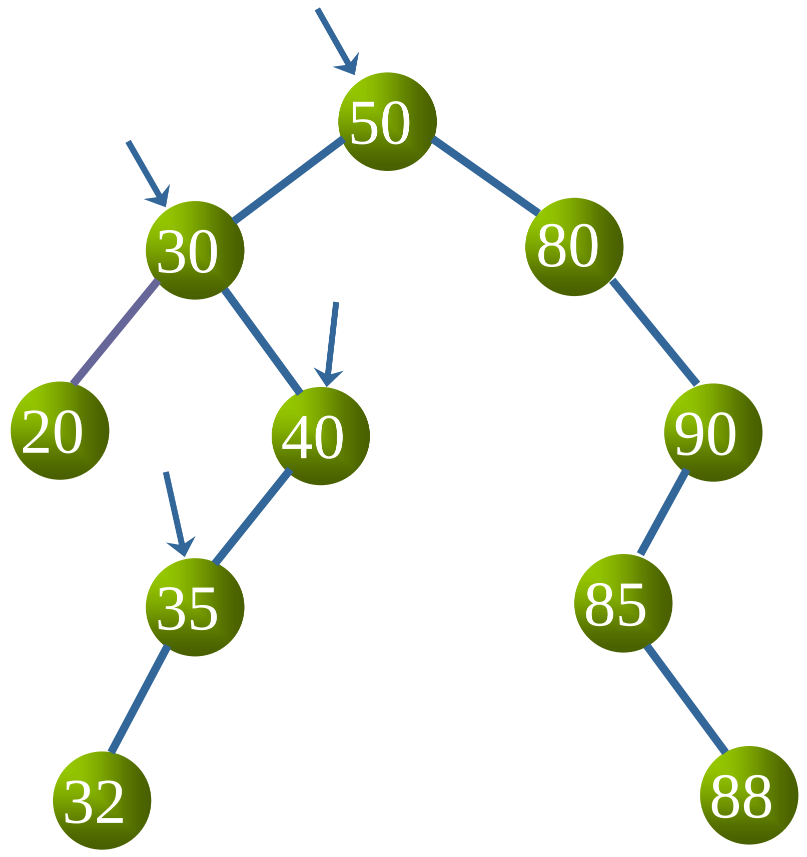
Search in BST - implementation

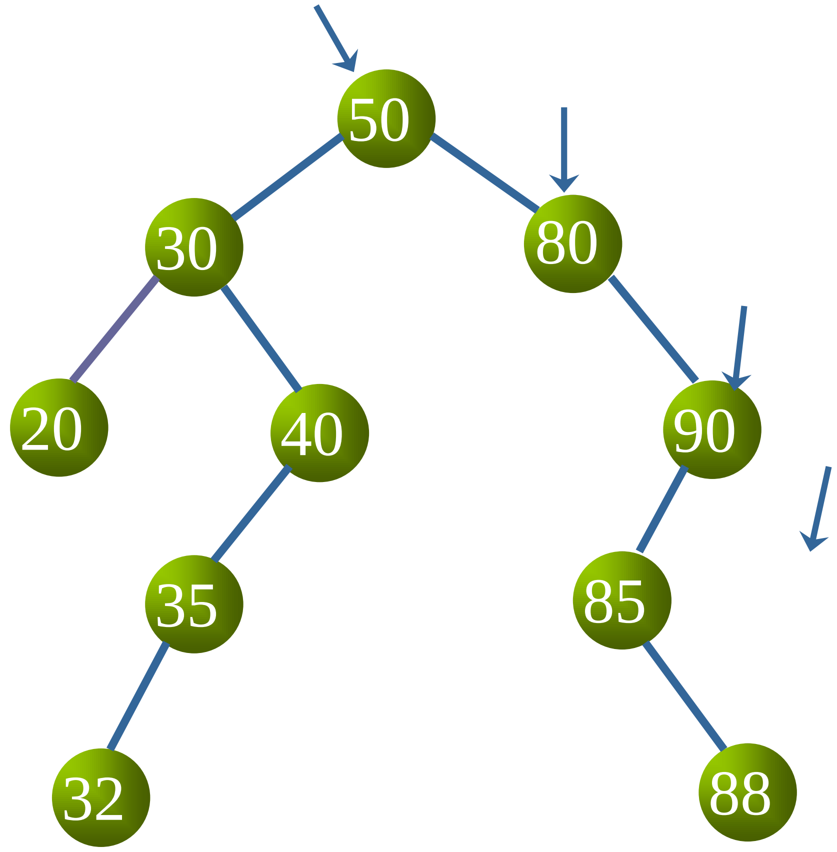


```
template <typename Key, typename E>
E BST<Key, E>::findhelp(BSTNode<Key, E>* root,
                              const Key& k) const {
  if (root == NULL) return NULL;          // Empty tree
  if (k < root->key())
    return findhelp(root->left(), k);    // Check left
  else if (k > root->key())
    return findhelp(root->right(), k);  // Check right
  else return root->element();  // Found it
}
```

Search in BST - Example

Search for 35， 95





BST Operations: Insertion

`method insert(key,it)`

- places a new key near the frontier of the BST while retaining its organization of data:
  - **starting at the root** it probes **down** the tree till it finds a node whose left or right pointer is empty and is a logical place for the new key
  - uses a binary search to locate the insertion point
  - is based on comparisons of the new key and values of nodes in the BST
    - *Elements in nodes must be comparable!*

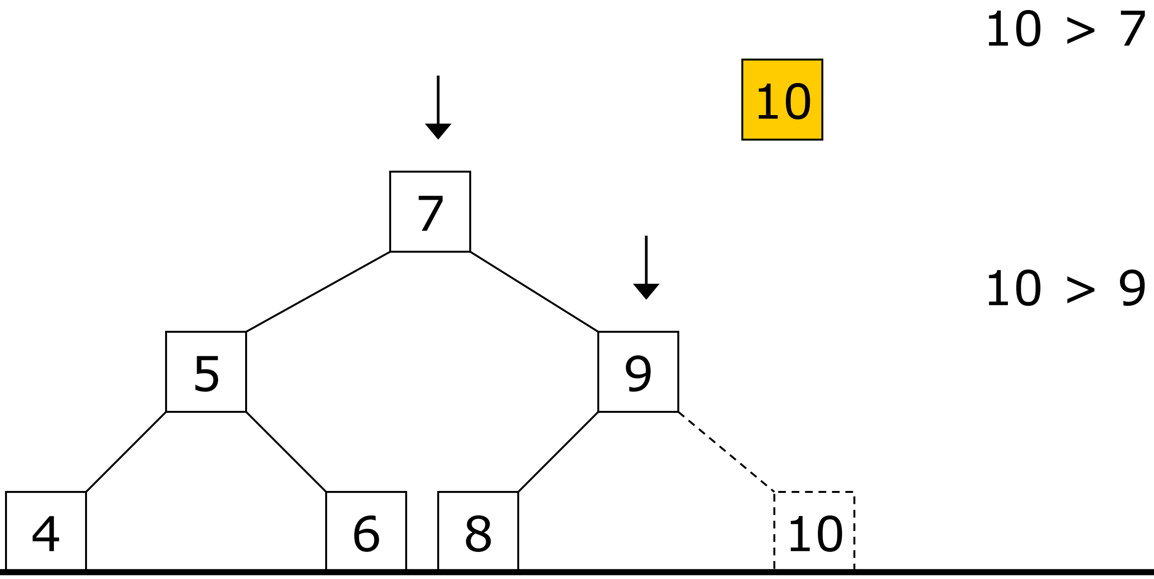
Insertion in BST - Example

Case 1: The Tree is Empty

- Set the root to a new node containing the key

Case 2: The Tree is Not Empty

- Call a recursive helper method to insert the key



Insertion in BST - Pseudocode

```
if  tree is empty
      create a root node with the new key
else
      compare key with the top node
      if key =  node key
             retuen the node
      else if  key >  node key
            compare key with the right subtree:
              if  subtree is empty create a leaf node
              else add key  in right subtree
       else  key <  node key
            compare key with the left subtree:
              if the subtree is empty create a leaf node
              else add key to the left subtree
```

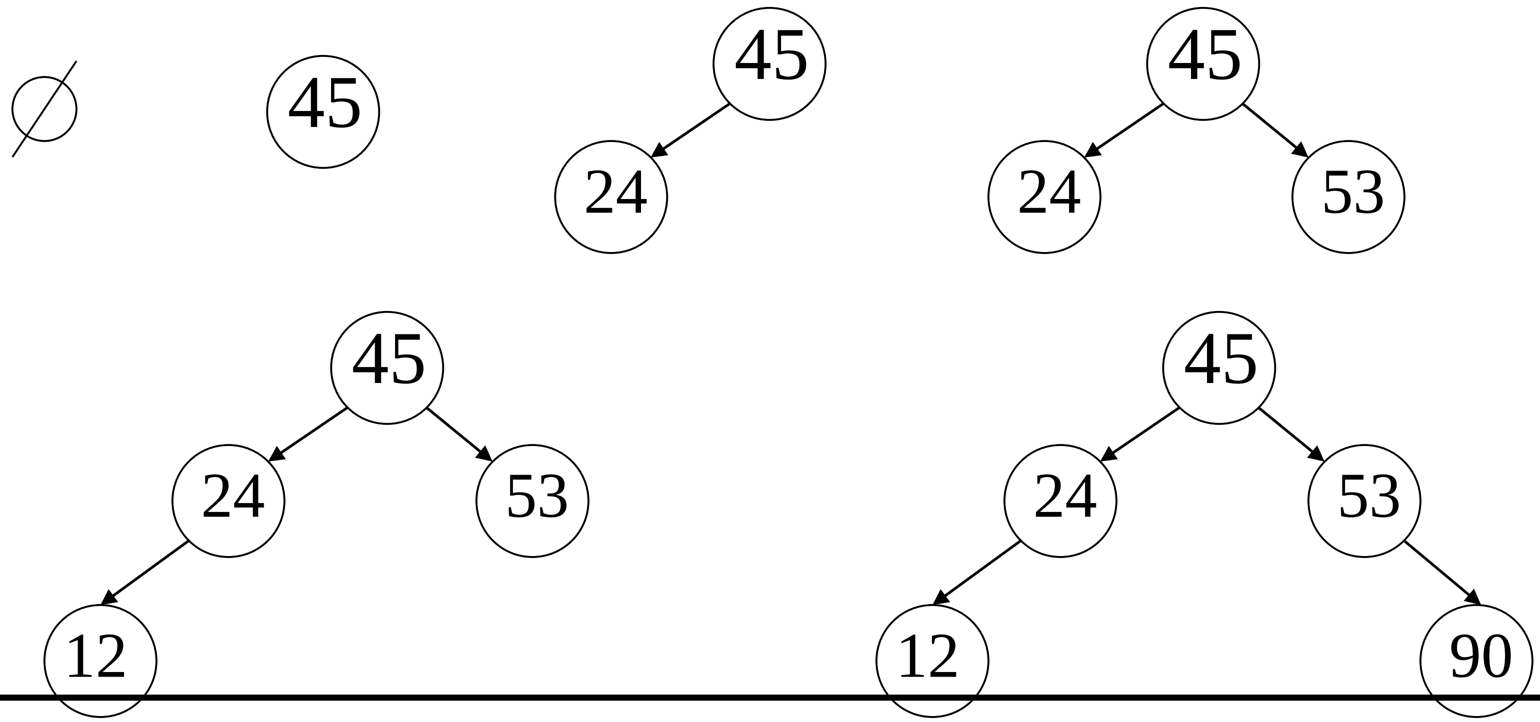
BST: Insertion

```
// Insert a record into the tree.
// k Key value of the record.
// e The record to insert.
void insert(const Key& k, const E& e) {
  root = inserthelp(root, k, e);
  nodecount++;
```

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::inserthelp(
    BSTNode<Key, E>* root, const Key& k, const E& it) {
  if (root == NULL)  // Empty tree: create node
    return new BSTNode<Key, E>(k, it, NULL, NULL);
  if (k < root->key())
    root->setLeft(inserthelp(root->left(), k, it));
  else root->setRight(inserthelp(root->right(), k, it));
  return root;       // Return tree with node inserted
}
```

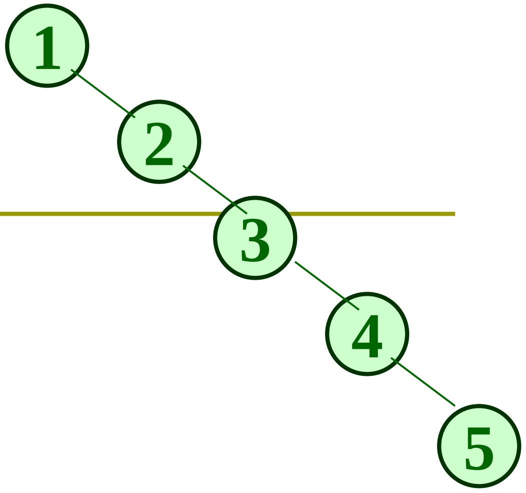
二叉查找树的生成(插入结点)

- 二叉查找树的生成(连续进行插入操作)
- 例如:对{45, 24, 53, 45, 12, 24, 90}
- 关键字序列的二叉查找树生成过程如下:



例如：

由关键字序列 **1， 2， 3， 4， 5**

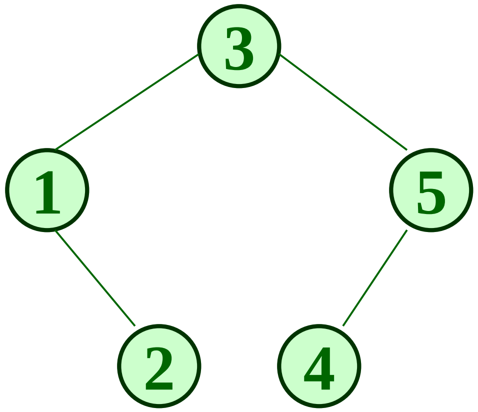


构造而得的二叉查找树，

ASL =（1+2+3+4+5）/ 5

= 3

由关键字序列 **3， 1， 2， 5， 4**



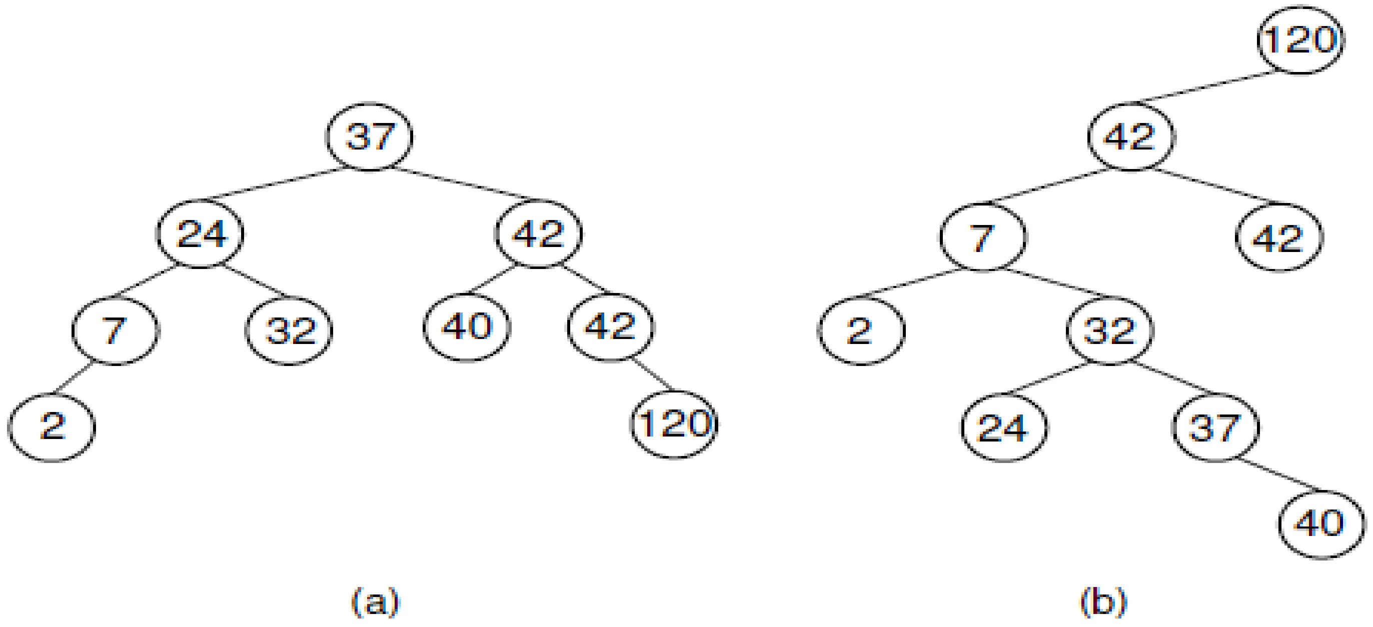
构造而得的二叉查找树，

ASL =（1+2+3+2+3）/ 5

= 2.2

BST Exercise

- (a) Build Binary Search Trees in the order 37, 24, 42, 7, 2, 40, 42, 32, 120.
- (b) Build Binary Search Trees in the order 120, 42, 42, 7, 2, 32, 37, 24, 40.



BST Operations: Removal

- **remove the smallest key value**

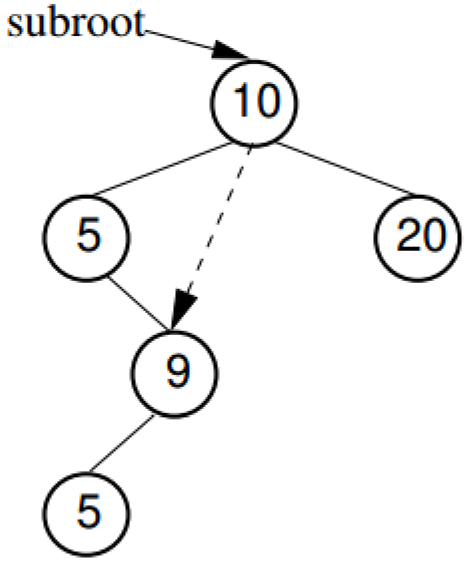
To remove the node with the minimum key value from a subtree

- **first find that node by continuously moving down the left link until there is no further left link to follow. Call this node *S*.**

- **To remove *S*, simply have the parent of *S* change its pointer to point to the right child of *S*.**

BST Operations: Removal

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
deletemin(BSTNode<Key, E>* rt) {
  if (rt->left() == NULL) // Found min
      return rt->right();
  else { // Continue left
    rt->setLeft(deletemin(rt->left()));
  return rt;
}
}
```



BST Operations: Removal

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
getmin(BSTNode<Key, E>* rt) {
  if (rt->left() == NULL)
    return rt;
  else return getmin(rt->left());
}
```

BST Operations: Removal

- **removes** a specified key from the BST and **adjusts** the tree
- uses a binary search to locate the target key:
  - **starting at the root** it probes down the tree till it finds the target or reaches a leaf node (target not in the tree)
- removal of a node must not leave a ‘gap’ in the tree,

Removal in BST - Pseudocode

```
method remove (key)
I  if the tree is empty return false
II      Attempt to locate the node containing the
   target key using the binary search  algorithm
   if the target is not found return false
   else the target is found, so remove its node:
     Case 1:  if the node has 2 empty subtrees
                replace the link in the parent with null
   Case 2:  if the node has a left and a right subtree
               - replace the node's value with the min
               value in the right subtree
               - delete the min node in the right subtree
```

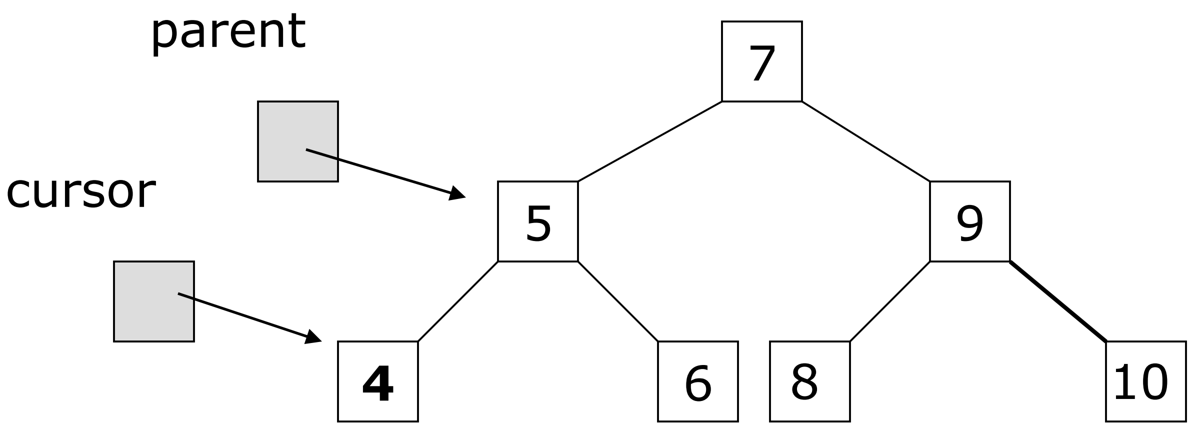
Removal in BST - Pseudocode

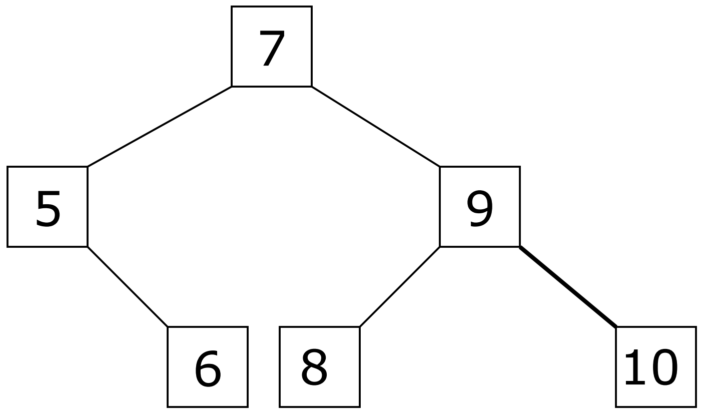
```
Case 3:  if  the node has no left child
    - link the parent of the node
      to the right (non-empty) subtree

Case 4:   if the node has no right child
    - link the parent of the target
      to the left (non-empty) subtree
```

Removal in BST: Example

Case 1: removing a node with 2 EMPTY SUBTREES





Removing 4
replace the link in the parent with **NULL**

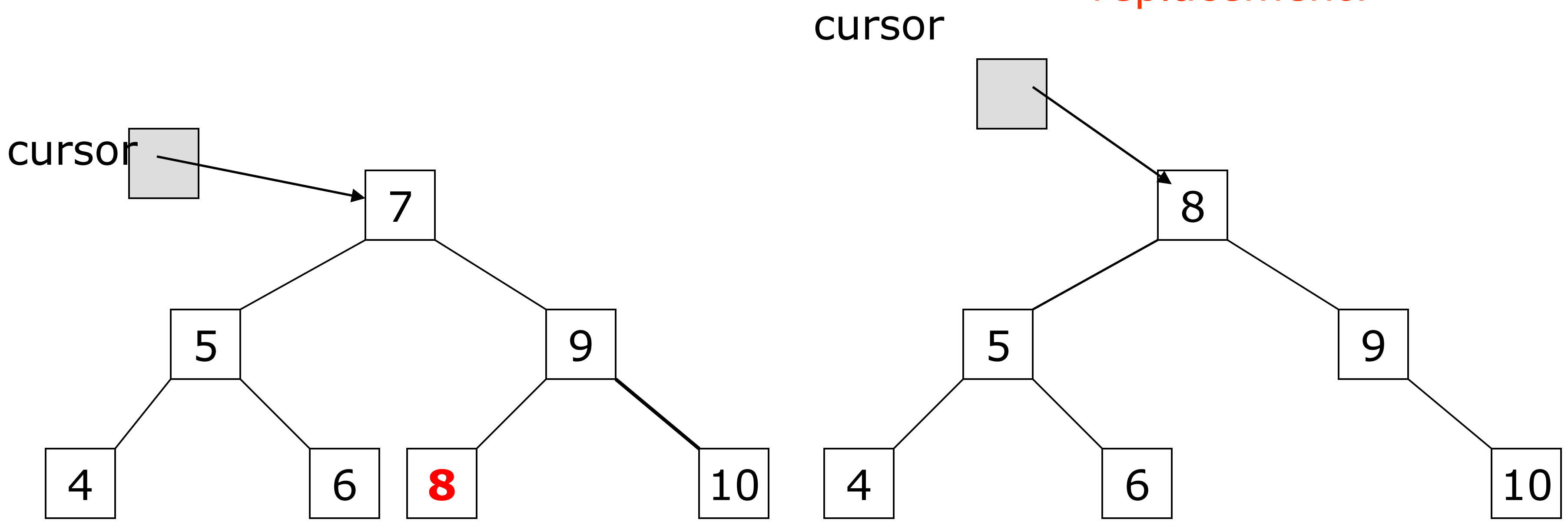
Removal in BST: Example

Case 2: removing a node with 2 SUBTREES

- replace the node's value with the min value in the right subtree
- delete the min node in the right subtree

What other element can be used as replacement?

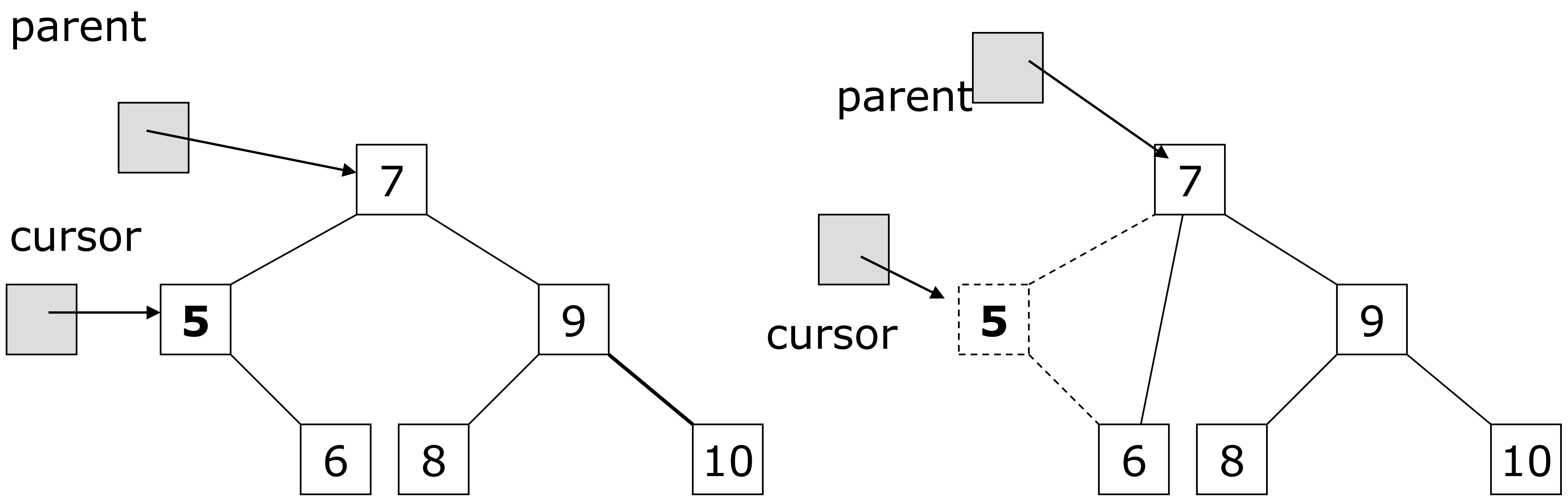
Removing 7



Removal in BST: Example

Case 3: removing a node with 1 EMPTY SUBTREE

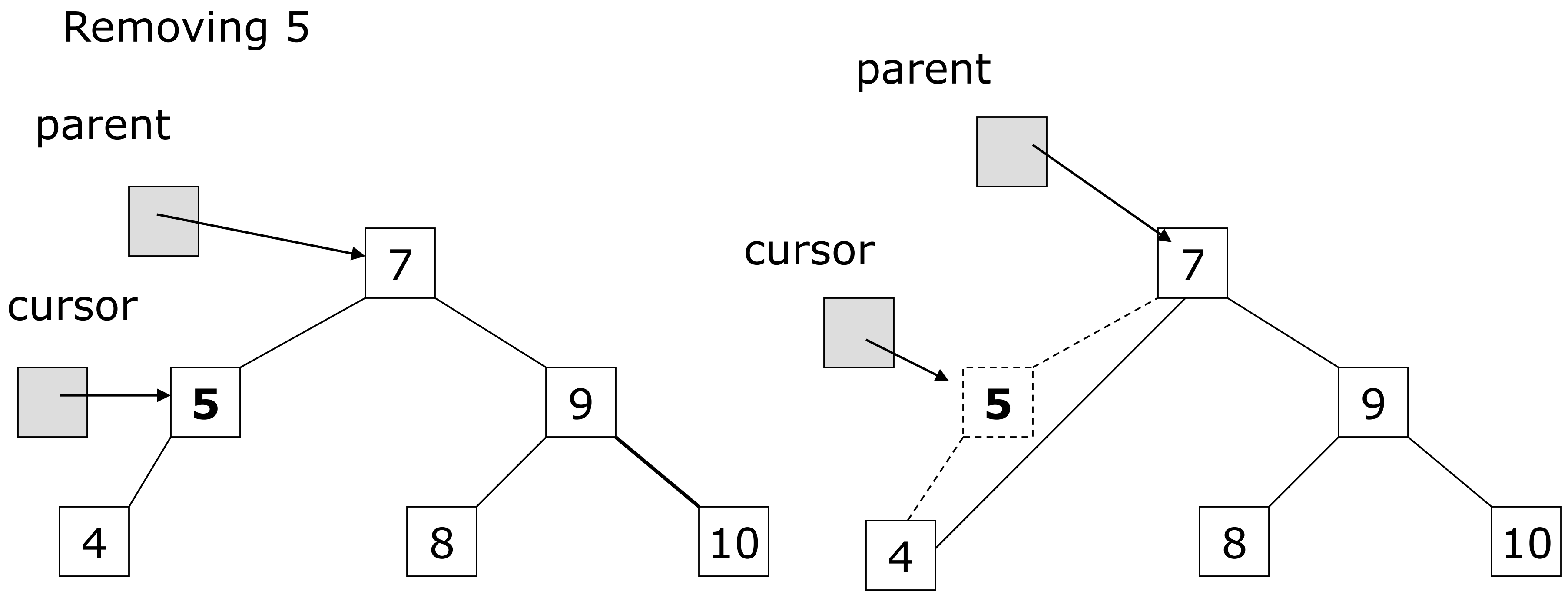
the node has no left child:
link the parent of the node to the right (non-empty) subtree



Removal in BST: Example

Case 4: removing a node with 1 EMPTY SUBTREE

the node has no right child:
link the parent of the node to the left (non-empty) subtree



Removal in BST: implementation

```
// Remove a node with key value k
// Return: The tree with the node removed
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
removehelp(BSTNode<Key, E>* rt, const Key& k) {
  if (rt == NULL) return NULL;    // k is not in tree
  else if (k < rt->key())
    rt->setLeft(removehelp(rt->left(), k));
  else if (k > rt->key())
    rt->setRight(removehelp(rt->right(), k));
  else {
    BSTNode<Key, E>* temp
    if (rt->left() == NULL
      rt = rt->right();
      delete temp;
    }
    else if (rt->right() =
      rt = rt->left();
      delete temp;
    }
    else {
      BSTNode<Key, E>* tem
      rt->setElement(temp-
      rt->setKey(temp->key());
      rt->setRight(deletemin(rt->right()));
      delete temp;
    }
  }
  return rt;
}
```

```
template <typename Key, typename E>
BSTNode<Key, E>* BST<Key, E>::
deletemin(BSTNode<Key, E>* rt) {
  if (rt->left() == NULL) // Found min
    return rt->right();
  else {                       // Continue left
    rt->setLeft(deletemin(rt->left()));
    return rt;
  }
}
```

BST Operations: clearhelp

- clearhelp, which returns the nodes of the BST to the freelist.
- Because the children of a node must be freed before the node itself, this is a postorder traversal.

```
template <typename Key, typename E>
void BST<Key, E>::
clearhelp(BSTNode<Key, E>* root) {
  if (root == NULL) return;
  clearhelp(root->left());
  clearhelp(root->right());
  delete root;
}
```

BST Operations: printhelp

- **printhelp, which performs an inorder traversal on the BST to print the node values in ascending order.**
- **Note that printhelp indents each line to indicate the depth of the corresponding node in the tree.**

```
template <typename Key, typename E>
void BST<Key, E>::
printhelp(BSTNode<Key, E>* root, int level) const {
  if (root == NULL) return;            // Empty tree
  printhelp(root->left(), level+1);    // Do left subtree
  for (int i=0; i<level; i++)          // Indent to level
    cout << "  ";
  cout << root->key() << "\n";         // Print node value
  printhelp(root->right(), level+1);   // Do right subtree
}
```

The ADT for a simple dictionary

- // The Dictionary abstract class.
- template <typename Key, typename E>
- class Dictionary {
- private:
- void operator =(const Dictionary&) {}
- Dictionary(const Dictionary&) {}
- public:
- Dictionary() {} // Default constructor
- virtual ˜Dictionary() {} // Base destructor
- // Reinitialize dictionary
- virtual void clear() = 0;

The ADT for a simple dictionary

- // Insert a record
- // k: The key for the record being inserted.
- // e: The record being inserted.
- virtual void insert(const Key& k, const E& e) = 0;
- // Remove and return a record.
- // k: The key of the record to be removed.
- // Return: A maching record. If multiple records match
- // "k", remove an arbitrary one. Return NULL if no record
- // with key "k" exists.
- virtual E remove(const Key& k) = 0;
- // Remove and return an arbitrary record from dictionary.
- // Return: The record removed, or NULL if none exists.
- virtual E removeAny() = 0;

The ADT for a simple dictionary

- // Return: A record matching "k" (NULL if none exists).
- // If multiple records match, return an arbitrary one.
- // k: The key of the record to find
- virtual E find(const Key& k) const = 0;
- // Return the number of records in the dictionary.
- virtual int size() = 0;
- };

Pointer-Based Node Implementation

```
// Simple binary tree node implementation
template <typename Key, typename E>
class BSTNode : public BinNode<E> {
private:
    Key k; // The node's key
    E it; // The node's value
    BSTNode* lc; // Pointer to left child
    BSTNode* rc; // Pointer to right child

public:
    // Two constructors -- with and without initial values
    BSTNode() { lc = rc = NULL; }
    BSTNode(Key K, E e, BSTNode* l =NULL, BSTNode* r =NULL)
        { k = K; it = e; lc = l; rc = r; }
    ~BSTNode() {} // Destructor
```

Pointer-Based Node Implementation

```
    // Functions to set and return the value and key
    E& element() { return it; }
    void setElement(const E& e) { it = e; }
    Key& key() { return k; }
    void setKey(const Key& K) { k = K; }

    // Functions to set and return the children
    inline BSTNode* left() const { return lc; }
    void setLeft(BinNode<E>* b) { lc = (BSTNode*)b; }
    inline BSTNode* right() const { return rc; }
    void setRight(BinNode<E>* b) { rc = (BSTNode*)b; }

    // Return true if it is a leaf, false otherwise
    bool isLeaf() { return (lc == NULL) && (rc == NULL); }
};
```

BST：Implementation

```
// Binary Search Tree implementation for the Dictionary ADT
template <typename Key, typename E>
class BST : public Dictionary<Key,E> {
private:
  BSTNode<Key,E>* root;   // Root of the BST
  int nodecount;         // Number of nodes in the BST

  // Private "helper" functions
  void clearhelp(BSTNode<Key, E>*);
  BSTNode<Key,E>* inserthelp(BSTNode<Key, E>*,
                          const Key&, const E&);
  BSTNode<Key,E>* deletemin(BSTNode<Key, E>*);
  BSTNode<Key,E>* getmin(BSTNode<Key, E>*);
  BSTNode<Key,E>* removehelp(BSTNode<Key, E>*, const Key&);
  E findhelp(BSTNode<Key, E>*, const Key&) const;
  void printhelp(BSTNode<Key, E>*, int) const;

public:
  BST() { root = NULL; nodecount = 0; }  // Constructor
  ~BST() { clearhelp(root); }            // Destructor

  void clear()   // Reinitialize tree
    { clearhelp(root); root = NULL; nodecount = 0; }
```

BST：Implementation

```
// Remove a record from the tree.
// k Key value of record to remove.
// Return: The record removed, or NULL if there is none.
E remove(const Key& k) {
  E temp = findhelp(root, k);    // First find it
  if (temp != NULL) {
    root = removehelp(root, k);
    nodecount--;
  }
  return temp;
}
```

BST：Implementation

```
// Remove and return the root node from the dictionary.
// Return: The record removed, null if tree is empty.
E removeAny() {  // Delete min value
  if (root != NULL) {
    E temp = root->element();
    root = removehelp(root, root->key());
    nodecount--;
    return temp;
  }
  else return NULL;
}
```

BST：Implementation

```
  // Return Record with key value k, NULL if none exist.
  // k: The key value to find. */
  // Return some record matching "k".
  // Return true if such exists, false otherwise. If
  // multiple records match "k", return an arbitrary one.
  E find(const Key& k) const { return findhelp(root, k); }

  // Return the number of records in the dictionary.
  int size() { return nodecount; }

  void print() const { // Print the contents of the BST
    if (root == NULL) cout << "The BST is empty.\n";
    else printhelp(root, 0);
  }
};
```

Analysis of BST Operations

- The complexity of operations `find`, `insert` and `remove` in BST is Θ(h) , where h is the height.
- $\Theta(\log n)$ when the tree is balanced. The updating operations cause the tree to become unbalanced.
- The tree can degenerate to a linear shape and the operations will become $\Theta(n)$
- **Traversing a BST costs Θ(*n*) regardless of the shape of the tree.**

Balanced Trees

- **BST has a high risk of becoming unbalanced, resulting in excessively expensive search and update operations.**
- **Solutions :**

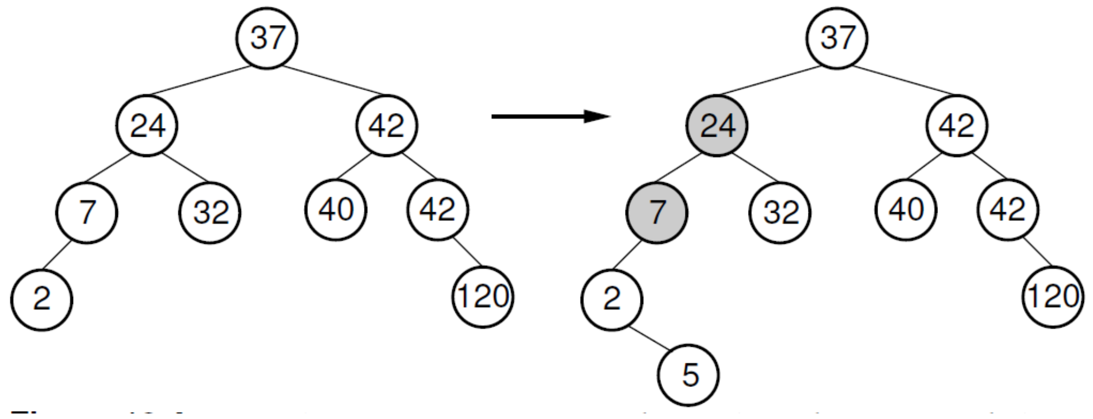
1. **to adopt another search tree structure such as the 2-3 tree or the binary trie.**
2. **to modify the BST access functions in some way to guarantee that the tree performs well.**
   - requiring that the BST always be in the shape of a complete binary tree requires excessive modification to the tree during update

- **If we are willing to weaken the balance requirements, we can come up with alternative update routines that perform well both in terms of cost for the update and in balance for the resulting tree structure, e.g., the AVL tree.**

The AVL tree

- The AVL tree (named for its inventors *Adelson-Velskii* and *Landis*) : a BST with the following additional property:
  - **For every node, the heights of its left and right subtrees differ by at most 1.**
- if a AVL tree contains n nodes, then it has a depth of at most $\Theta(\log n)$. As a result, search for any node will cost $\Theta(\log n)$ and if the updates can be done in time proportional to the depth of the node inserted or deleted, then updates will also cost $\Theta(\log n)$, even in the worst case.
- The key to making the AVL tree work is to alter the insert and delete routines so as to maintain the balance property.
  - **implement the revised update routines in $\Theta(\log n)$ time.**

Insertion in AVL tree: Example

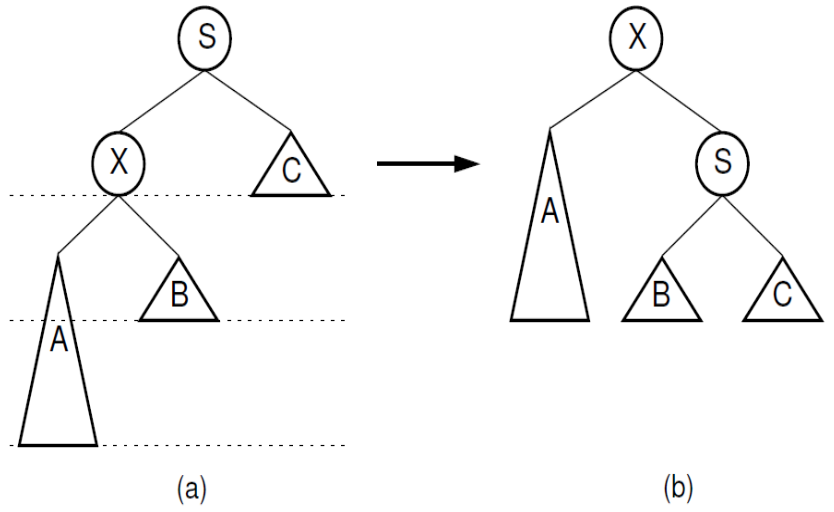


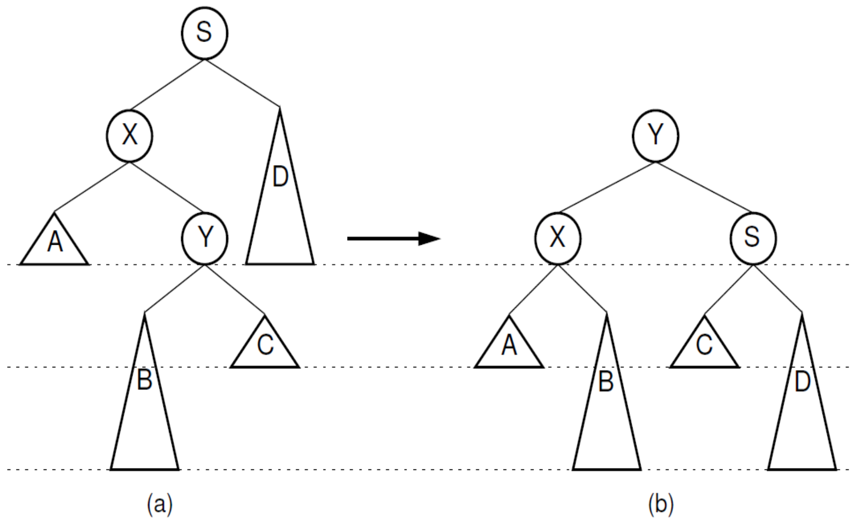
After inserting the node with value 5, the nodes with values 7 and 24 are no longer balanced.

For the bottommost unbalanced node, call it S, there are 4 cases:
1. The extra node is in the left child of the left child of S.
2. The extra node is in the right child of the left child of S.
3. The extra node is in the left child of the right child of S.
4. The extra node is in the right child of the right child of S.
Cases 1 and 4 are symmetrical, as are cases 2 and 3.
Note also that the unbalanced nodes must be on the path from the root to the newly inserted node

How to balance the tree in O(log n) time?

- using a series of local operations known as **rotations**





For case 1 and case 4:
single rotation

For case 2 and case 3:
double rotation

构造平衡二叉（查找）树的方法是：

在插入过程中，采用平衡旋转技术。

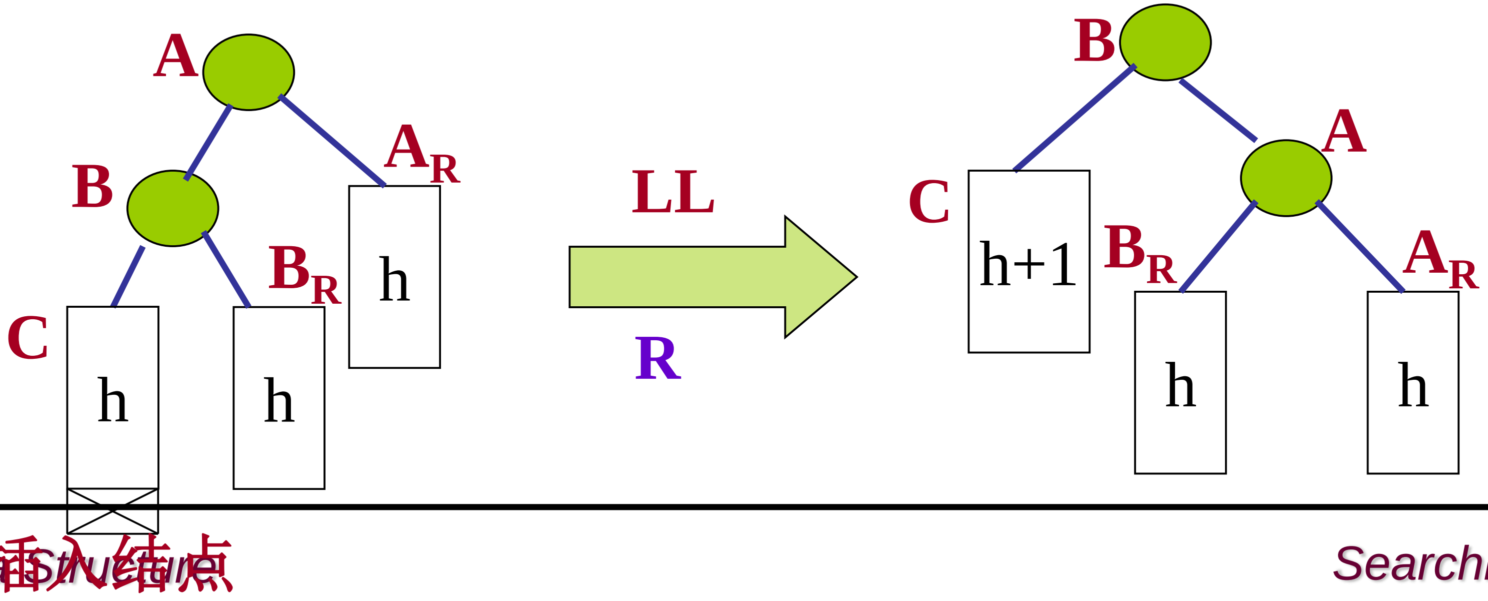
假设由于在二叉查找树上插入结点而失去平衡的最小子树根结点为A，沿插入路径取直接下两层的结点为B和C，如果这三个结点处于一条直线上，则采用单旋转进行平衡化；如果这三个结点处于一条折线上，则采用双旋转进行平衡化。失去平衡后进行调整的规律可归纳为下列4种情况：

（1）单向右旋平衡处理：（LLR）

原因：在A 的左子树根结点的左子树上插入结点，

A的平衡因子由1增至2，至使以A为根的子树失去平衡。

调整方法：以结点B为旋转轴，将结点A向右旋转成为B的右子树，结点B代替原来A的位置，原来B的右子树成为A的左子树。

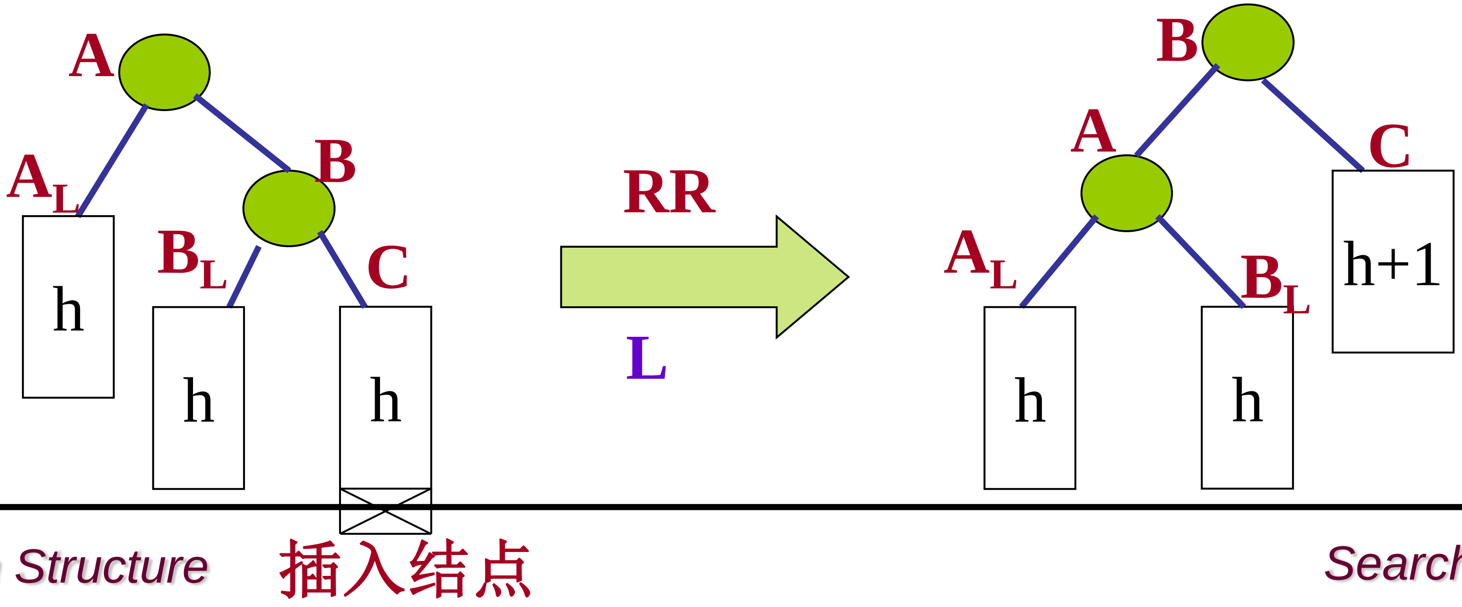


（2）单向左旋平衡处理：（RRL）

原因：在A 的右子树根结点的右子树上插入结点，

A的平衡因子由-1增至-2，至使以A为根的子树失去平衡。

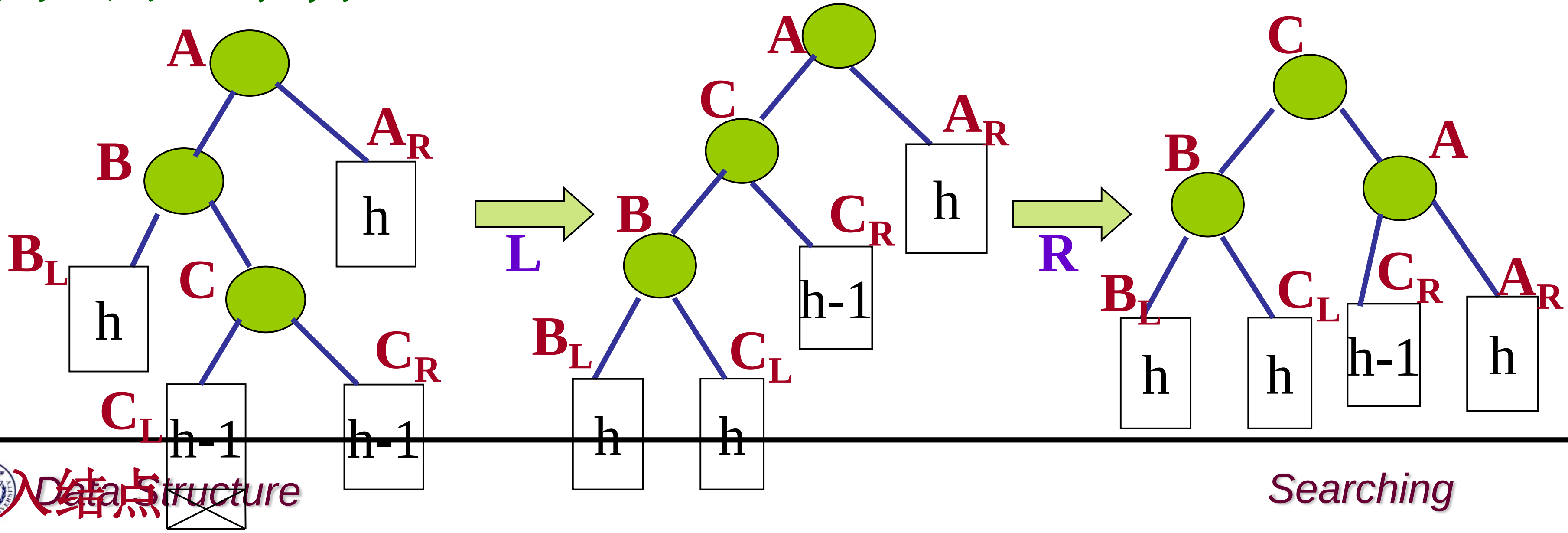
调整方法：以结点B为旋转轴，将结点A向左旋转成为B的左子树，结点B代替原来A的位置，原来B的左子树成为A的右子树。



（3）双向旋转（先左后右）平衡处理：（LRLR）

原因：在A 的左子树根结点的右子树上插入结点，A的平衡因子由1增至2，至使以A为根的子树失去平衡。

调整方法：先以C为旋转轴，将结点B向左旋转成为C的左子树，结点C代替原来B的位置，原来C的左子树成为B的右子树。再以C为旋转轴，将A向右旋转成为C的右子树，结点C代替原来A的位置，原来C的右子树成为B的左子树。

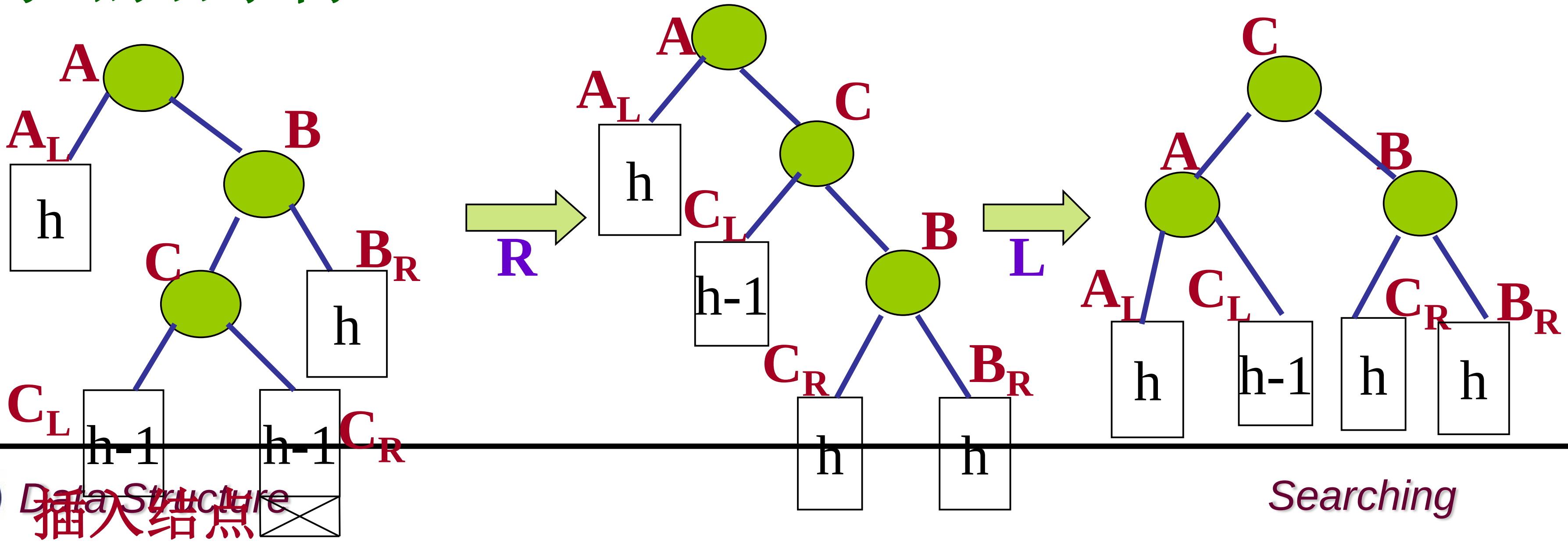


（4）双向旋转（先右后左）平衡处理：（RLRL）

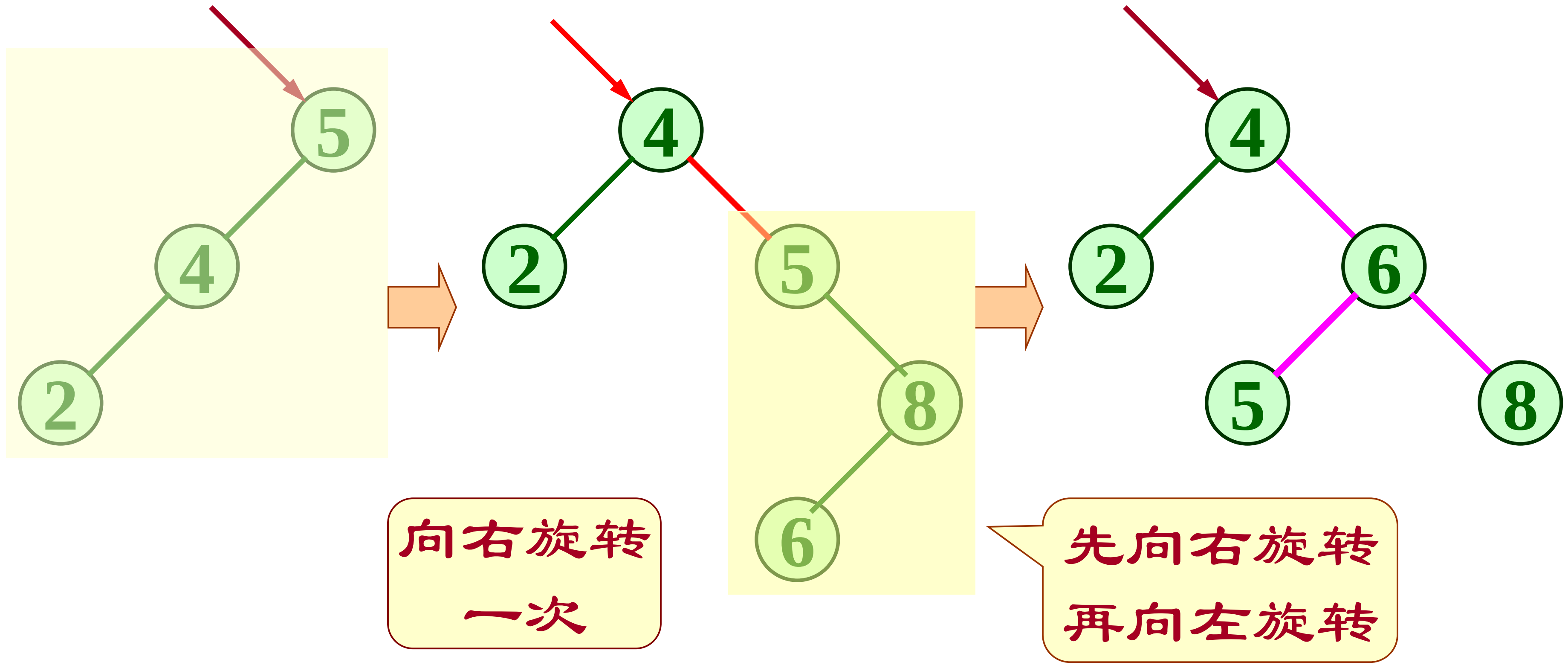
原因：在A 的右子树根结点的左子树上插入结点，

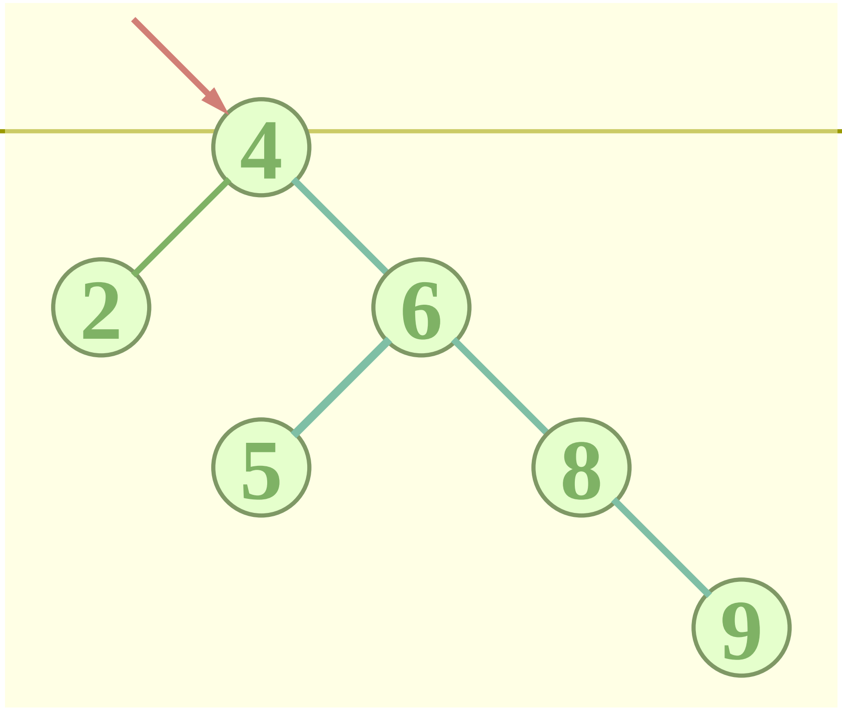
A的平衡因子由-1增至-2，至使以A为根的子树失去平衡。

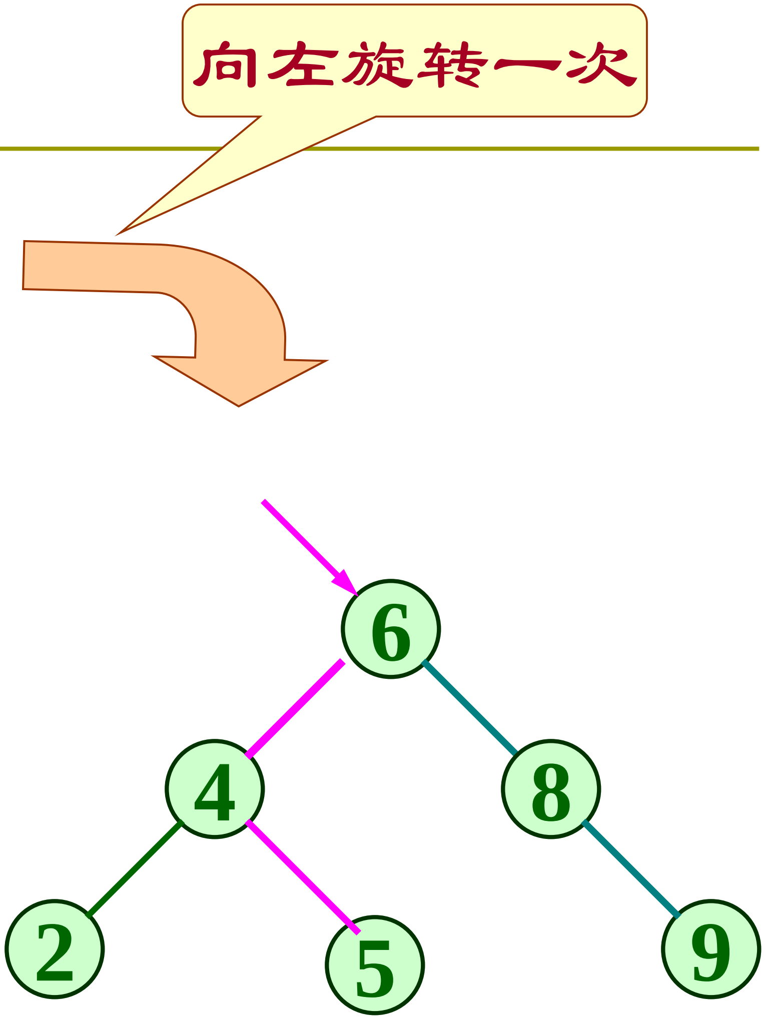
调整方法：先以C为旋转轴，将结点B向右旋转成为C的右子树，结点C代替原来B的位置，原来C的右子树成为B的左子树。再以C为旋转轴，将A向左旋转成为C的左子树，结点C代替原来A的位置，原来C的左子树成为B的右子树。



例如:依次插入的关键字为5, 4, 2, 8, 6, 9







继续插入关键字 9

Operations in AVL tree

- **Insertion algorithm:**

1. **begin with a normal BST insert**
2. **Then as the recursion unwinds up the tree, perform the appropriate rotation on any node that is found to be unbalanced.**

- **Deletion is similar**
  - consideration for unbalanced nodes must begin at the level of the deletemin operation.

References

- **Data Structures and Algorithm Analysis Edition 3.2 (C++ Version)**
  - P.168-185
  - P.442-445