


2020 年期中题目详解

填空题

1.自己编程运行一下或者手动模拟一下即可，答案为 292。

选择题

1.顺序存储与链式存储常用操作的时间复杂度比较：

 存取第 i 个元素的值分为两步：找到第 i 个元素 \rightarrow 存取这个元素的值。存取其前驱元素（也就是第 $\mathrm{i} - 1$ 个元素）的值也是类似的，不过在找第 $\mathrm{i} - 1$ 个的时候可以利用已经找到的第 i 个元素，可能不需要从头开始找了。

四个选项各个操作的时间复杂度比较如下：

	1. 找到第 i 个元素	2. 存取第 i 个元素的值	3. 找到第 $i - 1$ 个元素	4. 存储第 $i - 1$ 个元素的值
单链表	$O(n)$	$O(1)$	$O(n)$ （还要从头开始找）	$O(1)$
双链表	$O(n)$	$O(1)$	$O(1)$ （直接访问第 i 个元素的前驱字段）	$O(1)$
顺序表	$O(1)$	$O(1)$	$O(1)$	$O(1)$
循环单链表	$O(n)$	$O(1)$	$O(n)$ （还要从头开始找）	$O(1)$

综上，顺序表各个操作的效率最高。

\backslash newline\$

2.首先，在 `int *point, a = 4;` 语句中，定义了一个 `int` 类型的指针和一个 `int` 变量 `a`。在 `point = &a;` 中将 `point` 的值设成了变量 `a` 的地址。

然后介绍一下 C 语言中 `*` 和 `&` 符号的作用：

- `&`：逻辑运算符（按位与）；取地址符，`&a` 获得变量 `a` 的起始地址
- `*`：乘号；定义某一类型的指针，`int *point` 表示定义一个 `int` 类型的指针 `point`；解引用，如 `*point`，即从 `point` 指向的地址开始，取出四个字节，按照一个 `int` 来解释，获得一个 `int`（也就是给我一个地址，得到一个值）

可以看到，取地址和解引用是互逆的，即 `*(&a) = &(*a) = a`（虽然对于 `int` 变量 `a` 而言 `*a` 没有实际意义甚至是非法的，但是也符合语法，只是在逻辑上不对）。所以四个选项中：

- `a`（值）、`point`（地址）、`*&a`（值，相当于 `a`）
- `&(*a)`（值，相当于 `a`）、`&a`（地址）、`*point`（值，相当于 `a`）

- `*(&point)` (地址, 相当于 `point`)、`*point` (值)、`&a` (地址)
- `&a` (地址)、`&(*point)` (地址)、`point` (地址)

\$\newline\$

3.只要提到循环链表, 一定要知道一个关系: **尾结点的后继元素是头结点**。所以一定正确的选项是 A。

\$\newline\$

4.A 选项体现了数组和指针的重要差别之一, 数组名的值相当于数组的起始地址, 在初始化的时候已经被确定, 在后续过程中不可以被修改 (即不可以有 `int a[10]; a = b`), 但是指针可以 (即可以有 `int *a; a = b`)。

B 选项中 `sp` 是一个 `char` 类型的指针, 所以 `*sp` 得到的是一个 `char`, 不能把一个地址赋值给一个 `char` (B 选项的关键在于理解 `*` 的含义)。

C 选项正确, 定义了一个 `char` 类型的指针, 并把字符串常量 `BUAA` 的起始地址赋值给了这个指针。

D 选项不正确, 跟 B 一个问题。

补充一下, 对下面三行代码, 都是字符串初始化, 但是它们的行为不同:

```
char s1[] = "BUAA";
char s2[10] = "BUAA";
char *s3 = "BUAA";
```

- 第一行代码先推导出数组大小应该为 5, 然后再把几个字符依次拷贝进去 (`BUAA` + 空字符)
- 第二行代码把 `BUAA` + 空字符拷贝到数组的前 5 个位置
- 第三个行代码先在只读数据区中给 `"BUAA"` 分配了一片空间, 然后得到了一个起始地址, 再把这个起始地址给 `s3`

\$\newline\$

5.删除 `p` 的后继节点: 让 `p` 指向后继节点的后继节点 (相当于跳过后继节点了), 然后释放原后继节点的空间。故 B 正确。

\$\newline\$

6.可以参考 [作业讲解-第三次作业 \(链表\) -链表讲解](#) 一节, 顺序存储靠位置, 链式存储靠地址。所以选 C、A。

\$\newline\$

7.参考[选择题-1](#), 选 D。

\$\newline\$

8.区分结构类型、结构变量与结构成员:

- 结构类型 (类比为 `int x` 中的 `int`): `struct strutype`

- 结构成员（结构类型中定义的字段）：`a`、`b`
- 结构变量（类比为 `int x` 中的 `x`）：`var`

所以 C 不对，`var` 应该是结构变量名。

\$\newline\$

9.理解 `->` 运算符的作用，`->` 运算符实际上是一个简写形式，`p -> x` 等价于 `(*p).x`，也就是先解引用，再取出某字段。

A 正确，直接通过结构变量，利用 `.` 运算符取出某结构成员。

B 正确，利用某结构类型的指针变量，利用 `->` 运算符取出结构成员。

C 正确，`(*p)` 相当于先获得某结构变量，再用 `.` 运算符取出某结构成员。

D 不正确，因为 `.` 的优先级比 `*` 的优先级更高，所以会先执行 `p.x`，但是 `p` 是个指针，所以错误。

\$\newline\$

编程题

1.读完题目后，我们可以将这道题分为如下几步：

- 从输入中识别出标识符
- 将全部标识符记录下来
- 对标识符按照名称进行排序

首先，从输入中识别标识符，标识符的定义是由下划线或字母开头，并且由下划线、数字或字母组成。遇到这种**字符串模式匹配**的问题，尤其是看到注释的识别，我们就不难想到**状态转移的思想**。如果你还不了解状态转移的思想，可以先去看一下**作业讲解-程序设计基础练习-题目讲解**中第五道题的题解，这里的处理是大同小异的，可以设下面几个状态：

- 0：还没有读到标识符
- 1：正在读入一个标识符

作为一个练习，同学们可以试着自己画出这道题的状态转移图。

综上，处理识别出标识符的代码如下：

```
// 记录当前标识符的长度
int m = 0;
char ident[105];
for(int i = 0; i < len; i++) {
    // 如果在状态 0，而且读到了字母或下划线，转移到状态 1
    if(m == 0 && (str[i] == '_' || isalpha(str[i])))
        ident[0] = str[i], m = 1;
    // 如果在状态 1，且读入了字母或下划线，保持在状态 1
    else if (m != 0 && (str[i] == '_' || isalpha(str[i]) || isalnum(str[i])))
        ident[m] = str[i], m++;
    // 如果在状态 1，且读入了其它字符，转移回状态 0
    else if(m != 0) {
        ident[m] = 0;
    }
}
```

```

        // 记录下标识符
        printf("%s\n", ident);
        m = 0;
    }
}

```

在上面的代码中我们用 `m` 是否为 `0` 标识当前是在状态 `0` 还是在状态 `1`。同时因为最后以 `;` 结尾，所以我们不用在循环外再写一个判断 `if (m != 0)`（如果不以 `;` 结尾的话，可能会发生输入 `int hello` 这样的情况，`hello` 在循环内不会被处理）。

第二步，记录所有的标识符，也就是需要建立一个字符串的数组。这里我们有两种存储实现方式，二维数组或指针数组：



我这里用的是二维数组：

```

// 检查这个标识符之前是否出现过
for(int i = 0; i < cnt; i++)
    if(strcmp(identArr[i], ident) == 0) {
        flag = 1;
        break;
    }

// 如果这个标识符之前没出现过，就把它放在 identArr 数组中
if(flag == 0)
    strcpy(identArr[cnt], ident), cnt++;

```

第三步就是对字符串数组进行排序，也就是：

```

qsort(identArr, cnt, sizeof(identArr[0]), cmp);

```

这里根据字符串数组实现的不同（指针数组 or 二维数组），`cmp` 的写法也不相同。



用二维数组对字符串排序时，相当于各个字符串实实在在地存在数组中，每次都真正交换了几个字符串；用指针数组对字符串排序时，相当于数组中只存了几个地址值，这几个地址值标志了真正的各个字符串存在哪，排序时仅仅交换了几个地址值，真正的字符串并没有交换。

我们用的是二维数组的方式，`cmp` 这里就这么写：

```

int cmp(const void *a, const void *b) {
    char *x = (char *)a;
    char *y = (char *)b;
    return strcmp(x, y);
}

```

这道题的完整代码可以参考 [往年期中/参考代码/2020_1.c](#)。

\$\newline\$

2.这道题跟我们链表那次作业的第三题基本一样，就是把最佳适应策略换成了首次适应策略，感觉还更简单了。

首先不断地读入结点的位置和大小，创建一个链表。全部读入后，利用尾结点的后继是头结点，创建一个循环链表。

然后，每次读入一个空间，从当前位置，找首次适应这个空间的块，**把指定的空间分配出去，并更新当前位置**：

```
// alloc 函数分配空间，并返回新的`当前位置`  
while((sz = readInt()) != -1)  
    now = alloc(now, sz);
```

在 `alloc` 中，我们要从当前位置开始，遍历一圈链表，找到第一个 \geq 指定大小的结点。我们可以使用 `do while` 语句，其与 `while` 的区别仅仅在于 `do while` 至少会执行一次，这样我们就不用特判只有一个结点的情况了：

```
nptr p = now, ret = NULL;  
do {  
    if(p -> sz >= sz) {  
        ret = p;  
        break;  
    }  
    p = p -> next;  
} while(p != now);
```

当找到这个结点后，我们要根据这个结点的容量来进行后续处理。如果这个结点的容量刚好等于 `sz`，我们还要删除这个结点，由于涉及到了链表的删除操作，所以我们一开始可以维护一个双向链表（当需要删除时，最好再维护一下每个结点的前驱元素，这样就不用从头开始遍历了）：

```
// alloc 函数的剩余部分  
  
if(ret != NULL) {  
    // 大小恰好相等，按照规则 2，移除该结点，并返回当前位置为下一空闲块  
    if(ret -> sz == sz) {  
        ret -> next -> prev = ret -> prev;  
        ret -> prev -> next = ret -> next;  
        if(ret -> next == ret)  
            return NULL;  
        return ret -> next;  
    } else {  
        // 大于申请空间，按照规则 3，修改 sz 字段即可，并返回当前位置为这个空闲块
```

```
        ret -> sz -= sz;  
        return ret;  
    }  
}  
// 没有合适大小的空闲块, 按照规则 4, 当前位置不变  
return now;
```

这道题的完整代码可以参考 [往年期中/参考代码/2020_2.c](#)。