Module04 - C++ 标准库

标准库

- 标准库 (C++ Standard Library)
  本单元将通过以下几个方面的内容的介绍，学习并使用 C++ 标准库：
  - 常用数据结构简介 (Data Structures)
  - 标准容器 (STL Containers)
  - 常用算法介绍 (About Algorithms)
  - 算法与函数对象 (STL Algorithms & Functional Objects)
  - 迭代器 (Iterators)
  - 字符串 (String)
  - I/O 流 (Stream)
  - 数值 (Numeric)

Module04-01
C++ 标准库：数据结构简介

- 数据结构简介
- 标准容器
- 常用算法简介
- 标准算法与函数对象
- 迭代器
- 字符串
- I/O 流
- 数值

标准库 - 数据结构简介

- 常用数据结构 (Data Structures) 部分将介绍 C++ 标准库中常用的几种数据结构类型：
  - Dynamic Array
  - Linked List
  - Binary Search Tree
  - Red-black Tree
  - Hash Table
  - Stack & Queue

数据结构简介 - 概述

轩辕IT培训

- 常用的数据结构：
  - 标量 (Scalar)（单个基本类型的数据或 ADT 类型的对象）
  - 线性存储数据结构 (Sequence)：
    - 定长数组 (Array)、动态数组 (Dynamic Array)、链表 (Linked List)、栈 (Stack)、队列 (Queue)
  - 哈希表 (Hash Table)
  - 树结构 (Tree)
    - 二叉树：二叉搜索树 (BST)、自平衡二叉搜索树 (AVL)、红黑树 (Red-Black Tree)
    - 多叉树
  - 图结构 (Graph)

数据结构简介 - 参考书目

- 本单元参考书目：
  - Introduction to Algorithms, Second Edition
    作者： Thomas H. Cormen, Charles E. Leiserson
    Ronald L. Rivest and Clifford Stein

数据结构简介 - Dynamic Array

轩辕IT培训

- 数据结构简介
  - Dynamic Array
  - Linked List
  - Binary Search Tree
  - Red-Black Tree
  - Hash Table
  - Stack
  - Queue

数据结构简介 - Dynamic Array

- 关于 Dynamic Array - 动态数组
  - 能动态增长的 Array，如 C++ STL 中的 vector 容器
  - 当动态数组元素数目达到当前容量后，任何添加元素的操作将导致：
    - 重新分配 2 倍于原数组容量的空间，
    - 然后将原先数组中的元素复制到新空间，
    - 最后删除原数组的空间
  - 动态数组操作的时间复杂度
    - 访问元素的时间复杂度： O(1)
    - 在尾部增加元素：有需重新分配空间和不需重新分配空间两种情况，平均时间复杂度大约 O(1)
    - 在尾部删除元素的时间复杂度： O(1)
    - 在尾部以外位置删除、添加元素的时间复杂度： O(n)，即线性时间

数据结构简介 - Linked List

- 数据结构简介
  - Dynamic Array
  - Linked List
  - Binary Search Tree
  - Red-Black Tree
  - Hash Table
  - Stack & Queue

数据结构简介 - Linked List

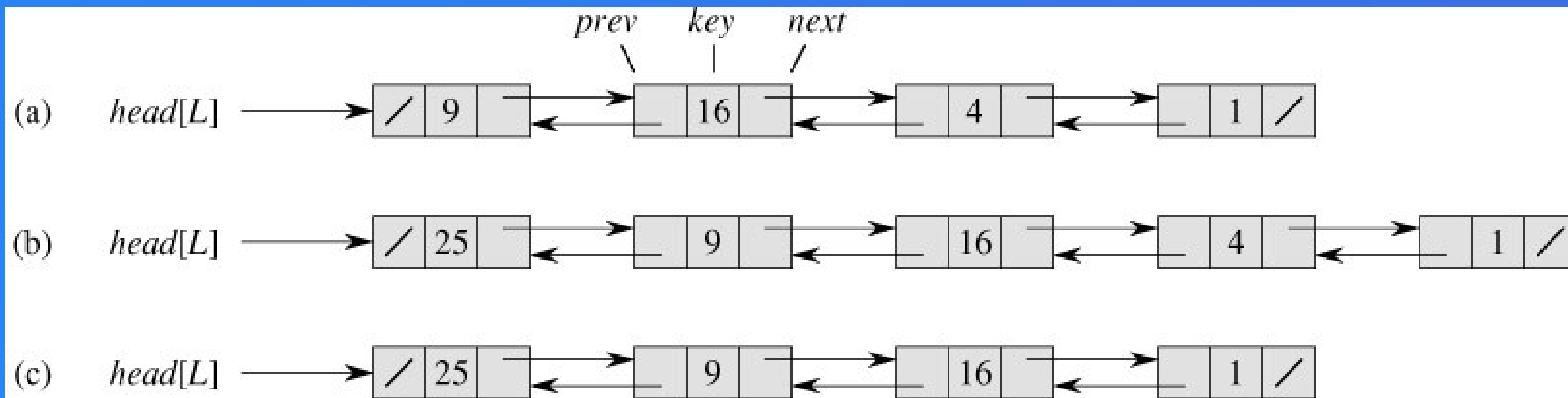
轩辕IT培训

- 关于 Linked List - 链表
  - 如同 Array，链表也是一种元素以线性存储的数据结构，但与 Array 不同的是，链表元素不能以下标方式直接访问，同时链表中的元素在内存中的存储不一定是连续的。
  - 链表有单链表和双链表：
    - 单链表中的每个元素拥有一个指向其下一个元素的指针；
    - 双链表中的每个元素除了有一个指向下一个元素的指针外，还有一个指向其前一个元素的指针。
      双链表有两种形式：环式双链表和开放式双链表。

以下仅讨论双链表。

数据结构简介 - Linked List

- Doubly Linked List
  - 下图所示为开放式双链表



- 双链表的存储单元为节点 (Node)，每个 Node 包含一个 key(存储元素的值)、一个 prev 指针指向其前一个节点、一个 next 指针指向下一个节点
- 每个链表有一个头部， head.next 指向链表的实际上的第一个元素
- 链表最后一个元素 tail， tail.next 指向 NULL，标志链表的结束
- 上图 (b)：往链表中添加一个元素 25
- 上图 (c)：从链表中删除一个元素 4

数据结构简介 - Linked List

轩辕IT培训

- Doubly Linked List - 开放式双链表操作 – 查找元素
  - 以下假设有链表对象： ls
  - 伪码：

```
LIST-SEARCH(k)
    node* x = ls.head
    while x ≠ NULL and x->key ≠ k
            do x = x->next
    return x
```

  - 如果不存在所要查找的元素，返回 NULL
  - 在有 n 个元素的链表中查找元素，该操作最坏情况下的时间复杂度： Θ(n)，因为要查找的元素可能位于链表末端

数据结构简介 - Linked List

- Doubly Linked List - 开放式双链表操作 – 在表头插入节点
  - 以下假设有链表对象： ls
  - 伪码：

```
LIST-INSERT(x)
  x->next = ls.head
  if ls.head ≠ NULL
      then ls.head->prev = x
  ls.head = x
  ls.prev = NULL
```

  - 该操作时间复杂度： O(1)

数据结构简介 - Linked List

- Doubly Linked List - 开放式链表操作 – 在表中插入节点
  - 以下假设有链表对象： ls
  - 伪码：

```
LIST-INSERT(x, y) //  将 x 插入到 y 之前
  x->prev = y->prev
  x->nex = y
  y->prev->next = x
  y->prev = x
```

  - 该操作时间复杂度： O(1)

数据结构简介 - Linked List

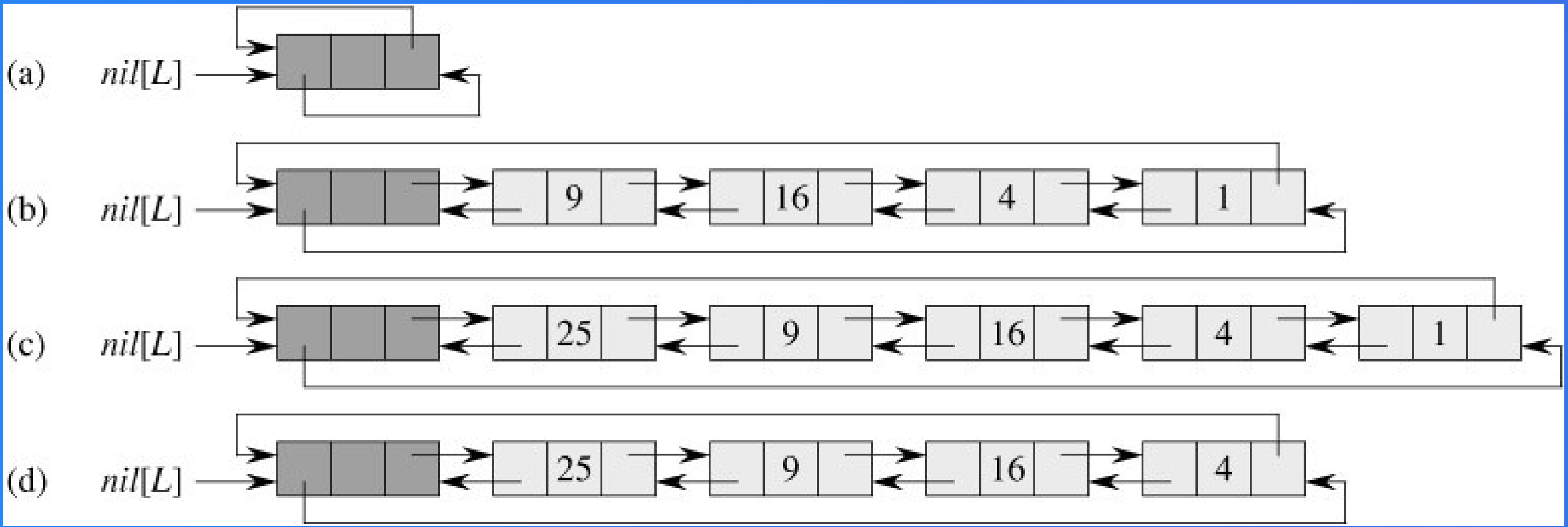
- Doubly Linked List - 开放式链表操作操作 – 删除节点
  - 以下假设有链表对象： ls
  - 伪码：

```
LIST-DELETE(x)
if x->prev ≠ NULL
     then x->prev->next = x->next
     else ls.head = x->next
if x->next ≠ NULL
     then x->next->prev = x->prev
```

  - 该操作时间复杂度： O(1)

数据结构简介 - Linked List

- Doubly Linked List - 环式链表
  - 下图所示为环式双链表



- 与开放式双链表相比，环式双链表多了一个哨位节点 (Sentinel Node)，其 prev 指向链表的最后一个元素， next 指向链表的第一个元素，而链表第一个实际节点的 prev 指向哨位节点、链表最后一个节点的 next 指向哨位节点，从而形成一个首尾相连的环状链表

数据结构简介 - Linked List

- Doubly Linked List - 环式链表操作 - 查找元素
  - 假设链表对象为： ls
  - 伪码：

```
LIST-SEARC' (k)
  x = ls.nil->next
  while x ≠ ls.nil and x->key ≠ k
        do x = x->next
  return x
```

  - 该操作的时间复杂度：最坏情况下为 $\Theta(n)$

数据结构简介 - Linked List

轩辕IT培训

- Doubly Linked List - 环式链表操作 - 删除节点
  - 假设链表对象为： ls
  - 伪码：

```
LIST-DELET' (x)
1   x->prev->next = x->next
2   x->nex-prev = x->prev
```

  - 该操作的时间复杂度： O(1)

数据结构简介 - Linked List

- Doubly Linked List - 环式链表操作 - 表头部插入节点
  - 假设链表对象为： ls
  - 伪码：

```
LIST-INSER'  (x)
  x->next = ls.nil->next
  ls.nil->next->prev = x
  ls.nil->next = x
  x->prev = ls->nil
```

  - 该操作的时间复杂度： O(1)

数据结构简介 - Linked List

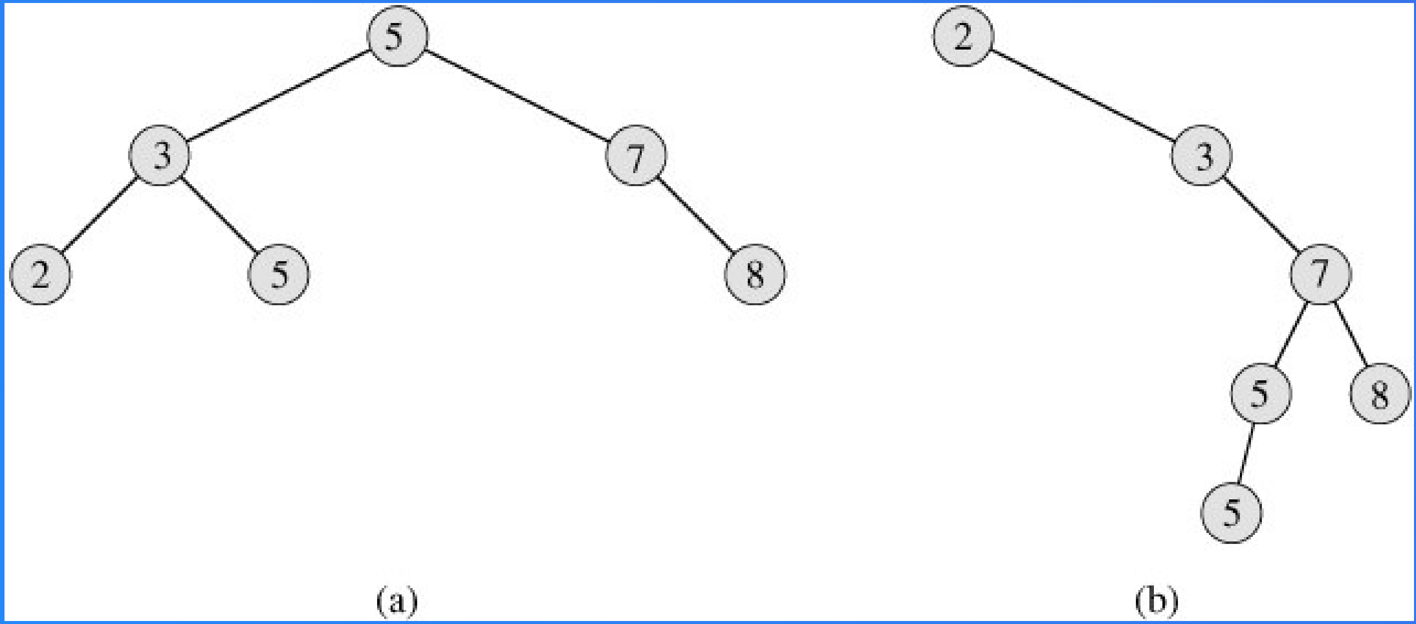
- Linked List - 小结
  - 优点：
    - 链表的 insert 、 delete 操作具有很好的时间复杂度： O(1)
    - 对于排序、合并等操作，链表的开销通常比其它数据结构如 Array 、 Tree 等小
  - 缺点：
    - 链表访问、查找的效率较低，最坏情况下为 Θ(n)
    - 与 Array 相比，需为每个元素额外分配 1 个（单链表）或 2 个（双链表）指针的存储

数据结构简介 - Binary Search Tree

- 数据结构简介
  - Dynamic Array
  - Linked List
  - Binary Search Tree
  - Red-Black Tree
  - Hash Table
  - Stack & Queue

数据结构简介 - Binary Search Tree

- 关于 Binary Search Tree - 二叉搜索树
  - 如下图所示



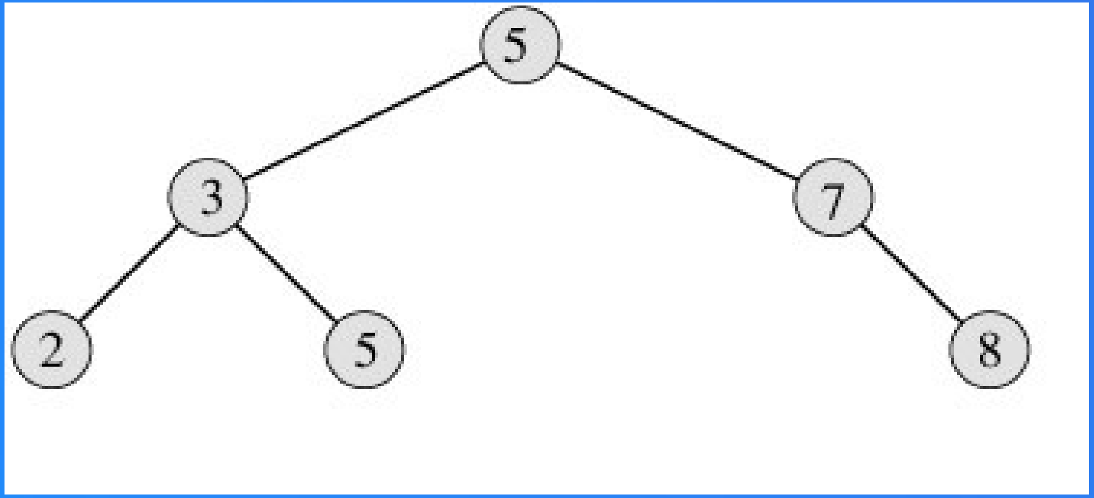
数据结构简介 - Binary Search Tree

- 关于 Binary Search Tree - 二叉搜索树（续）
  - 二叉搜索树的特性
    - 树的存储单元为节点 Node，每个 Node 包含 4 个部分：
      - key：所容纳的元素
      - parent：指向父节点的指针
      - left：指向左子树的指针
      - right：指向右子树的指针
    - 每个节点的左子树的所有节点的 key 小于或等于其父节点的 key 值，右子树的所有节点的 key 大于或等于其父节点的 key 值
    - 除树的根节点 (root) 的 parent 为 NULL 外，其它节点的 parent 不可为 NULL
    - 一个节点的可以有或没有左、右子树（即左、右节点为 NULL ）

数据结构简介 - Binary Search Tree

- 遍历二叉搜索树
  - 深度优先遍历二叉搜索树：
    - 中序遍历 (In-Order Walk)：
      - 左子树 → 根节点 → 右子树
    - 先序遍历 (Pre-Order Walk)：
      - 根节点 → 左子树 → 右子树
    - 后序遍历 (Post-Order Walk)：
      - 左子树 → 右子树 → 根节点
  - 广度优先遍历二叉树
    - 即按层次遍历二叉树

- 遍历二叉搜索树（续）
  - 示例说明，以下图 (a) 二叉树为例
    - 中序遍历： 2, 3, 5, 5, 7, 8 （如果以迭代器方式访问树节点，则是按中序遍历的方式）
    - 先序遍历： 5, 3, 2, 5, 7, 8
    - 后序遍历： 2, 5, 3, 8, 7, 5
    - 按层次遍历： 5, 3, 7, 2, 5, 8



数据结构简介 - Binary Search Tree

- 遍历二叉搜索树 - 伪码

```
INORDER-TREE-WALK(x)
if x ≠ NULL
    then INORDER-TREE-WALK(x->left)
         print x->key
         INORDER-TREE-WALK(x->right)
```

```
PREORDER-TREE-WALK(x)
if x ≠ NULL
    then print x->key
         INORDER-TREE-WALK(x->left)
         INORDER-TREE-WALK(x->right)
```

```
POSTORDER-TREE-WALK(x)
if x ≠ NULL
    then INORDER-TREE-WALK(x->left)
         INORDER-TREE-WALK(x->right)
         print x->key
```

数据结构简介 - Binary Search Tree

- 二叉搜索树操作 - 查找元素
  - 伪码：递归方式

```
TREE-SEARCH (x, k)   // x 为节点, k 为要查找的值
if x == NULL or k == x->key
    then return x
if k < x->key
    then return TREE-SEARCH(x->left, k)
    else return TREE-SEARCH(x->right, k)
```

  - 伪码：迭代方式

```
ITERATIVE-TREE-SEARCH(x, k)   // x 为节点, k 为要查找的值
while x ≠ NULL and k ≠ x->key
     do if k < x->key
              then x = x->left
              else x = x->right
return x
```

  - 该操作的时间复杂度： O(h) h 为子树 x 的高度

数据结构简介 - Binary Search Tree

- 二叉搜索树操作 - 找最小节点和最大节点
  - 说明：某些时候查找一个子树的最左或最右的节点操作是必须的，如 C++ 实现二叉树的迭代器， begin() 和 end() 操作即依赖于上述操作
  - 伪码：找某节点（子树）中最小（最左）节点

```
TREE-MINIMUM (x) //  任意节点
    while x->left ≠ NULL
        do x = x->left
    return x
```

  - 伪码：找某节点（子树）中最大（最右）节点

```
TREE-MAXIMUM (x) //  任意节点
    while x->right ≠ NULL
        do x = x->right
    return x
```

  - 该操作的时间复杂度： O(h) h 为子树 x 的高度

- 二叉搜索树操作 - 找下一个节点
  - 说明：在某些时候确定一个节点的前一个节点或下一个节点十分重要，比如 C++ 针对二叉树实现的迭代器中， ++ 和 -- 操作就是依赖于上述操作
  - 伪码：查找某节点的下一个节点

```
TREE-SUCCESSOR(x)
if x->right ≠ NULL
    then return TREE-MINIMUM (x->right)
y = x->parent
while y ≠ NULL and x == y->right
    do x = y
       y = y->parent
return y
```

  - 该操作的时间复杂度： O(h) h 为子树 x 的高度

注：以上操作按中序遍历的方式(Io-Order Walk)

数据结构简介 - Binary Search Tree

- 二叉搜索树操作 - 找前一个节点
  - 伪码：查找某节点的前一个节点

```
TREE-PREDECESSOR(x)
  if x->left ≠ NULL
      then return TREE-MAXIMUM (x->left)
  y = x->parent
  while y ≠ NULL and x == y->left
      do x = y
         y = y->parent
  return y
```

  - 该操作的时间复杂度： O(h) h 为子树 x 的高度

注：以上操作按中序遍历的方式(Io-Order Walk)

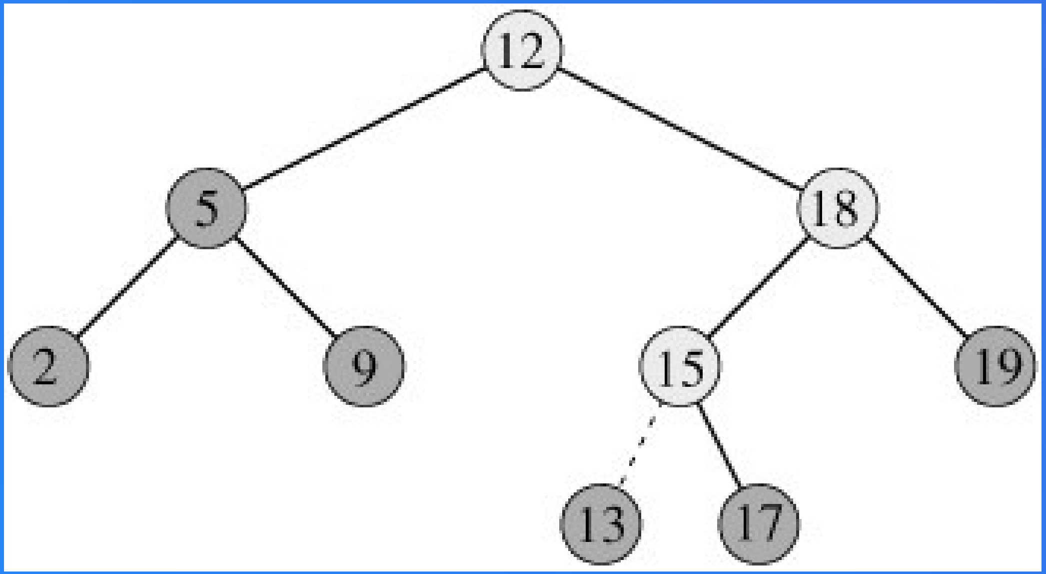
数据结构简介 - Binary Search Tree

- 二叉搜索树操作 - 插入节点
  - 伪码：

```
TREE-INSERT(z)
  y = NULL
  x = tree->root
  while x ≠ NULL
      do y = x
         if z->key < x->key
            then x = x->left
            else x = x->right
  z->parent = y
  if y == NULL
     then tree->root = z // tree was empty
     else if z->key < y->key
             then y->left = z
             else y->right = z
```

数据结构简介 - Binary Search Tree

- 二叉搜索树操作 - 插入节点（续）
  - 图示：

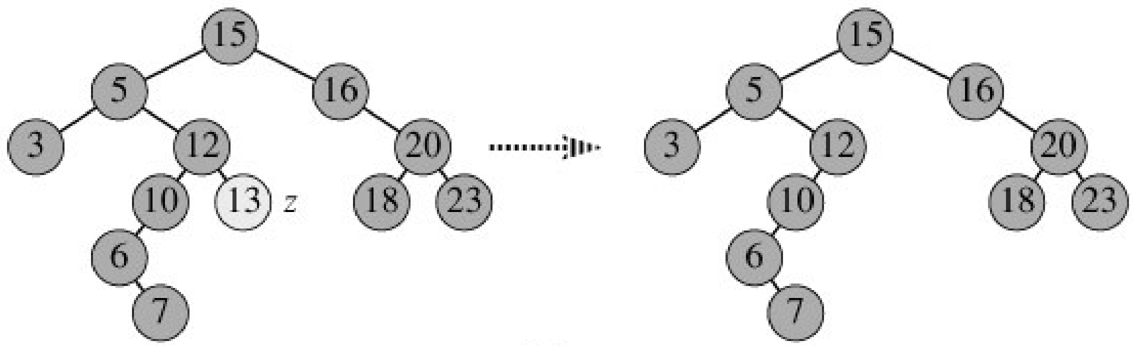


- 二叉搜索树操作 - 删除节点
  - 伪码:

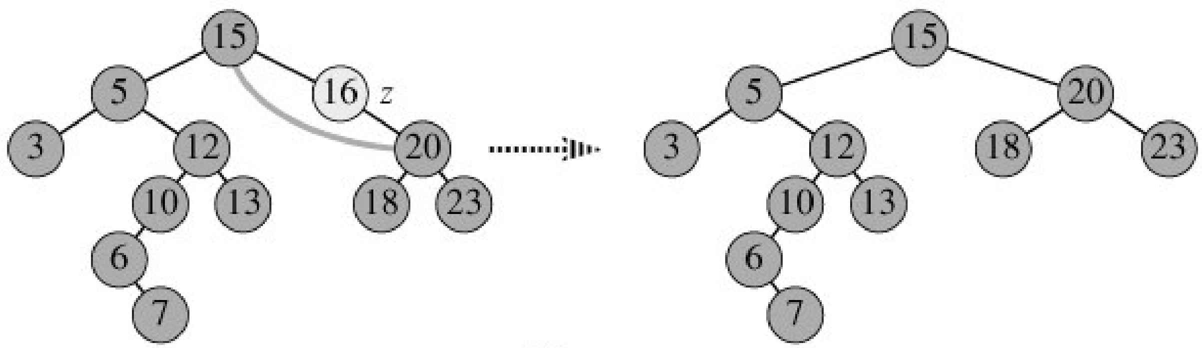
```
TREE-DELETE(z)
if z->left == NULL or z->right == NULL
    then y = z
    else y = TREE-SUCCESSOR(z)
if y->left ≠ NULL
    then x = y->left
    else x = y->right
if x ≠ NULL
    then x->parent = y->parent
if y->parent == NULL
    then tree->root = x
    else if y == y->parent->left
            then y->parent->left = x
            else y->parent->right = x
if y ≠ z
    then z->key = y->key
         copy y's satellite data into z
return y
```

数据结构简介 - Binary Search Tree

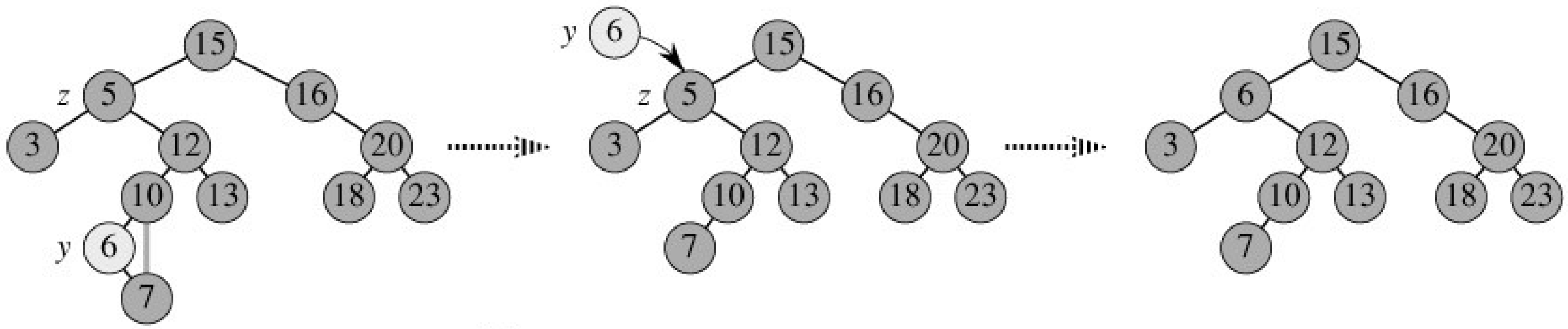
轩辕IT培训



(a)



(b)



(c)

- 二叉搜索树操作 - 删除节点（续）
  - 删除和插入节点操作的时间复杂度:
    - O(h) h 为树的高度

- 二叉搜索树 - 小结
  - 优点：
    - 搜索、插入、删除操作有较好的时间复杂度： O(h) ，另一种表示方式： O(lg n) n 为树的节点数
    - 保持已序
  - 缺点：
    - 由于二叉搜索树不一定都是平衡树，如果树的高度等于树的节点数，则退化为链表形式

数据结构简介 - Red-Black Tree

- 数据结构简介
  - Dynamic Array
  - Linked List
  - Binary Search Tree
  - Red-Black Tree
  - Hash Table
  - Stack & Queue

数据结构简介 - Red-Black Tree

- 关于 Red-black Tree 红黑树
  - 本质上讲，红黑树是一种近似平衡二叉搜索树。
  - 由于为每个节点赋予了颜色：红色或黑色的性质，故称红黑树
  - 同二叉搜索树，红黑树的存储单元也是 Node ，包含 5 个部分：
    - color ： Red/Black
    - key
    - parent ： 指向父节点的指针
    - left ： 指向左子树的指针
    - right ： 指向右子树的指针

数据结构简介 - Red-Black Tree

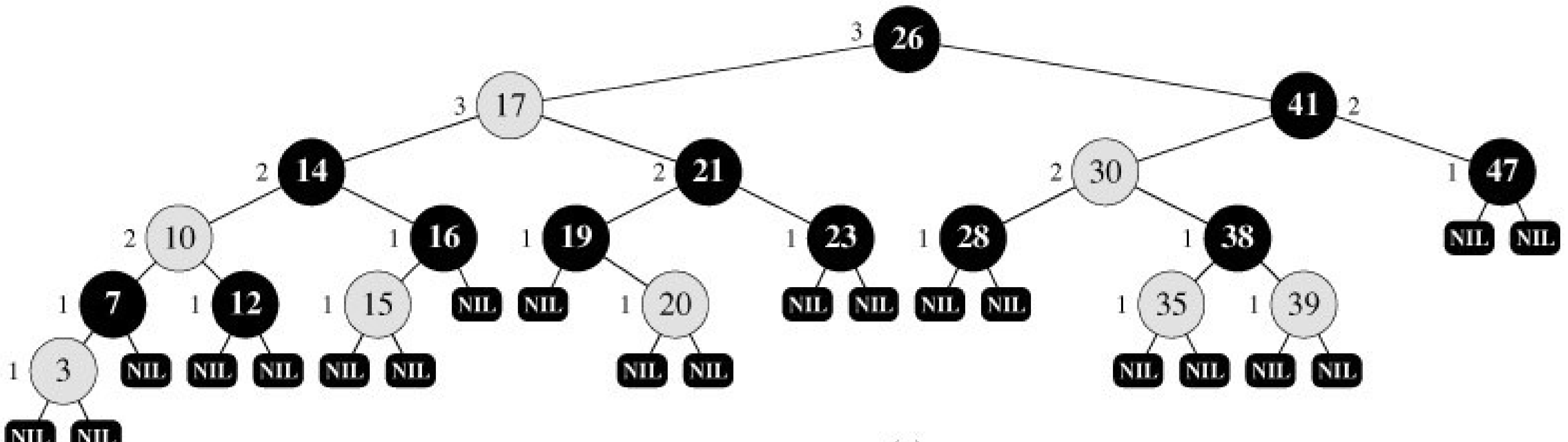
轩辕IT培训

- Red-black Tree 性质
  - 如果一个二叉搜索树符合以下性质，即为红黑树：
    - 所有节点非红即黑
    - 根节点为黑
    - 叶子 (NIL) 为黑
    - 如果一个节点为红，则其两个子节点均为黑
    - 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点
  - 由上述性质推导出：
    - 一个有 n 个节点的红黑树，其高度最大不超过 2*lg(n+1)
    - 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长

数据结构简介 - Red-Black Tree

轩辕IT培训

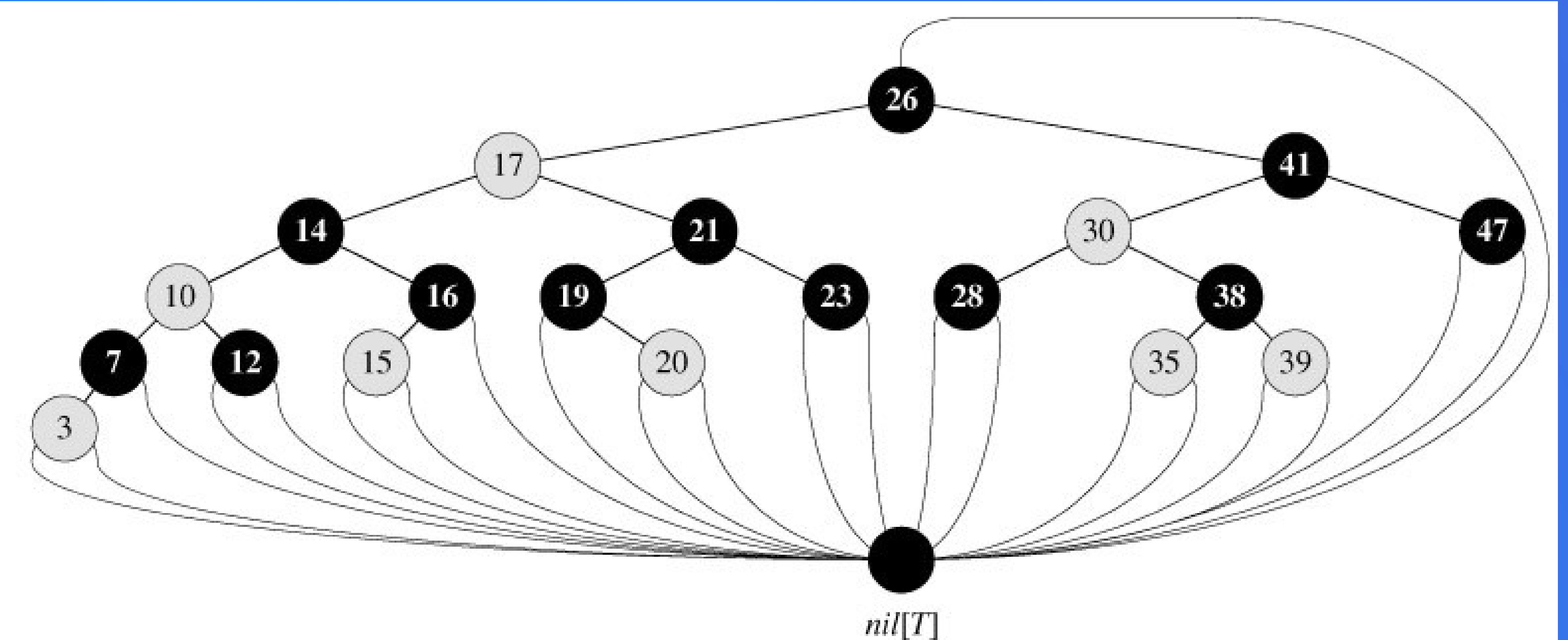
- Red-black Tree 图示 1



(a)

轩辕IT培训

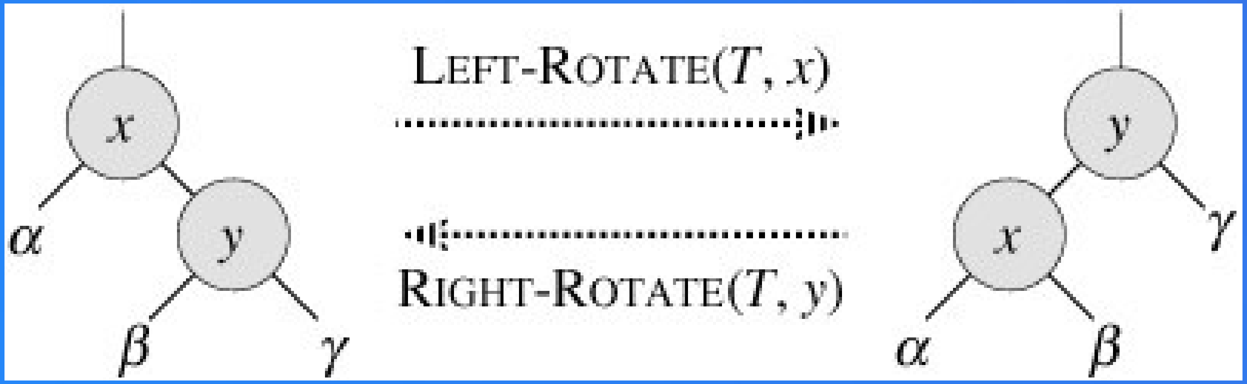
- Red-black Tree 图示 2



(b)

共享 NIL（叶子）的表示

- Red-black Tree 操作 - 旋转和右旋
  - 注：旋转操作由插入和删除操作触发
  - 图示：



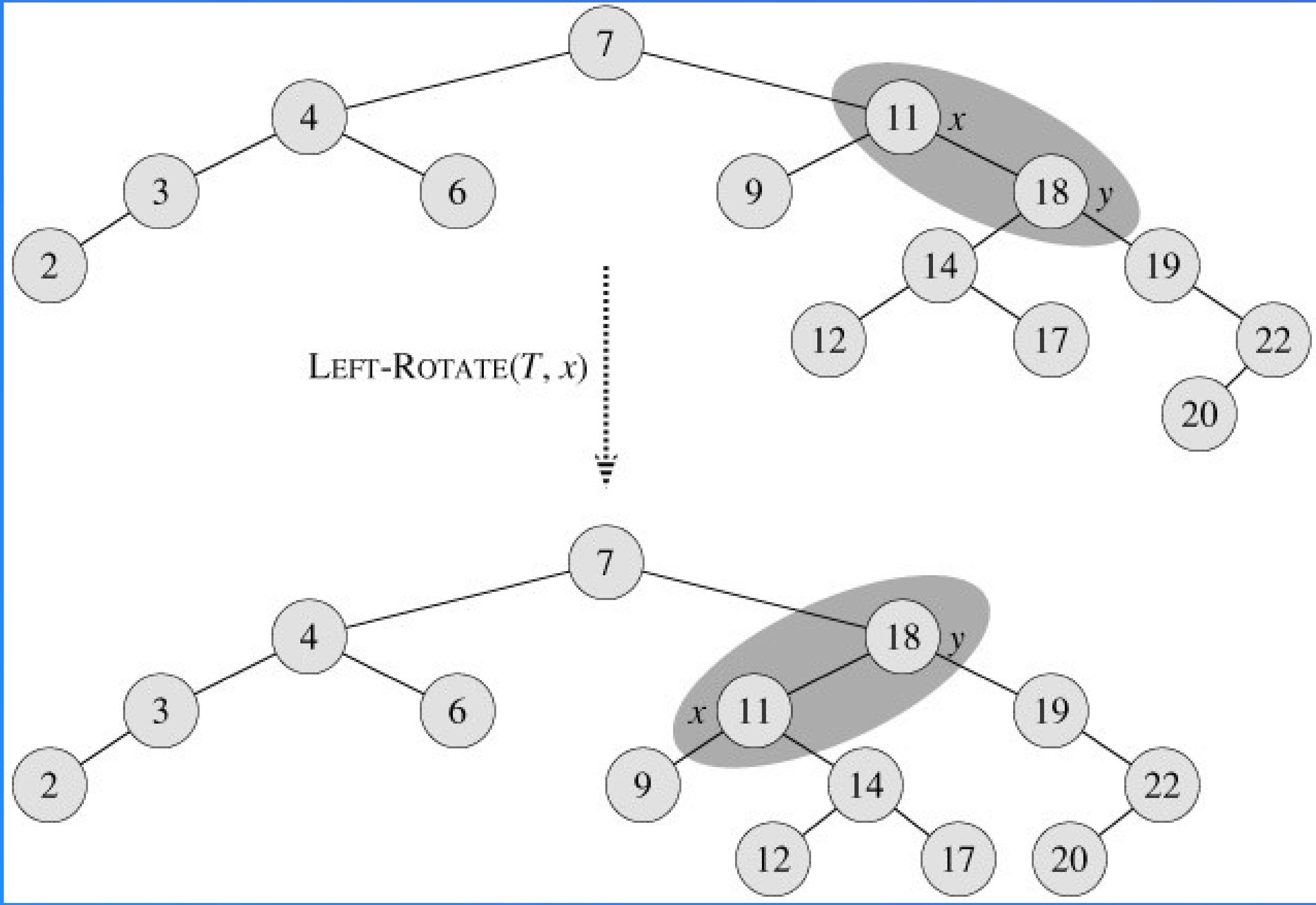
数据结构简介 - Red-Black Tree

- Red-black Tree 操作 - 左旋
  - 伪码：
    该操作前置条件：
    - x->right ≠ tree->nil && root->parent==tree->nil

```
LEFT-ROTATE(x)
y = x->right  // Set y.
x->right = y->left  // Turn y's left subtree into x's right subtree.
y->left->parent = x
y->parent = x->parent // Link x's parent to y.
if x->parent == tree->nil
   then tree->root = y
   else if x == x->parent->left
           then x->parent->left = y
           else x->parent->right = y
y->left = x // Put x on y's left.
x->parent = y
```

数据结构简介 - Red-Black Tree

- Red-black Tree 操作 - 左旋图示



数据结构简介 - Red-Black Tree

- Red-black Tree 操作 - 右旋
  - 伪码：
    该操作前置条件：
    - x->left ≠ tree->nil && root->parent==tree->nil

```
RIGHT-ROTATE(x)
y = x->left  // Set y.
x->left = y->right  // Turn y's right subtree into x's left subtree.
y->right->parent = x
y->parent = x->parent // Link x's parent to y.
if x->parent == tree->nil
   then tree->root = y
   else if x == x->parent->right
           then x->parent->right = y
           else x->parent->left = y
y->right = x // Put x on y's right.
x->parent = y
```

数据结构简介 - Red-Black Tree

- Red-black Tree 操作 - 插入节点
  - 插入节点一般做以下 3 个动作：
    - 1，按二叉搜索树的插入操作方式，将节点插入到树中；
    - 2，将插入的节点的颜色设为 Red
    - 3，第 2 个操作有可能导致不符合红黑树的性质，所以需要对树的节点进行调整

数据结构简介 - Red-Black Tree

- Red-black Tree 操作 - 插入节点（续 1 ）
  - 伪码：插入操作

```
RB-INSERT(z)
y = tree->nil
x = tree->root
while x ≠ tree->nil
    do y = x
       if z->key < x->key
          then x = x->left
          else x = x->right
z->parent = y
if y == tree->nil
   then tree->root = z
   else if z->key < y->key
           then y->left = z
           else y->right = z
z->left = tree->nil
z->right = tree->nil
z->color = RED
RB-INSERT-FIXUP(z)
```

Red-black Tree 操作 - 插入节点（续 2 ）

伪码：修复操作

```
RB-INSERT-FIXUP(z)
while z->parent->color == RED
  do if z->parent == z->parent->parent->left
    then y = z->parent->parent->right
      if y->color == RED
        then z->parent->color = BLACK   // Case 1
          y->color = BLACK              // Case 1
          z->parent->parent->color = RED   // Case 1
          z = z->parent->parent            // Case 1
        else if z == z->parent->right
          then z = z->parent          // Case 2
            LEFT-ROTATE(z)            // Case 2
          z->parent->color = BLACK    // Case 3
          z->parent->parent->color = RED   // Case 3
          RIGHT-ROTATE(z->parent->parent) // Case 3
    else (same as then clause
          with "right" and "left" exchanged)
tree.root->color = BLACK
```

- Red-black Tree 操作 - 插入节点（续 3 ）
  - 关于修复操作的 case
    - case 1: z 节点的 uncle 节点 y 为红色
    - case 2: z 节点的 uncle 节点 y 为黑色，且 z 节点是右子节点
    - case 3: z 节点的 uncle 节点 y 为黑色，且 z 节点是左子节点

- Red-black Tree 操作 - 插入操作分析
  - 插入操作的时间复杂度：O(lg n)
  - 插入修复操作最多作 2 次旋转
  - 整个插入以及修复操作的时间复杂度为：O(lg n)

数据结构简介 - Red-Black Tree

- Red-black Tree 操作 - 删除节点

```
RB-DELETE(z)
if z->left == tree->nil or z->right == tree->nil
    then y = z
    else y = TREE-SUCCESSOR(z)
if y->left ≠ tree->nil
    then x = y->left
    else x = y->right
x->parent = y->parent
if y->parent == tree->nil
    then tree.root = x
    else if y == y->parent->left
            then y->parent->left = x
            else y->parent->right = x
if y ≠ z
    then z->key = y->key
         copy y's satellite data into z
if y->color == BLACK
    then RB-DELETE-FIXUP(x)
return y
```

数据结构简介 - Red-Black Tree

Red-black Tree 操作 - 删除修复

```
RB-DELETE-FIXUP(x)
while x ≠ tree->root and color[x] == BLACK
  do if x == x->parent->left
       then w = x->parent->right
         if w->color == RED
           then w->color = BLACK       // Case 1
             x->parent->color = RED    // Case 1
             LEFT-ROTATE(x->parent)    // Case 1
             w = x->parent->right      // Case 1
         if w->left->color == BLACK and w->right->color == BLACK
           then w->color = RED             // Case 2
             x = x->parent                 // Case 2
           else if w->right->color == BLACK
             then w->left->color = BLACK    // Case 3
                 w->color = RED             // Case 3
                 RIGHT-ROTATE(w)            // Case 3
                 w = x->parent->right       // Case 3
               w->color = x->parent->color   // Case 4
               x->parent->color = BLACK      // Case 4
               w->right->color = BLACK       // Case 4
               LEFT-ROTATE(x->parent)        // Case 4
               x = tree->root                // Case 4
       else (same as then clause with "right" and "left" exchanged)
x->color = BLACK
```

数据结构简介 - Red-Black Tree

- Red-black Tree 操作 - 删除节点（续 2 ）
  - 关于修复操作的 case
    - case 1: x 节点的 sibling 节点 w 为红色
    - case 2: x 节点的 sibling 节点 w 为黑色，且 w 的 2 个子节点均为黑色
    - case 3: x 节点的 sibling 节点 w 为黑色，且 w 的左子为红色、右子为黑色
    - case 4: x 节点的 sibling 节点 w 为黑色，且 w 的右子为红色

- Red-black Tree 操作 - 删除操作分析
  - 删除操作的时间复杂度： O(lg n)
  - 删除修复操作最多作 3 次旋转
  - 整个删除以及修复操作的时间复杂度为： O(lg n)

数据结构简介 - Red-Black Tree

- 红黑树 - 小结
  - 红黑树的各项操作在最坏情况下能保证 O(lg n) 的时间复杂度
  - C++ 标准库的 STL 容器 map 和 set 的实现中大多以红黑树为内部数据结构

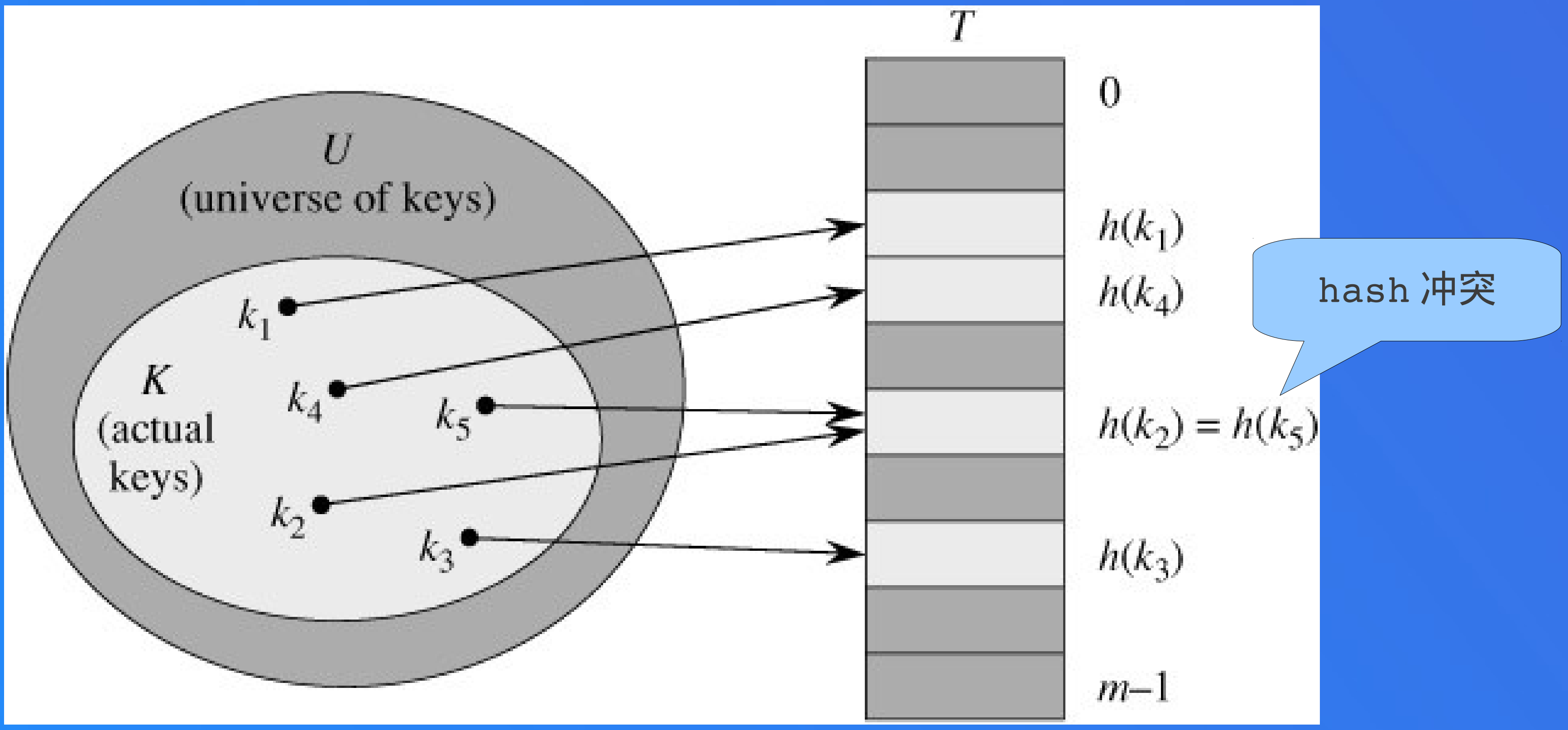
- 数据结构简介
  - Dynamic Array
  - Linked List
  - Binary Search Tree
  - Red-Black Tree
  - Hash Table
  - Stack & Queue

数据结构简介 - Hash Table

- 关于 Hash Table
  - Hash Table 是一种能够高效的通过 hash 函数将元素的 key 映射到确切 index 的数据结构。
  - Hash Table 非常适合用于实现字典结构，其 insert、search、delete 操作（注意：没有针对排序操作）非常高效，通常能保证 O(1) 或接近 O(1) 的性能。

轩辕IT培训

- Hash Table 图示



数据结构简介 - Hash Table

- 访问和操作 Hash Table 中的元素
  - 查找元素：
    - 首先通过 hash 函数计算元素的 key 的 hash 值；
    - hash 值 % 该 hash Table 的桶 (Buckets) 数，得出元素所在的桶的下标 (index of bucket)；
    - 在桶中查找元素
  - 插入元素：
    - 首先通过 hash 函数计算元素的 key 的 hash 值；
    - hash 值 % 该 hash Table 的桶 (Buckets) 数，得出元素所在的桶的下标 (index of bucket)；
    - 将元素放入该桶中
  - 删除和修改：
    - 同插入类似

数据结构简介 - Hash Table

- 决定 Hash Table 性能的几个方面
  - 1， hash 函数
  - 2， load factor（装载比例）
  - 3， buckets size（桶的数目）
  - 4， hash 冲突的解决方案

数据结构简介 - Hash Table

轩辕IT培训

- Hash 函数
  - hash 函数就是计算 key 的 hash 值的算法
  - 一个好的 hash 函数可以减少 hash 冲突，所谓 hash 冲突即两个不同的 key 计算出的 hash 值相同

数据结构简介 - Hash Table

轩辕IT培训

- Load Factor（装载比例）
  - 所谓装载比例即： n / s
    - n - Hash Table 中的元素个数
    - s - 桶的个数
  - 当该比例值接近 0.7，查找的性能将会变差，所以保持适当的装载比例是保证高效查找的条件之一
  - 一般而言，尽量保证装载比例值不超过 0.7
  - 当装载比例大于 0.7 后，需增加桶的数目，这时需将所有元素的位置（所在的桶）进行重新调整，该过程被称为： rehash
    - 在某些 hash table 的实现中， rehash 操作可能会使原先指向元素的指针（或迭代器）失效

数据结构简介 - Hash Table

轩辕IT培训

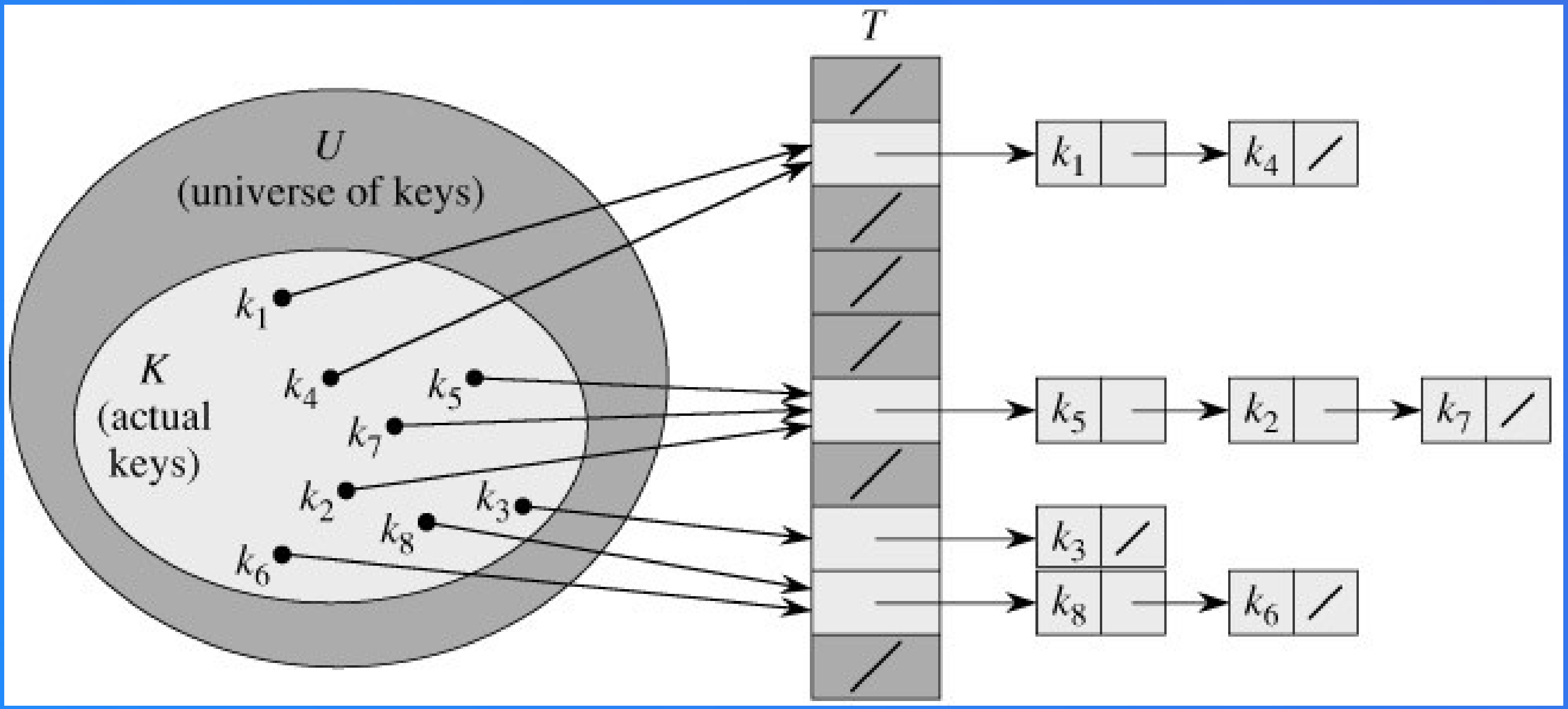
- Buckets Size （桶的个数）
  - 与 hash 函数一样，桶的个数也是决定 hash 冲突的几率的条件之一
  - 一般而言，桶的个数越大，出现 hash 冲突的几率越小（前提是拥有一个高质量的 hash 函数，能保证元素尽量均匀的放置在不同的桶中），但是，从另一个方面考虑，桶的数目越大，则造成空间浪费的趋势越明显
  - 桶的数目取值有多种方式，但以质数为桶数是比较常用的方式

数据结构简介 - Hash Table

- hash 冲突的解决方案
  - hash 冲突总是存在的，除了实现高效的 hash 函数和合理的装载比例之外，解决 hash 冲突的方式还有：
    - Separate chaining
    - Open addressing
    - Coalesced hashing
    - ...

数据结构简介 - Hash Table

- 解决冲突示意图
  - 下图示意采用 Separate chaining 方式解决冲突



数据结构简介 - Hash Table

- Hash Table - 小结
  - 与线性数据结构和树结构相比，一个好的 Hash Table 能提供更优秀的查找、插入、删除的操作性能（ O(1) 或接近 O(1) ）
  - 常用于实现字典结构、 Hash Map 、 Hash Set 等关联容器
  - 有关 hash 算法的资源：
    - GNU gperf ： 产生 perfect hash function 的组件 http://www.gnu.org/software/gperf/
    - 详细介绍 hash table 的文章 http://en.wikipedia.org/wiki/Hash_table

数据结构简介 - Stack & Queue

- 数据结构简介
  - Dynamic Array
  - Linked List
  - Binary Search Tree
  - Red-Black Tree
  - Hash Table
  - Stack & Queue

数据结构简介 - Stack & Queue

轩辕IT培训

- 关于 Stack - 栈
  - 栈是一种拥有元素 先进后出 (FILO) 性质的数据结构
  - 栈通常提供的操作：
    - push ：压栈，将元素放入栈中，时间复杂度为 O(1)
    - pop ：出栈，取得并删除栈顶的元素（最后放入的元素），时间复杂度为 O(1)

注：也有些实现将 pop 动作拆分为 2 个动作：先取得栈顶元素，然后删除它（如：C++ 标准库中的 stack ）

数据结构简介 - Stack & Queue

- 关于 Queue - 队列
  - 队列是一种拥有元素 先进先出 (FIFO) 性质的数据结构
  - 队列通常提供的操作：
    - push_back： 入列，在队列的末尾加入元素，时间复杂度为 O(1)
    - pop_front： 出列，删除队列前端的元素，时间复杂度为 O(1)

注：也有些实现将 pop_front 动作拆分为 2 个动作：先取得队列前端的元素，然后删除它（如： C++ 标准库中的 queue ）

数据结构简介 - 小结

- 常用数据结构的适用场合：
  - 如果访问元素的操作的效率十分重要，可以使用 Array、 Dynamic Array、 Hash Table
  - 如果插入、删除操作的效率十分重要，可以使用 Linked List 和 Hash Table
  - 如果查找元素的操作效率十分重要，首选 Hash Table，其次为 Red-Black Tree 或其它自平衡二叉树
  - 如果要求一个有序的序列，首选 Red-Black Tree 或其它自平衡二叉树，其次是 Linked List
  - 关于 Hash Table 和 Red-Black Tree 的折衷：
    - 如果查找、删除、添加元素的效率十分重要：在空间允许的条件下，首选 Hash Table；如果空间有要求的情况下，则可考虑首选 Red-Black Tree