数据结构讲义

马彬

目录

一、数据结构

所谓数据结构，就是数据在计算机内存中的存储和组织结构，不同的组织方式有着不同的特性，以适用于不同的应用场景。

1.1、线性数据结构

所谓线性数据结构，是指每个节点只有唯一的前驱节点和后继节点.

1.1.1 数组

数组是最简单最基础最直观的一种线性数据结构. 假如我们要使用数组存储 10 个数据，那我们其实只需要把这 10 个数据挨个按顺序放入内存即可，同时记录内存的首地址. 这样一来，如果我们需要查找第 4 个数据的位置，那么只需要根据首地址再加上 3 个数据长度的偏移即可. 所以数组的查询时间复杂度是 O(1).

不过如果我们要在数组中间某个位置插入一个数据，那就费劲了，因为我们需要把数组这个位置后面的所有数据都向右位移一个单位. 在最极端的情况下，我们需要插到第一个元素前面，那么我们需要将所有的元素都进行位移，所以其时间复杂度为 O(N).

删除数据同理，需要将要删除的元素后面的元素都向左位移一个单位，所以其时间复杂度也为 O(N).

C/C++中的数组定义方式：

```
int A[100]; // 长度为 100 的 int 型数组
```

Java 中的数组定义方式：

```
int[] A = new int[100];
```

1.1.2 链表

链表也是一种线性数据结构，它并不像数组一样在内存中是连续分布的. 它在内存中是离散的，所以每一个节点上除了保存数据之外，还需要保存下一个节点的地址，像一根链条一样，这也是链表这个名词的来源.

由于链表的节点地址不是连续的，所以我们不能利用简单的"首地址+偏移"的方式去寻址，而要顺着链表的链去挨个寻找，所以链表的查询复杂度是 O(N).

如果链表的节点中只保存后继节点的地址，那么这种链表我们称之为单向链表，或者单链表；如果链表中的节点除了保存后继节点的地址，还保存前驱节点的地址，那么这种链表就叫做双向链表.

C/C++中的单链表节点定义：

```
typedef struct _node {
        int value;
        struct _node* next;
} node, *pnode;
```

虽然链表查询慢，但是它插入删除起来确实很快的：假如我们要删除一个链表的某个节点，那么我们只需要将这个节点的前驱节点中保存的地址改成这个节点的后继节点的地址即可，所以链表的删除复杂度为 O(1)．如果我们要在链表某个节点后面插入一个节点，我们首先在这个新节点中保存那个节点的后继节点的地址，然后将那个节点中的地址改成这个新节点即可，所以它的插入复杂度也是 O(1)．

单链表删除节点：

```
//假设 p->next 为要删除的节点（即 p 为要删除节点的前驱）
void deleteNode(pnode p) {
    p->next = p->next->next;
}
```

单链表插入节点：

```
//假设要在 p 后面插入一个节点
void insertNode(pnode p, pnode newNode) {
    newNode->next = p->next;
    p->next = newNode;
}
```

1.1.3 队列

所谓队列，顾名思义就是一种模拟现实中排队机制的数据结构．它只能从尾端插入新节点，从头部删除节点，也即先进先出(FIFO)．队列既可以用数组实现，也可以双向链表来实现，但是通常为了节省空间，建议使用双向链表来实现．不管使用哪一种方式来实现，队列的插入和删除均可以达到 O(1)的时间复杂度．

有一种被称为双端队列的数据结构，它可以支持从任何一端插入和删除．

还有一种被称为优先队列的数据结构，它每次并不是删除队首的元素，而是删除整个队列中优先级最高的元素，但这种队列和此处所讲的队列有很大的不同，一般使用堆来实现，关于堆我们将在后文详述．

```
#include <queue>
  queue<int> q;
```

1.1.4 栈

栈和队列稍有不同，它是一种先进后出(FILO)的数据结构，它只能从尾端插入和删除，对于栈来说它的尾端被成为栈顶，另一端被称为栈底．与队列相同，栈也可以用数组实现，也可以用双向链表来实现．

C++ STL 中的栈：

```
#include <stack>
  stack<int> s;
```

1.2、非线性数据结构

所谓非线性数据结构，是指每个节点可以有多个前驱和后继节点.

1.2.1 树

树是最典型的非线性数据结构，它几乎也是所有数据结构中最重要的一个分支.

树的每个节点可以有任意多个后继节点(包括零个)，但是只能有一个前驱节点，所以这种数据结构的形状看起来像一棵倒长的树，这也是它名称的来历. 对于树来将，它的前驱节点被称为父节点，而后继节点被称为子节点，这一点也十分容易理解：一个人只能有一个父亲，但是却可以有多个儿子.

N 叉树

我们给树加一个约束条件：如果一棵树每个节点的子节点个数不超过 N，则称这棵树为 N 叉树. 子节点的个数也称为这个节点的"度".

二叉树

在所有 N 叉树中，最最常见和常用的当属二叉树，"二"这个数字很重要，远比三和四重要，因为世界上有很多事物是二值的.

C/C++中二叉树的节点定义：

```
typedef struct _node {
        int value;
        struct _node* left;
        struct _node* right;
  } node, *pnode;
```

完全二叉树：除了最下面两层，其余各层的节点度数均为 2，而且最后一层的节点均集中在左侧.

满二叉树：除了最下面一层，其余各层的节点度数均为 2.

二叉搜索树：每个节点的左节点的值均小于该节点，而右节点的值均大于该节点.

平衡二叉树：如果对于一棵树，它之中所有节点的左右子树的高度差均不大于 1，则称这棵树是平衡的. 有很多树满足平衡二叉树的特性，比如 AVL 树，红黑树，平衡二叉搜索树等...

所有的数据结构都有两种遍历方式即深度优先遍历和广度优先遍历(又称层次遍历)，对于线性数据结构，这

两种遍历方式并无差异(因为线性数据结构没有所谓的广度，那就更谈不上采用广度优先了)．二叉树当然也能够支持这两种遍历方式．

二叉树的深度优先遍历有三种：先序遍历，中序遍历，后序遍历．这三种遍历方式的不同之处在于遍历顺序，对于先序遍历，我们先遍历当前节点，再遍历它的左节点，再遍历它的右节点；对于中序遍历，则是先遍历左节点，再遍历当前节点，再遍历右节点；对于后序遍历，则是先遍历左节点，再遍历右节点，再遍历当前节点．深度优先遍历一般使用递归来实现，或者利用栈来进行非递归实现．

二叉树的深度优先遍历(先序)：

```
void preOrder(pnode root) {
    if (root == NULL)
```

```
void preOrder(pnode root) {
    if (root == NULL)
        return;
    preOrder(root->left);
    someOperation(root); //  做操作
    preOrder(root->right);
}
```

二叉树的深度优先遍历(中序)：

二叉树的深度优先遍历(后序)：

二叉树的广度优先遍历比较简单，它通常利用一个队列来实现：当遍历完当前节点的时候，把当前节点的左右子节点依次加入待遍历队列，每次遍历下一个节点的时候就从这个队列中取出队首的节点．这种做法可以保证上层节点总可以在下层节点之前被访问，而且对于同一层节点，也严格满足了从左到右的访问顺序．

二叉树的广度优先遍历：

```
void BFS(pnode root) {
    queue<pnode> q;
    q.push(root);
    while (!q.empty()) {
        pnode cnt = q.front();
        q.pop();
        if (cnt == NULL)
            continue;
        someOperation(cnt); //  做操作
        q.push(cnt->left);
        q.push(cnt->right);
    }
}
```

1.2.2 堆

在这里我们讲二叉堆，所以这里的堆特指二叉堆。所谓二叉堆，其实就是一棵二叉树，这棵二叉树满足一个

特殊的性质：每个节点的值均不大于(或不小于)它的子节点的值。如果是不大于，那么这个堆被称为小根堆，顾名思义就是堆顶(根)最小；如果是不小于那就是大根堆。

建堆过程：

```
void buildHeap(int A[], int n) {
void heapify(int heap[], int n, int i) {
        int largest = i;
        int left = LEFT(i);
        int right = RIGHT(i);
        if (left >= 0 && heap[largest] < heap[left])
            largest = left
        if (right < n && heap[largest] < heap[right])
            largest = right
        int tmp = heap[i];
        heap[i] = heap[largest];
        heap[largest] = tmp;
        if (largest != i)
            heapify(heap, n, largest);
    }
```

保持堆性质：

LEFT 和 RIGHT 的定义：

```
#define LEFT(x) ((x) << 1) // 2 * x
#define RIGHT(x) (((x) << 1) + 1) // 2 * x + 1
```

1.2.3 图

图是一种约束更松的非线性数据结构，对于图来说，每个节点可以有任意多个前驱节点. 树是一种特殊的图. 对于图来说，我们使用"入度"来表示节点的前驱节点的个数，用"出度"来表示节点的后继节点的个数. 值得一提的是，对于树来说，其入度总为 1，所以我们省略了这个定义，仅仅用"度"来表示出度. 对于图来说，它的度等于入度+出度.

图由顶点和边两部分构成，顶点即节点，而边则表示两个节点之间的一种连接关系.

如果一条边连接了某个节点和它自身，则称这条边为一个自环.

如果图中的某些边可以构成一个回路，则称这个图是有环的，这种图被称作有环图.

有向图

如果图中的边都是单向的，那么这个图就被成为有向图.

完全有向图：对于一个拥有 N 个顶点的有向图，如果每个顶点的入度和出度均为 N-1，则称这个有向图为完全有向图.

无向图

如果图中的边都是双向的，那么这个图就被成为无向图.

二、算法

2.1、递归

递归是指一个函数调用自身的行为，举一个求解斐波那契数列的例子：

```
int getFab(int x) { // 求第 x 个斐波那契数
    if (x == 1) // 边界条件
        return 0;
    if (x == 2) // 边界条件
        return 1;
    return getFab(x - 1) + getFab(x - 2); // 递归表达式
}
```

如果不使用递归，我们可以采用迭代的方式来完成同样的功能：

```
int getFab(int x) {
    fab[0] = 0; // fab 为一个足够大的 int 数组
    fab[1] = 1;
    for (int i = 2; i < x; i++)
        fab[i] = fab[i - 1] + fab[i - 2];
    return fab[x - 1];
}
```

2.2、排序

排序是将一个无序集合变成有序集合的过程。

2.2.1 冒泡排序

保证所有相邻元素之间的大小关系，既可以保证集合有序.

平均时间复杂度 O(n^2)，最坏时间复杂度 O(n^2)，空间复杂度 O(1).

```
void bubbleSort(int a[], int n) {
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (a[j] > a[j - 1]) {
                    int tmp = a[j];
                    a[j] = a[j - 1];
                    a[j - 1] = tmp;
                }
            }
        }
    }
```

2.2.2 选择排序

每次在剩下的待排序的数中选出一个最值：

平均时间复杂度 O(n^2)，最坏时间复杂度 O(n^2)，空间复杂度 O(1).

```
void selectionSort(int a[], int n) {
for (int i = 0; i < n; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (a[j] < a[minIndex])
                    minIndex = j;
            }
            int tmp = a[minIndex];
            a[minIndex] = a[i];
            a[i] = tmp;
        }
    }
```

2.2.3 堆排序

堆排序通常基于二叉堆实现，以大根堆为例，堆排序的实现过程分为两个子过程。第一步为取出大根堆的根节点(当前堆的最大值)，由于取走了一个节点，故需要对余下的元素重新建堆。重新建堆后继续取根节点，循环直至取完所有节点，此时数组便已经排好序。

平均时间复杂度 O(nlogn)，最坏时间复杂度 O(nlogn)，空间复杂度 O(1).

```
void heapSort(int a[], int n) {
    buildHeap(n, a);
    for (int i = n - 1; i >= 1; i--) {
        int tmp = a[i];
        a[i] = a[0]; // a[0]是堆顶元素，是最大值
        a[0] = tmp;
        headpify(a, i, 0); //  以前 i 个元素作为一个堆，重新调整产生最大值 a[0]
    }
}
```

2.2.4 归并排序

归并的思想很简单，如果有两个已经排好序的数组(两路归并)，那么将其合成一个大的排好序的数组只需要耗时 O(n)。

平均时间复杂度 O(nlogn)，最坏时间复杂度 O(nlogn)，空间复杂度 O(n).

```
        } else {
            result[cnt++] = a[j];
            j++;
        }
    }
    if (i < m) //  如果前半段有剩下的
        for (int k = i; k < m; k++)
            result[cnt++] = a[k];
    if (j < n) //  如果后半段有剩下的
        for (int k = j; k < n; k++)
            result[cnt++] = a[k];
    for (int i = 0; i < n; i++)
        a[i] = result[i];
    delete[] result;
}
```

2.2.5 快速排序

快排的思想和归并有些类似，都是一分为二的思路(分治法，分而治之)。快排会先在数组中选择一个值作为基准值，然后将比基准值小的元素放到基准值左边，将比基准值大的元素放到基准值右边。这样一来左边数组的值都小于右边，然后对左右两段数组分别做快排即可。

平均时间复杂度 O(nlogn)，最坏时间复杂度 O(n^2)，空间复杂度 O(logn).

```
void quickSort(int a[], int n) {
        if (n <= 1)
            return;
        // 基准值选 a[0]
        int l = 1;
        int r = n - 1;
        while (l <= r) {
            while (l <= r && a[l] < a[0]) {
                l++;
            }
            while (a[r] > a[0]) {
                r--;
            }
            if (l > r) // 成功找到「边界」
                break;
            int tmp = a[l];
            a[l] = a[r];
            a[r] = tmp;
        }
       // 此时 a[r]便是「边界」值，且 a[r]<=a[0]
        int tmp = a[r];
        a[r] = a[0]; // 与基准值交换
        a[0] = tmp;
        quickSort(a, r); // 基准值左侧
        quickSort(a + r, n - r) // 基准值右侧(包含基准值)
}
```

2.3 动态规划

动态规划是通过拆分问题，定义问题状态和状态之间的关系，使得问题能够以递推（或者说分治）的方式去解决.

下面用一个简单的例子说明一下：

爬楼梯问题

小明爬楼梯有个习惯，要么一步走一阶，要么一步走两阶。假如有一个楼梯长度为 n，请问小明有多少种不同的走法？

定义状态：dp[i]为长度为 i 的楼梯的总走法数

找转移方程(拆解问题)：如果小明现在到了第 i 个台阶上，那么有两种情况，要么小明是从第 i-1 个台阶上来的，要么就是从第 i-2 个台阶上来的。所以 dp[i]的求解可以进行拆解：dp[i] = dp[i - 1] + dp[i - 2]

找初始值：显然 dp[1] = 1, dp[2] = 1

2.4 哈希(Hash)

哈希俗称散列，是一种通过数据映射来达到快速查询效果的手段。一般这种映射关系被称作哈希函数，哈希函数是离散的，它的定义域是源数据的范围，值域是哈希后的值的范围。哈希函数的值域一般会小于定义域。

常见的哈希函数

直接取余法：

f(x) = x mod p (p 为一个素数)

冲突处理

如果对于两个 Key，K1 和 K2，有 f(K1) = f(k2)，则称产生了冲突。

一般有两类处理方法：

开放地址法

线性探测法：从冲突的位置开始往右边一个一个找空位，有了就放进去

平方探测法：把线性探测法的偏移改为平方数

随机探测法

链表法

在哈希表冲突的地方接一条单链表，把冲突的值追加到链表末尾

2.5 常见数据结构与算法的复杂度

这里有一个十分全面的资料：http://bigocheatsheet.com

三、经典题目

不同路径数(动态规划): http://algorithm.yuanbin.me/zh-cn/dynamic_programming/unique_paths.html

最大子段和(动态规划): http://algorithm.yuanbin.me/zh-cn/dynamic_programming/maximum_subarray.html

最短路径和(动态规划): http://algorithm.yuanbin.me/zh-cn/dynamic_programming/minimum_path_sum.html

找落单的数: http://algorithm.yuanbin.me/zh-cn/math_and_bit_manipulation/single_number.html

找 2 个落单的数: http://algorithm.yuanbin.me/zh-cn/math_and_bit_manipulation/single_number_iii.html

快速幂: http://algorithm.yuanbin.me/zh-cn/math_and_bit_manipulation/fast_power.html

O(1)时间判断某个数是否是 2 的幂次: http://algorithm.yuanbin.me/zh-cn/math_and_bit_manipulation/o1_check_power_of_2.html

求平方根(二分法): http://algorithm.yuanbin.me/zh-cn/binary_search/sqrt_x.html

求前 K 大的数(快排思想): http://algorithm.yuanbin.me/zh-cn/integer_array/kth_largest_element.html

阶乘末尾连续 0 的个数: http://algorithm.yuanbin.me/zh-cn/math_and_bit_manipulation/factorial_trailing_zeroes.html