

## Section 2.2 Data Structures 数据结构

### 预备知识

- 图论

### 如何选择最完美的数据结构

从一些数据结构中选择一个合适的数据结构来表示一个题目中的数据。

### 可否工作？

如果数据结构将不能正常工作，这是完全没有用的。对于一个问题，什么样的算法可以决定什么样的数据结构，并确保数据结构可以处理算法。如果不能，那么要么必须添加一些数据结构，或者你需要找到另外的数据结构来构建这个算法。

### 可否编码？

如果您不知道或不记得如何编写一个给定的数据结构，选择其他的数据结构。确保你有一个清醒的认识，知道每一个操作对数据结构的影响。在此，另一个要考虑的是内存。这个数据结构能在适当大小的内存空间里运行吗？如果不能就简化它，或选择一个新的数据结构。否则，从一开始就注定了它不能正常工作。

### 按时完成？

由于比赛是限定时间的，5 小时内解决 3 至 5 道题。如果你在第一题上为了数据结构就花了一个半小时，那么你基本上已经悲剧了。

### 可否调试？

在选择数据结构的时候很容易忘掉调试这部分。请记住一点，一个程序除非它能正常工作，否则它就是无用的。不要忘记，在整个比赛的时间中，调试占的比例是很大的，因此必须把调试时间考虑到写程序的时间中。一个数据结构是否容易调试取决于下面两个属性。

- **静态的数据结构更易于检查** 通常来说，规模更小、更紧凑的表达形式更容易检查。此外，静态分配的数组比链表甚至于动态数组更容易检查。
- **静态的数据结构更容易被显示** 对于更复杂的数据结构，最简单的检验方法是写一个小例子来输出数据。可惜，由于时间的限制，你可能想限制自己的文本输出。这意味着，像树和图的结构将难以检查。

### 是否快速？

很奇怪，速度是在选择数据结构的时候是最后一个要考虑的。一个慢的程序很容易发现是什么导致了慢，但是一个快速的错的程序却不容易发现什么导致了错，除非运气很好。

### 结论

总的说来，遵循 KISS 原则：“使其简单，傻瓜化。” (Keep It Simple, Stupid.) 有时候一定的复杂度是有必要的，但请确保它值得。请牢记：在开始的时候花时间去确保你选择了一个合适的数据结构，比之后不得不用另一个数据结构去替代它，要划算得多。

### 避免动态内存

通常情况下，你应当避免使用动态内存，因为：**使用动态内存会很容易犯错误!!** 重写已分配的内存，忘记释放内存，忘记分配内存，只是使用动态内存时引入的一点错误。此外，这些错误的出错代码很难告诉我们发生错误的位置，因为它可能发生在（可能更晚）内存操作时。**检查数据结构的含义太难了!!** 集成开发环境不能很好地操作动态内存，尤其对于 C 语言更是一塌糊涂。尝试考虑使用并行数组来实现动态内存，比如使用链表时用另一个数组来存储 next 值序列。有时你可以动态分配这些，但是因为它只需要完成一次，所以用数组来实现插入或删除操作会比分配释放内存更简单。尽管如此，有时动态内存也是一种好的办法，特别是对于未知数据范围的大型数据结构。

## 避免猎奇想法

不要掉进“猎奇”的陷阱。你可能刚发现了最有趣的结构，但是要记住：

- 好点子，不工作，没有用。
- 酷想法，编不出，也没用。

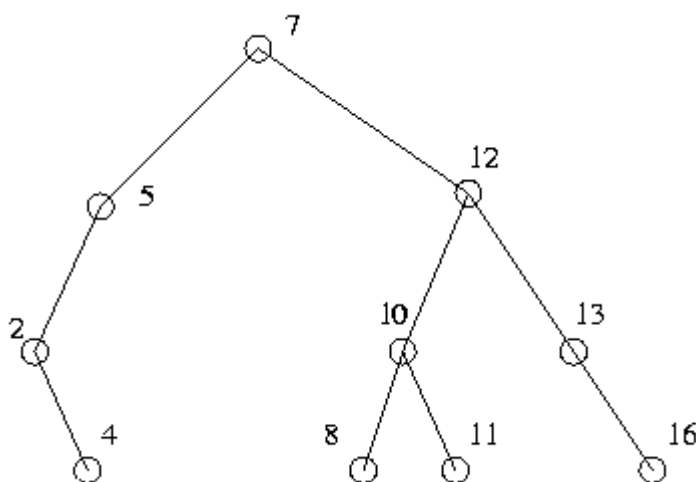
你的数据结构是有效的，比你如何改进你的数据结构更加重要。

## 基本结构

有五种基本数据结构：数组、链表、栈、队列和双向队列。你可能在之前已经见过这些东西；如果没有，可以问你老师。

## 二叉查找树

二叉查找树使您能够迅速搜索对象的集合（实型或整数）以确定给定的数值是否在集合中。基本上，二叉查找树是一个加权，有根的二叉规则树。这样的描述意味着树中的每个节点可能有一个右‘子节点和一个左’子节点（但双方或一方可能会丢失）。此外，每个节点都有与之相关联的对象，权值是对象在该节点的数值。二叉查找树有这样的属性：每个节点的左子树上的值小于该节点的值，每个节点的右子树的值大于或等于它。



每个节点通常由四个域构成，一个指针域指向左子节点，一个指针域指向右子节点，一个域存储权值，一个域存储对象。

## 为何二叉查找树可用？

给出一个具有  $N$  个对象的集合，二叉查找树查找一个对象只需  $O(\lg n)$  的时间，如果它不是一棵效率较低的树（如，一棵所有节点都没有左节点的树完成查找需要  $O(n)$  的时间）。此外，不同于存储在数组中，插入和删除对象也只需  $O(\lg n)$  的时间。

## 二叉查找树的延伸结构

有时在节点上加入一个指针域指向节点的父节点也是很有用的。这里有几种变化，以确保二叉查找树不会低效率：splay tree、红黑树、Treap、B 树、AVL 树。这些数据结构的实现有一定难度，而且随机构造的二叉查找树通常效率已经很高了，所以一般没必要实现这些算法了。

## 散列表（哈希表、HASH 表）

散列表通过一种可以快速查找的方式存储数据。假设有一个集合和一种数据结构，要求快速回答：“这个对象是否在数据结构中？”（如，这个单词是否在字典中？）散列表可以用少于二叉查找所用的时间完成这个功能。可以这样想：找到一个函数把集合的所有元素映射到一个整数从 1 到  $x$ （ $x$  大于集合里的元素个数）。定义一个从 1 到  $x$  的数组，把每个元素存储在元素经函数映射后的位置。之后确定一个元素是否在所给出的集合里，只要把它代入函数，看所映射的位置是否为空。如果不确信元素在上面，那么就去看一下是否和你存储的相一致。

举个例子，假设函数定义 3 个字符的单词上，有  $(\text{首字母} + (\text{中字母} * 3) + (\text{尾字母} * 7)) \bmod 11$  (定义 A=1, B=2, ..., Z=26)，单词有 'CAT', 'CAR', 和 'COB'。以 ASCII 为标准，则 'CAT' 在函数上的映射是 3, 'CAR' 在函数上的映射是 0, 'COB' 在函数上的映射是 7，故散列表如下： 0: CAR 1 2 3: CAT 4 5 6 7: COB 8 9 10 现在来看 'BAT' 在散列表中的情况，把 'BAT' 代入散列函数得到 2，散列表在 2 的位置是空的，所以它不在集合里。另一方面，把 'ACT' 代入散列函数得到 7，所以程序必须检验条目 'COB' 和 'ACT' 是否相同。考虑如下函数：

```
#define NHASH 8999          /* 保证它是素数 */
```

```
hashnum(p)
char *p;
{
    unsigned int sum = 0;
    for ( ; *p; p++)
        sum = (sum << 3) + *p;
    return sum % NHASH;
}
```

对每个输入，函数返回 0..NHASH-1 的某个数。它的输出是均匀随机的。这个简单的函数要求 NHASH 是素数。把上面的函数与下面的主程序合在一起：

```
#include

main() {
    FILE *in;
    char line[100], *p;
    in = fopen ("/usr/share/dict/words", "r");
    while (fgets (line, 100, in)) {
        for (p = line; *p; p++)
            if (*p == '\n') { *p = '\0'; break; }
        printf("%6d %s\n", hashnum(line), line);
    }
    exit (0);
}
```

就会产生类似这样的英文单词表（当然这得在 Linux 下运行）：

```
4645 aback
4678 abaft
6495 abandon
2634 abandoned
4810 abandoning
142 abandonment
7080 abandons
4767 abase
2240 abased
7076 abasement
4026 abasements
2255 abases
4770 abash
222 abashed
237 abashes
2215 abashing
```

```

361 abasing
4775 abate
2304 abated
3848 abatement

```

... ..

你可以看到函数产生的数字全部是均匀随机的，并且起码在这个例子中没有重复。当然，如果你有 `NHASH+1` 个单词，鸽笼定理证明了至少会有两个单词返回值相同，这就叫“冲突”。实际上，散列表使用链表技术解决了这种冲突，即相同值的单词列在同一区域。看看散列表究竟该怎么用吧。首先，建立散列表的链表结构，就像这样：

```

struct hash_f {
    struct hash_f *h_next;
    char *h_string;
    int   h_value;   /* 一些和字符串相关的值 */
                /* 完全自由选择如何使用或它已经存在 */
};

struct hash_f *hashtable[NHASH];   /* 每个链表的头指针 */
                /* 全局变量自动置空 */

```

产生如此散列表，比如用两个元素为父亲：

	hashtable	*hash_f	*hash_f
0		+-----+   +-----+	
	+-----+	*  -+	0
1		+-----+   +-----+	
	+-----+	'string1'       'abc def'	
2		*   -> +-----+ +-> +-----+	
	+-----+	val=1234       val=43225	
3		+-----+   +-----+	
	+-----+		
...			
8998			
	+-----+		

下面是插入操作：

```

struct hash_f *
hashinsert(p, val)
char *p;
int val;
{
    int n = hashnum(p);   /* 表的位置 */
    char *h = malloc( sizeof (struct hash_f) ); /* 新建散列元素 */

    /* 链接到表头: */
    h->h_next = hashtable[n];
    hashtable[n] = h;

    /* 可选值: */
    h->h_val = val;
}

```

```

/* 然后在链中找到适合元素放置的地方: */
h->h_string = malloc( strlen(p) + 1 );
strcpy (h->h_string, p);

return h;
}

```

下面是查找操作（若找到则返回其指针）:

```

struct hash_f *
hashlookup(p) {
    struct hash_f *h;

    int n = hashnum(p);          /* 初始位置 */

    for (h = hashtable[n]; h; h=h->h_next) /* 遍历链表 */
        if (0 == strcmp (p, h->h_string)) /* 匹配? 完成! */
            return h;
    return 0;                    /* 未找到 */
}

```

现在你可以快速地插入和查找了，平均需要(链表大小/2)次字符串比较。

### 为何散列表可用？

散列表只占用一点点内存，而程序查找元素几乎只要常数时间。通常程序要评估函数的价值，并且可能需要在表中比较一次或若干次【这句话翻得不好】。

### 散列函数

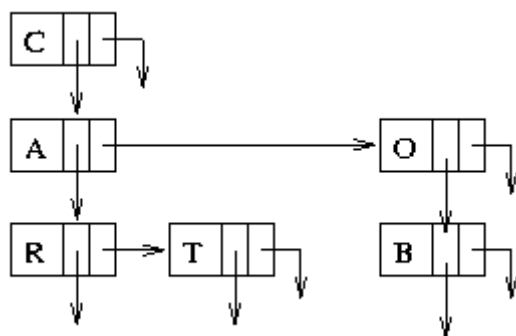
经常忘记的是，更精确点说，避免冲突的方法是找一个好的散列函数。举个例子，以单词前三个字母作为散列值显然很悲催。在此散列函数下，前缀 "CON" 会产生一个巨大的序列。你选择的函数要尽可能使不同的元素分配到不同的位置：

- 把巨大值用表的大小求模（选一个素数会干得特别好）。
- 素数是你的朋友。用它们相乘。
- 试着用小的 `changes` 映射到完全不同的地方。
- 不要想用两个小 `changes` 就能撤销映射到表外的函数（如一个变换）。
- 有一个研究的全域，要求创造一个“完美散列函数”以至于没有冲突，但是，完全产生均匀随机显然是要大量工作的；希望以后会有吧，起码现在木有啊。

### 散列变量

只用数值处理信息经常十分有用。比如当在一个大集合中搜寻一个小子集，用散列表处理已访问域，你可能想把它搜索的价值定位在哈希表中。甚至一个小散列表通过彻底减少搜寻空间都能改进运行时间。比如用首字母标识字典，你只需要找首字母就可以了。

### 一种特殊的树——“Trie”



这一节的图示：

简单来说，Trie 就是指有根树。它有不限制的出度(即一个节点可能有任意多个子节点)。一个节点的子节点存储在一个链表之中，所以一个节点有两个指针，下一个兄弟和第一个子节点（这样实际上就把一个普通的树转化为了一棵普通的二叉树）。Tries 存储一个序列集合。每条从根节点指向叶子节点的路径都对应集合中的一个元素。


比如，对于图中的 trie，指定了的集合是"CAR"，"CAT"，和 "COB"假定没有其它节点存在。要测定一个序列是否在集合之中，可以从根开始，在它的子节点中寻找序列的起始元素。如果没有匹配的，这个序列就不在集合中。否则，再在这个节点的子节点中寻找后面的元素，以此类推。trie 有几个不错的特点。如果一个字符串在列表中，检索的时间复杂度将不超过字符串的长度乘以一个节点的的最大子节点数。此外，比起其他的来说，这个数据结构往往可以使用较少的内存，因为前缀只出现一次（在我们的例子中，虽然'CAR'和'CAT'同时存在，但是只有一个'CA'的节点出现）。在一般情况下，对于已知前缀，查找字符序列（语句，多位数的号码，等等）的问题，trie 是很好用的。

### Trie 优化

一些常见的轻微改动有：

对于节点增加标记信息。如果列表里可能包含一个是其他单词前缀的词，你必须添加一个标志在每个节点上说'一个字到此为止'。如上例中，如果'CA'也作为单词出现在你的列表中，它会被标记。在链表中保持子节点有序。这增加了时间来建立树，但减少了查询时间。建立字符串节点；如果你有很多单词前缀一致但后缀独立，有时为其建立“特殊节点”更好。例如，存储“CARTING”，“COBBLER”，“CATCHING”这三个词，用“RTING”、“BBLER”、“TCHING”这三个节点显然可以节约内存。注意这同时增加了复杂性。

### 堆

堆（有时称为优先级队列）是一个完全二叉树，每个节点的值小于其两个孩子的值: 

### 堆的表示法

如果树(tree)是一层一层地从左到右地填入的（也就是说除了树最底层以外其他层都是完整的，最底层的元素是按从左到右填入的），那么这个堆(heap)能存储为一个数组，这个数组中元素的排列顺序是从根到底层，每层从左到右。这个例子中的堆可以表示为

3 5 9 6 12 13 10 8 11

在这个表示方法中，位于  $x$  的节点 (node) 的子节点 (children) 位于  $2x$  和  $2x+1$ ，(假设变址为 1)， $x$  的父节点 (parent) 是向下取整 (truncate)  $x/2$

### 堆中结点的插入和移动操作

将结点置于数组末端，接着交换结点与父节点的位置，直到找到一个合适的父节点。例如，插入数字 4 的过程中，堆数组（小根堆）变化如下：

3 5 9 6 12 13 10 8 11 4

3 5 9 6 4 13 10 8 11 12

3 4 9 6 5 13 10 8 11 12

删除一个结点相对来说也很简单。用数列末尾的结点取代要删除的，再进行调整：当结点的孩子比它小时，与较小的孩子交换。例如，删除数字 3 的过程：

11 5 9 6 12 13 10 8

5 11 9 6 12 13 10 8

5 6 9 11 12 13 10 8

5 6 9 8 12 13 10 11

### 如果我需要修改一个变量的值呢？

要把一个变量加大，则改变它的值，然后如果需要的话不断和它的父变量交换位置。T 要把一个变量的值减小，则改变它的值，然后如果需要的话和它的子变量中较小的交换位置。

### 堆的适用范围

对于动态的一组数询问最小值时，用堆处理十分便利。它结构紧凑，便于调整。Dijkstra 算法的堆优化就是一个很好的例子。

### Heap Variations

In this representation, just the weight was kept. Usually you want more data than that, so you can either keep that data and move it (if it's small) or keep pointers to the data. Since when you want to fiddle with values, the first thing you have to do is find the location of the value you wish to alter, it's often helpful to keep that data around. (e.g., node  $x$  is represented in location 16 of the heap).