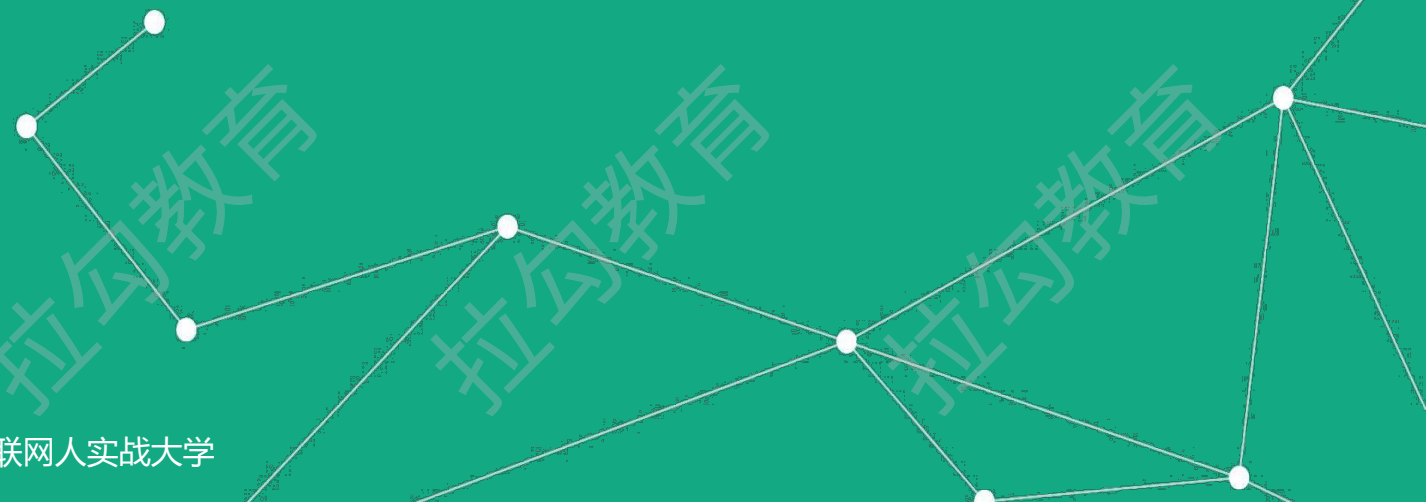
课时 09

树的基本原理

1. 均摊时间复杂度
2. 哈希表的应用——缓存
3. 哈希表在Facebook中的应用



有了数组和链表，为什么还需要树

拉勾 LAGOU.COM

数组是一个很容易编程实现的静态数据结构，可以很好的支持随机访问

链表相对来说多了一些动态特性，链表适合的应用场景为需要比较频繁的增、删和更新操作

相比数组，它的缺点是随机访问的时间复杂度为 O(n)

有了数组和链表，为什么还需要树

拉勾 LAGOU.COM

队列是先进先出，堆栈是先进后出

哈希表的重要应用场景是快速查询和更新，哈希表的查询和更新时间复杂度都是均摊 O(1)

数组和链表的局限性正是来自于它们的线性特点

当你需要在数组或者链表中查询一个元素时

则需要从头到尾遍历这个列表，此时的时间复杂度为 O(n)

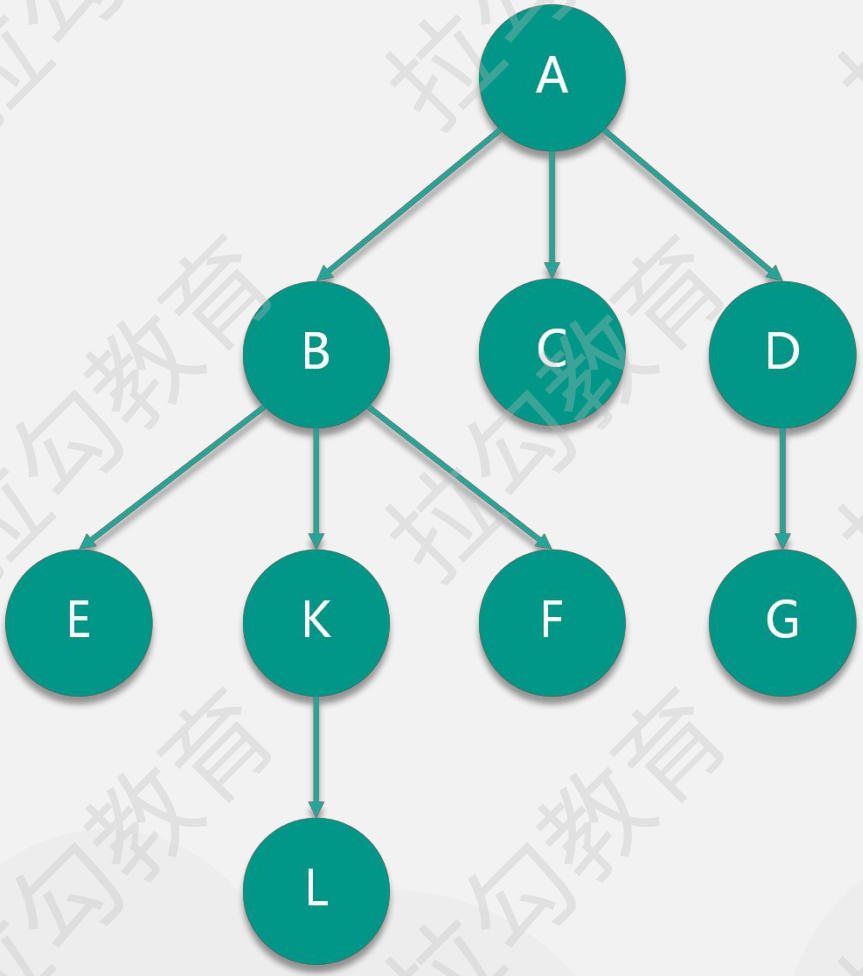
有了数组和链表，为什么还需要树

在现代的互联网应用中，1 万的 QPS 非常常见

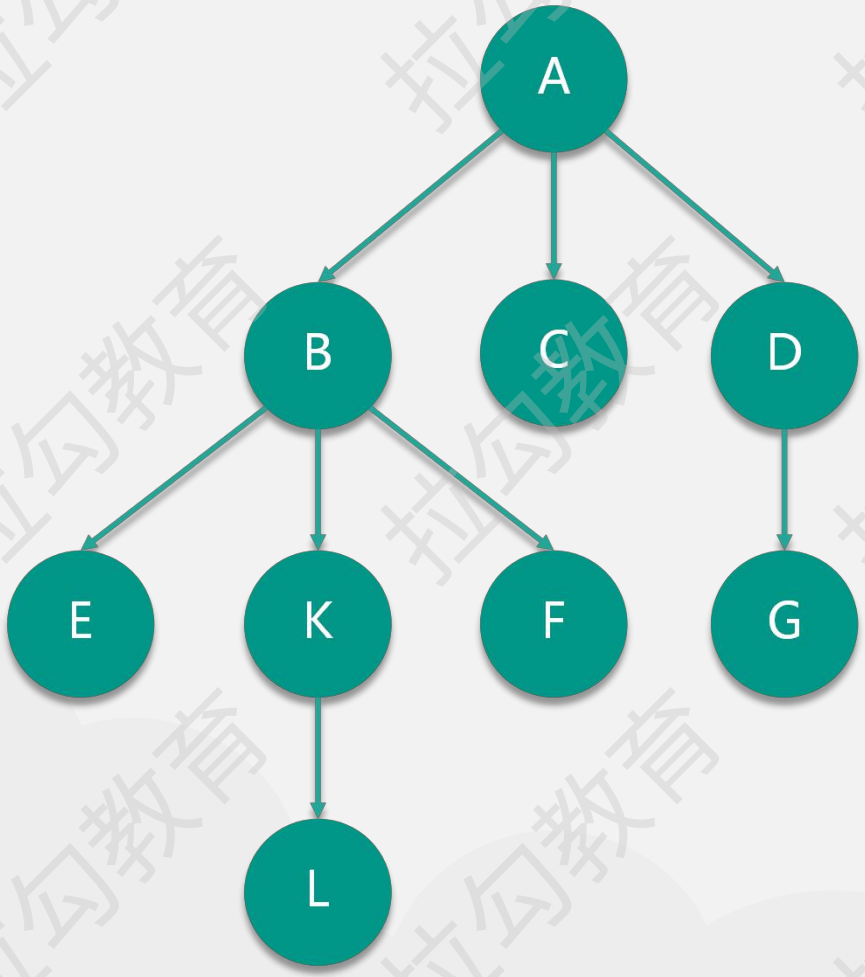
如果要支撑每秒高达 1 万的访问，显然如果仅仅使用数组和链表是不够的

所以树应运而生，相比数组和链表，其抽象表达元素之间的关系更为复杂

树的定义和例子



树的定义和例子



- 一个节点的深度是从根节点到自己边的数量

  比如 K 的深度是 2

- 节点的高度是从最深的节点开始到自己边的数量

  比如 B 的高度是 2

一棵树的高度也就是它根节点的高度

树的定义和例子

树的递归定义： n（n>=0）个节点的有限集合，其中每个节点都包含了一个值，并由边指向别的节点（子节点），边不能被重复，并且没有节点能指向根节点

广义的树可以用来表达有层次结构的数据关系，比如文件系统（File System）是一个有层次关系的集合文件系统中每一个目录（Directory）可以包含多个目录，没有子目录的叶子节点也就是文件（File）

树的编程实现方式

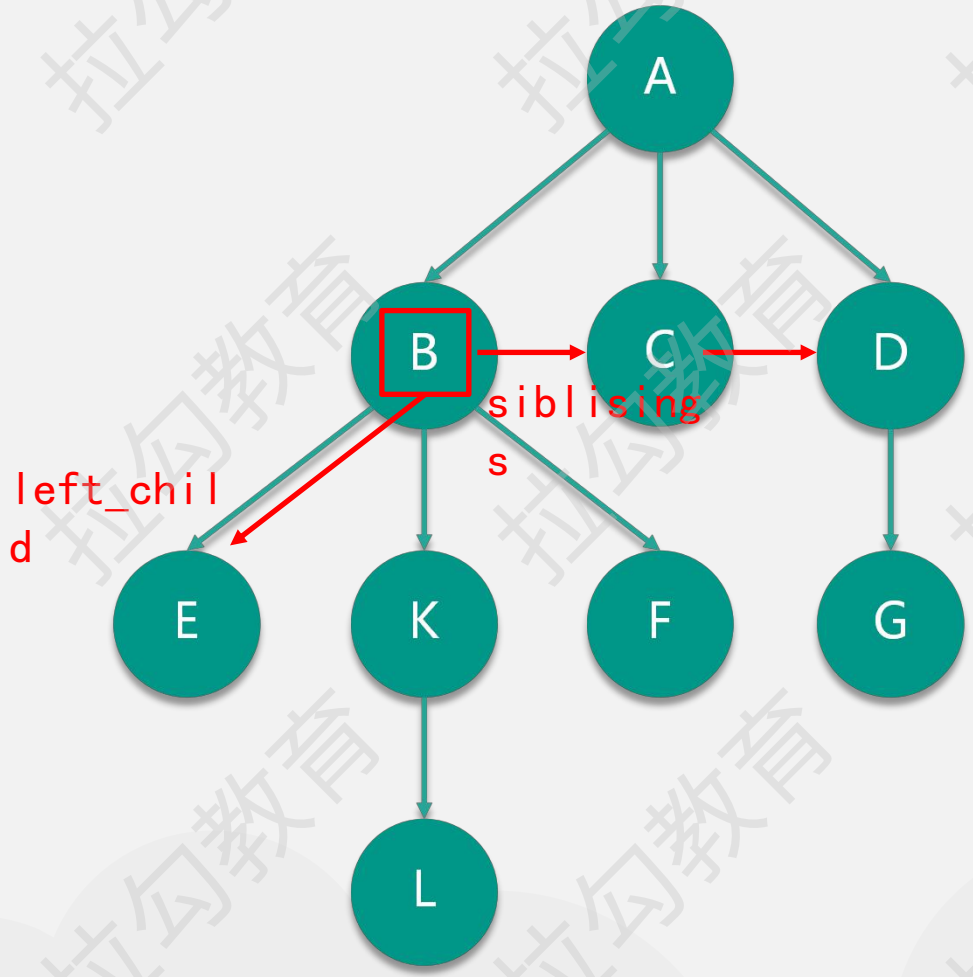
常见的实现方式一种是基于链表的实现，另一种是基于数组的实现

树的编程实现方式

```
class TreeNode {
    Data data;
    LinkedList siblings;
    TreeNode left_child;
}
```

基于链表的实现一般是每一个节点类型维护一个子节点指针和一个指向兄弟节点的链表，我们把它称作**左孩子兄弟链表法**

树的编程实现方式

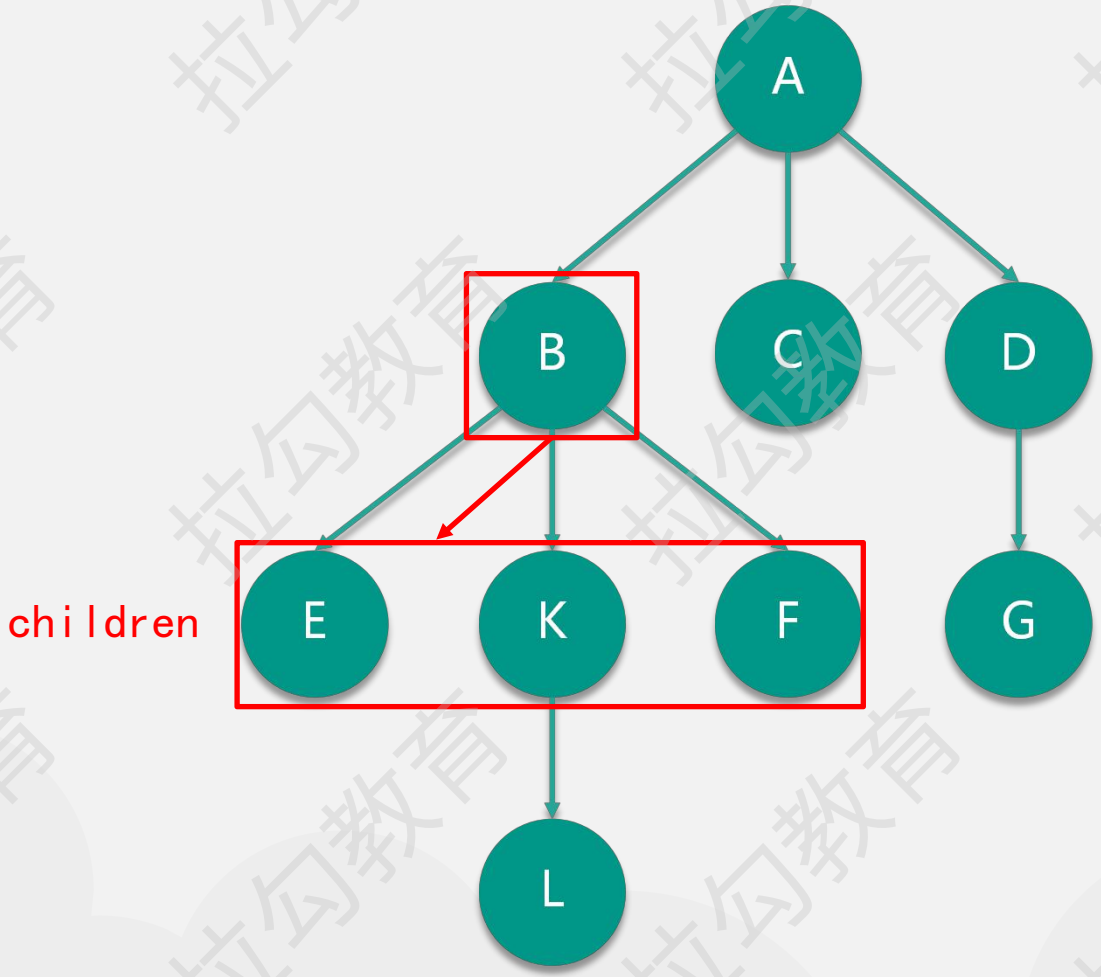


树的编程实现方式

```
class TreeNode {
        Data data;
        ArrayList
children;
}
```

另一种基于数组的实现方式是每一个节点维护一个包含它所有子节点的数组，我们把它称作孩子数组法

树的编程实现方式



树的编程实现方式

在实现一个树的编程中，最容易犯的错误在于内存管理和节点的增、删、改

简单幼稚的孩子数组法

```
class TreeNode {
        Data data;
        std::vector<TreeNode>
children;
};
```

所有子节点的内存都是由父节点管理

也就是说当父节点被删除时

子节点的内存也会被自动清理

树的编程实现方式

```
class TreeNode {
    Data data;
    std::vector<TreeNode*>
children;
};
```

子节点的内存由谁来管理呢?

树的编程实现方式

```
class NodePool {
  std::vector<TreeNode> nodes;
}
```

一个解决办法是除了实现这样一个 TreeNode 类

再去实现一个 NodePool 类用来管理所有的节点内存

另一个解决办法则是模仿链表的内存管理，在左孩子兄弟链表法中比较容易实现

因为它实际上就是一个二叉的链表

树的编程实现方式

在左孩子兄弟链表法中

插入一个树节点的复杂度是 O(1)

相当于是在 siblings 链表中插入一个元素，或者是增加一个 left_child 节点

而在孩子数组法中插入一个节点和数组插入元素类似，需要挪动其后的所有节点，则是 O(n) 的复杂度

删除节点相比插入节点更为复杂一些，同样的，在孩子数组法中，删除单个节点的复杂度是 O(n)

因为你需要去拷贝这个节点的子树到上层节点。在左孩子兄弟链表法中，删除单个节点的复杂度为 O(1)

只需要去重新整理几根指针引用就可以了

树的遍历和基本算法

除了树的增、删、改，树最最常见的操作就是查找操作，也就是遍历

如果没有遍历，增、删、改根本无从谈起，因为你都不知道去操作哪个节点！

树的遍历根据根节点的访问顺序，可以分为前序遍历、后续遍历和按层遍历等多种

前序遍历用伪代码

```
先访问根节点N
递归访问N的子树
```

后续遍历

```
递归访问N的子树
后访问根节点N
```

树的遍历和基本算法

拉勾 LAGOU.COM

```
class TreeNode {
  TreeNode* left_child;
  TreeNode* sibling;
}

void PreorderVisit(TreeNode root) {
  Visit(root);
  TreeNode* child = root->left_child;
  for (TreeNode* child = root->left_child; child != nullptr; child = child->sibling) {
    PreorderVisit(child);
  }
}
```

总结

这一讲了解了我们为什么需要树

掌握了树的概念和定义

更重要的是比较了树的实现方式和遍历方相信你肯定也已经看出来了，是 O(n)