# 数据结构第二次作业

学号：2022212408 姓名：胡宇杭

2023 年 10 月 12 日

## 1 选择题

1. 在 N 个结点的顺序表中，算法的时间复杂度为 ***O(1)*** 的操作是 (**A**)

   A. 访问第 $\boldsymbol{i}$ 个结点 ($1 \leq i \leq N$) 和求第 $\boldsymbol{i}$ 个结点的直接前驱 ($2 \leq i \leq N$)

   B. 在第 $\boldsymbol{i}$ 个结点后插入一个新结点 ($1 \leq i \leq N$)

   C. 删除第 $\boldsymbol{i}$ 个结点 ($1 \leq i \leq N$)

   D. 将 $\boldsymbol{N}$ 个结点从小到大排序

2. 若某表最常用的操作是在最后一个结点之后插入一个结点或删除最后一个结点。则采用哪种存储方式最节省运算时间 (**D**)

   A. 单链表

   B. 双链表

   C. 单循环链表

   D. 带头结点的双循环链表

3. 将两个结点数都为 N 且都从小到大有序的单向链表合并成一个从小到大有序的单向链表，那么可能的最少比较次数是 (**B**)

   A. **1**

   B. **N**

   C. **2N**

   D. **NlogN**

4. 已知表头元素为 ***c*** 的单链表在内存中的存储状态如下表，现将 ***f*** 存放于 **1014H** 处，并插入到单链表中，若 ***f*** 在逻辑上位于 ***a*** 和 ***e*** 之间，则 ***a***、***e***、***f*** 的“链接地址”依次是 (**D**)

   A. **1010H**, **1014H**, **1004H**

   B. **1010H**, **1004H**, **1014H**

   C. **1014H**, **1010H**, **1004H**

   D. **1014H**, **1004H**, **1010H**

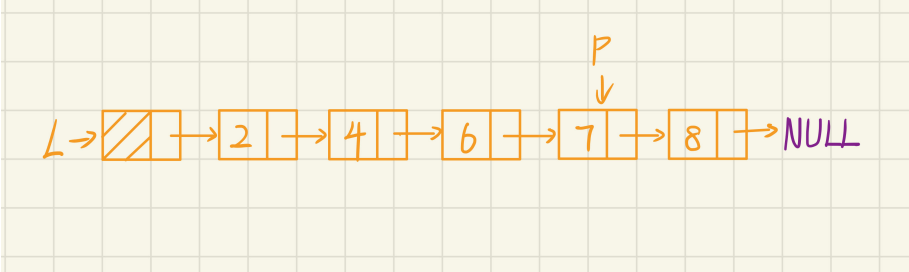
| 地址 | 元素 | 链接地址 |
|---|---|---|
| 1000H | a | 1010H |
| 1004H | b | 100CH |
| 1008H | c | 1000H |
| 100CH | d | NULL |
| 1010H | e | 1004H |
| 1014H | | |

# 2 简答题

1. 画出执行下列各行语句后最终的各指针及链表的示意图

```
1  L = (LinkList)malloc(sizeof(LNode));
2  p = L;
3
4  for (i = 1; i <= 4; i ++ )
5  {
6      P->next = (LinkList)malloc(sizeof(LNode));
7      P = P->next; P->data = i * 2 - 1;
8  }
9
10 P->next = NULL;
11 for (i = 4; i >= 1; i -- )
12     Ins_LinkList(L,i + 1, i * 2);
13
14 for (i = 1; i <= 3; i ++ )
15     Del_LinkList(L,i);
```

最终各指针及链表的示意图如下图所示




2. 已知 ***P*** 结点是某双向链表的中间结点，写出以下操作的语句序列

A. 在 ***P*** 结点后插入 ***S*** 结点的语句序列；

```
S->next = P->next;
S->prior = P;
```

```
P->next->prior = S;
P->next = S;
```

B. 在 ***P*** 结点前插入 ***S*** 结点的语句序列；

```
S->prior = P->prior;
S->next = P;
P->prior->next = S;
P->prior = S;
```

C. 删除 ***P*** 结点的直接后继结点的语句序列；

```
P->next = P->next->next;
free(P->next->prior);
P->next->prior = P;
```

D. 删除 ***P*** 结点的直接前驱结点的语句序列；

```
P->prior = P->prior->prior;
free(P->prior->next);
P->prior->next = P;
```

E. 删除 ***P*** 结点的语句序列。

```
P->prior->next = P->next;
P->next->prior = P->prior;
free(P);
```

3. 简述以下算法的功能

A. 第一小题

```
1 // L是无头结点的单链表
2 Status A(LinkedList &L)
```

```
{
    if(L && L->next){
        Q = L; L = L->next; P = L;
        while(P->next) P = P->next;
        P->next = Q; Q->next = NULL;
    }
    return OK;
}
```

在一个至少有两个结点的单链表中，将第一个结点断开，并将其添加到链表的尾部

B. 第二小题

```
void BB(LNode *s, LNode *q)
{
    p = s;
    while (p->next != q) p = p->next;
    p->next = s;
}// BB

void AA(LNode *pa, LNode *pb)
{
    // pa和pb分别指向不带头结点单循环链表中的两个结点
    BB(pa, pb);
    BB(pb, pa);
}// AA
```

假设 $pa$ 的前驱结点为 $pa_{prior}$，$pb$ 的前驱结点为 $pb_{prior}$

则该算法的功能为将该单循环链表分裂为

$pa$->...->$pb_{prior}$->$(loop)$ 和 $pb$->...->$pa_{prior}$->$(loop)$ 两个单循环链表

# 3 基础编程练习 (无需提交)

1. 顺序表的基础操作部分代码 git 地址为：`https://gitee.com/rainysun/sqlist_demo`，请在此基础上补充编写并调试完成相关操作

   A. 按值查找，**LocateElem(SqList L, ElemType e)**

   B. 删除，**ListDelete(SqList &L, int i, ElemType &e)**

   C. 划分，**Part(SqList &L)**

   D. 合并，**Merge(SqList A, Sqlist B, Sqlist &C )**

2. 链表的基础操作部分代码 git 地址为:`https://gitee.com/rainysun/linklist_demo`，请在此基础上补充编写并调试完成相关操作

   A. 删除第 **i** 个元素，**ListDelete_L(LinkList L, int i, ElemType &e)**

   B. 尾插法建立单链表，**Create_TL(LinkList &L, int n)**

   C. 两个有序单链表合并，**Merge(LinkList A, LinkList B, LinkList &C)**

   D. 单链表逆置，**Reverse(LinkList &C)**

# 4 算法设计题

1. 从一个给定的顺序表 **L** 中删除值在 **[x, y]** $(x \leq y)$ 之间的所有元素，要求以较高的效率来实现，并分析你的算法时间复杂度。(提示：移动位置一步到位)

   (a) 定义两个指针 ***p***、***q***，***p*** 指针用来遍历原顺序表，***q*** 指针指向新顺序表的末尾，***p*** 的每一次迭代中，我们检查 **elem[p]** 的值是否合法，如果合法，就将其加入新顺序表，并令 ***q*** 后移一位。操作完成后，新顺序表中存放的即是结果，同时，由于 ***q*** 指针一值指向表尾后一位，可以用来更新表长

   (b) 注意到每一次操作后，两指针的位置关系有 $q \leq p$，所以我们可以直接在原顺序表上进行操作，同时保证不会意外将合法元素移除。

   由于只需要一次遍历即可求解出答案，因此时间复杂度为：***O(n)***

```
1  void DeleteRange(SqList &L, ElemType x, ElemType y)
2  {
3      if (y < x) return;
4
5      int q = 0;
6      for (int p = 0; p < L.length; p ++ )
7          if (L.elem[p] < x || L.elem[p] > y)
8          {
9              L.elem[q] = L.elem[p];
10             q ++ ;
11         }
12
13     L.length = q;
14 }
```

2. 设计一个空间复杂度为 ***O(1)*** 的算法 **shift(SqList L, int k)** 将顺序表 **L** 中的元素整体循环左移 **k** 位，要求以较高的效率来实现，并分析在如下示例的 **10** 个数据，循环左移 **4** 位时这 **10** 个数据总共移动的次数。

   示例数据：

顺序表 **L** 初始数据为：**1, 2, 3, 4, 5, 6, 7, 8, 9, 10**

整体循环左移 **4** 位后为：**5, 6, 7, 8, 9, 10, 1, 2, 3, 4**

当我们将最终得到的序列反转后，不难发现其为 **(1, k)** 和 **(k + 1, length)** 区间各反转一次的结果，以下给出证明：

设元素 $e$ 的位置为 $x$ $(1 \le x \le length)$，由题意可知，最终其位置应为 **(x - k) % length**

可以分为 $x \le k$ 和 $x > k$ 两种情况讨论

- $x \le k$ 时，第一次反转后其位置为 $k - x + 1$，整体反转后其位置为 $length - (k - x + 1) + 1 = length - k + x = (x - k) \ \% \ length$；
- $x > k$ 时，第一次反转后其位置为 $length - x + 1$，整体反转后其位置为 $length - (length - x + 1) + 1 = x - k = (x - k) \ \% \ length$

```
1  void reverse(SqList &L int start, int end)
2  {
3      while (start < end)
4      {
5          // swap(L.elem[start], L.elem[end])
6          ElemType temp = L.elem[start];
7          L.elem[start] = L.elem[end];
8          L.elem[end] = temp;
9          start ++ ; end -- ;
10     }
11
12     return;
13 }
14
15 void shift(SqList &L, int k)
16 {
17     reverse(L, 0, k - 1);
18     reverse(L, k, L.length - 1);
19     reverse(L, 0, L.length - 1);
20 }
```

下面分析图示数据所需移动次数

我们首先反转了前 **4** 个元素，每个元素移动 **1** 次，共移动 **4** 次；再反转后 **6** 个元素，共移动 **6** 次；最后我们反转整个序列，共移动 **10** 次

综上，我们一共移动了 **20** 次

3. 设计一个算法删除一个单链表倒数第 **k** 个结点

定义两个指针 ***p*** 和 ***q***，首先先让 ***p*** 指针走到第 ***k*** 个结点，然后再让 ***p*** 指针和 ***q*** 指针一起走到 ***p*** 指针到达表尾时，此时 ***q*** 指针的位置刚好在倒数第 $k+1$ 个结点的位置

```
void ListDel_L(LinkList L, int k)
{
    if (!L || k <= 0) return;

    LNode *p = L, *q = L;
    for (int i = 0; i < k; i ++ )
        if (!p->next) return;
        else p = p->next;

    while (p->next)
    {
        p = p->next;
        q = q->next;
    }

    LNode *temp = q->next;
    q->next = q->next->next;

    free(temp);

    return;
}
```

4. 设计一个算法检测一个单链表 **L** 上是否因为某种错误操作而产生了环

判断成环的一个经典算法是 ***Floyd* 判圈算法**，定义一个快指针和一个慢指针，慢指针每次移动一位，快指针每次移动两位，如果链表成环，快指针一定会在某一时刻与慢指针指向同一结点

```
1  bool check_circle(LinkList L)
2  {
3      if (!L) return false;
4
5      LNode *slow = L, *fast = L;
6
7      while (fast->next)
8      {
9          fast = fast->next->next;
10         slow = slow->next;
11
12         if (fast == slow)
13             return true;
14     }
15
16     return false;
17 }
```