中国科学技术大学
University of Science and Technology of China

2021 春数据结构第二次习题课

范围：作业 4-7 及实验

中国科学技术大学，计算机科学与技术系

2021 年 11 月 13 日

大纲

4.22(4) 假设以块链结构表示串。试编写将串 s 插入到串 t 中某个字符之后的算法（若串 t 中不存在此字符，则将串 s 联接在串 t 的末尾）。

和线性表的链式存储结构相类似，也可采用链表方式存储串值。由于串结构的特殊性——结构中的每个数据元素是一个字符，则用链表存储串值时，存在一个“结点大小”的问题，即每个结点可以存放一个字符，也可以存放多个字符。例如，图 4.2(a)是结点大小为 4(即每个结点存放 4 个字符)的链表，图 4.2(b)是结点大小为 1 的链表。当结点大小大于 1 时，由于串长不一定是结点大小的整倍数，则链表中的最后一个结点不一定全被串值占满，此时通常补上“#”或其他的非串值字符(通常“#”不属于串的字符集，是一个特殊的符号)。

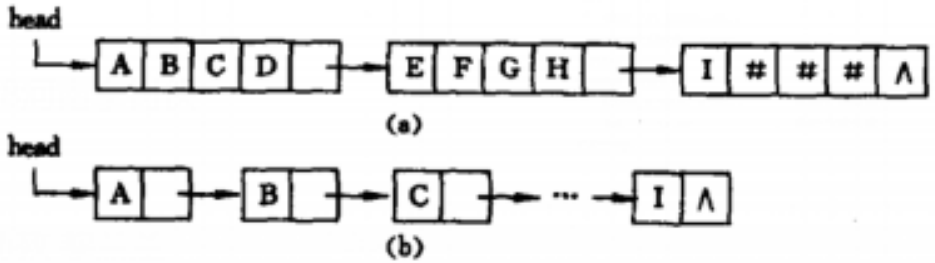


图 4.2 串值的链表存储方式

(a) 结点大小为 4 的链表；(b) 结点大小为 1 的链表

4.22(4) 假设以块链结构表示串。试编写将串 s 插入到串 t 中某个字符之后的算法（若串 t 中不存在此字符，则将串 s 联接在串 t 的末尾)。

为了便于进行串的操作，当以链表存储串值时，除头指针外还可附设一个尾指针指示链表中的最后一个结点，并给出当前串的长度。称如此定义的串存储结构为**块链结构**，说明如下：

```
//== == == 串的块链存储表示 == == ==
#define  CHUNKSIZE  80        // 可由用户定义的块大小
typedef  struct Chunk {
    char  ch[CHUNKSIZE];
    struct Chunk   * next;
}Chunk;
typedef struct {
    Chunk * head, * tail;     // 串的头和尾指针
    int     curlen;           // 串的当前长度
}LString;
```

```
void LString_Concat(LString &t,LString &s,char c)//用块链存储结构,把串 s 插入
到串 t 的字符 c 之后
{
  p=t.head;
  while(p&&!(i=Find_Char(p,c))) p=p->next; //查找字符 c
  if(!p) //没找到
  {
    t.tail->next=s.head;
    t.tail=s.tail; //把 s 连接在 t 的后面
```

```
}
else
{
  q=p->next;
  r=(Chunk*)malloc(sizeof(Chunk)); //将包含字符 c 的节点 p 分裂为两个
  for(j=0;j<i;j++) r->ch[j]='#'; //原结点 p 包含 c 及其以前的部分
  for(j=i;j<CHUNKSIZE;j++) //新结点 r 包含 c 以后的部分
  {
    r->ch[j]=p->ch[j];
    p->ch[j]='#'; //p 的后半部分和 r 的前半部分的字符改为无效字符'#'
  }
  p->next=s.head;
  s.tail->next=r;
  r->next=q; //把串 s 插入到结点 p 和 r 之间
}//else
t.curlen+=s.curlen; //修改串长
s.curlen=0;
}//LString_Concat
```

```
int Find_Char(Chunk *p,char c)//在某个块中查找字符 c,如找到则返回位置是第几
个字符,如没找到则返回 0
{
  for(i=0;i<CHUNKSIZE&&p->ch[i]!=c;i++);
  if(i==CHUNKSIZE) return 0;
  else return i+1;
}//Find_Char
```

课堂布置题：按扩展线性链表重写求广义表的深度的递归方法

```
1  int GListDepth(GList L){
2      if(!L) return 0; //扩展线性链表空表也存在一个表结点
3      for(max = 0, pp = L; pp; pp = pp -> tp){ //扩展线性链表的原子结点也有后继
4          if(L->tag == LIST){
5              dep = GListDepth(pp->hp)+1;
6              if(dep > max) max = dep;
7          }
8      }
9      return max;
10 }
```

5.32(4) 试编写判别两个广义表是否相等的递归算法

```
1  Status GListCompare(GList L, GList T){
2      if(!L && !T) return OK; // 二者均为空
3      if(L && T && L->tag == T->tag){ // 非空且类型相同
4          if(L->tag == ATOM)
5              if(A->atom == T->atom) return OK;
6          else
7              if(GListCompare(L→ptr.hp, T→ptr.hp) &&
       GListCompare(L→ptr.tp, T→ptr.tp))
8                  return OK;
9      }
10     return ERROR;
11 }
```

大纲

5.34(5) 试编写递归算法，逆转广义表中的数据元素。例如：将广义表

$$(a,((b,c),()),(((d),e),f))$$

逆转为：

$$((f,(e,(d))),((),(c,b)),a)$$

解答：这一题的递归关系比较简单——此处以单个元素 x（可以是表或者原子）作递归对象为例，我们可以将一个复杂的表拆成组成它第一层表结构的每一个元素，分而治之

- 如果 x 是表，那么它的逆转就是将其内部的所有组成元素逆转后再倒序封装成表
- 如果 x 是原子，那么它的逆转就是其本身

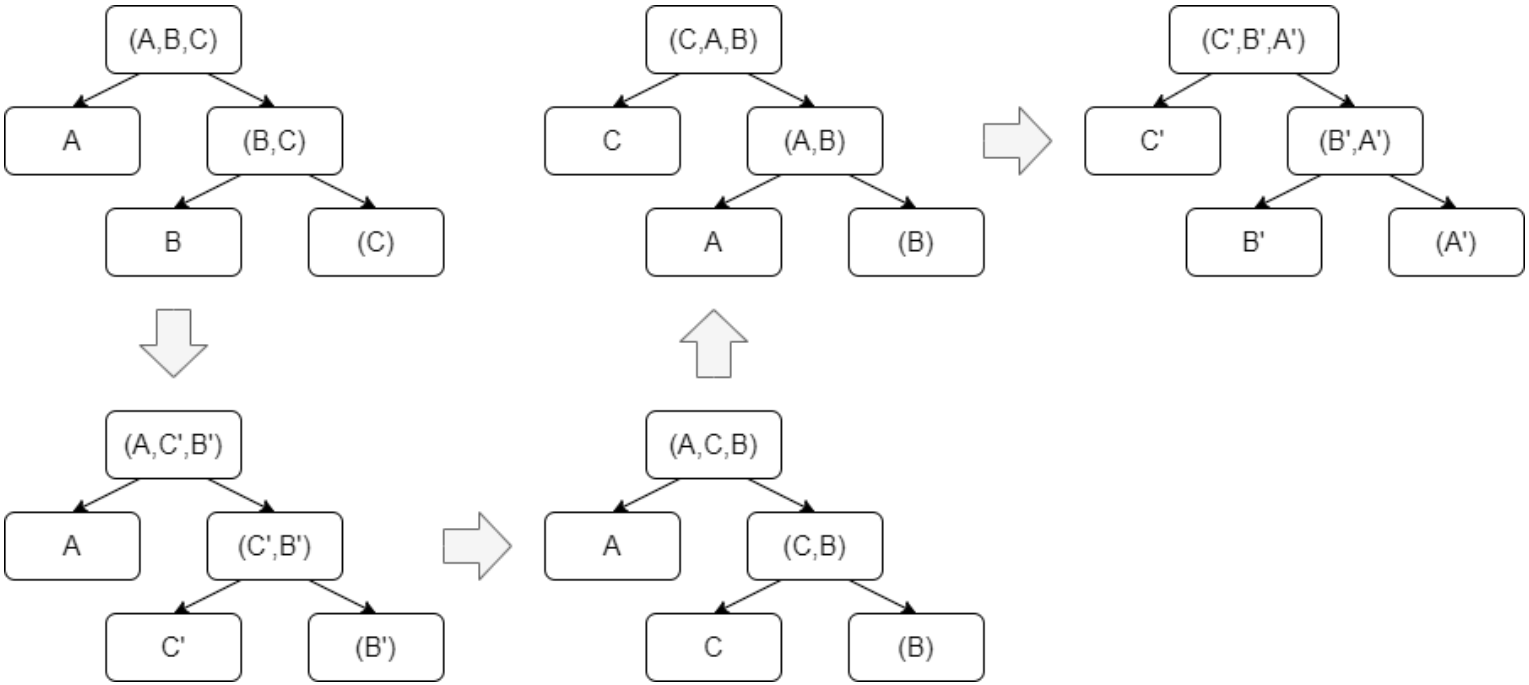
伪代码：

```
1  status T5_34(GList &L){
2      if(L.type == ATOM) return OK; // 原子无需操作
3      else{ // 如果是表就对下一层所有元素递归
4          for each element in L do
5              T5_34(element);
6          L.reverse(); // 倒排表 L
7      }
8      return OK;
9  }
```

该算法在以扩展线性链表为存储结构的情况下实现较为简洁，但如果是头尾链表的话，可以尝试另一种递归策略——

以广义表单个元素 x（可以是表或者原子）作递归对象，我们可以将一个复杂的表拆成头和尾两部分，分而治之

- 如果 x 是表，那么它的逆转就是将其表头和表尾分别逆转，随后重新拼合出新的表头和表尾
- 如果 x 是原子，那么它的逆转就是其本身



```
1  status T5_34(GList &L){ // 伪代码
2      if(L.type == ATOM || !L) return OK;
3      else{ // 如果是表就对下一层做拆分
4          T5_34(L.tail); // 递归处理表尾
5          if(!L.tail){
6              T5_34(L.tail.head);
7              T5_34(L.tail.tail);
8          }
9          L.tail.head <-> L.head; // 空的情况另作考虑
10         T5_34(L.head);
11         T5_34(L.tail);
12     }
13     return OK;
14 }
```

6.36(3) 若已知两棵二叉树 $B1$ 和 $B2$ 皆为空，或者皆不空且 $B1$ 的左、右子树和 $B2$ 的左、右子树分别相似，则称二叉树 $B1$ 和 $B2$ 相似。试编写算法，判别给定两棵二叉树是否相似。

解答： 这一题可直接按照题干中相似的定义来编写递归算法

```
1  bool T6_36(BiTree B1, BiTree B2){ // 伪代码
2      if(B1 && B2)
3          return(T6_36(B1.lchild, B2.lchild) && T6_36(
       B1.rchild, B2.rchild));
4      else if(B1 || B2)
5          return FALSE;
6      else
7          return TRUE;
8  }
```

6.43(3) 编写递归算法，将二叉树中所有结点的左、右子树相互交换。

解答：这一题可直接对左右子树分而治之递归处理

```
1  status T6_43(BiTree T){ // 伪代码
2      if(!T) return OK;
3      T6_43(T.lchild);
4      T6_43(T.rchild);
5      T.lchild <-> T.rchild;
6      return OK;
7  }
```

6.48(5) 已知在二叉树中，*root 为根结点，*p 和 *q 为二叉树中两个结点，试编写求距离它们最近的共同祖先的算法。

解答：最近共同祖先，又称（Lowest Common Ancestor）。这一题的解法很多，罗列几种

- 从 *p 开始往上一层层查找并检查是不是 *q 的祖先，查到满足条件的第一个结点就是 LCA
- 查找出从根通往 *p 和 *q 的路径，比较两条路径，分岔点处就是 LCA
- 计算 *p 和 *q 的深度，抹平差距后同时往上查找，相遇处就是 LCA
- 递归遍历，检查左右子树是否能同时查到 *p 或 *q，该结点正是 LCA

* Tarjan's off-line lowest common ancestors algorithm

这里拿递归遍历的算法做一个示例

```
1  BiTNode* LCA(BiTree root, BiTNode* p, BiTNode* q){
2      if( !root || root==p || root==q ) return root;
3      left = LCA(root.lchild, p, q);
4      right = LCA(root.rchild, q, q);
5      if(left)
6          if(right) return root;
7          else return left;
8      else
9          if(right) return right;
10         else return NULL;
11 }
```

大纲

6.38(4) 试利用栈的基本操作写出后序遍历的非递归算法（提示：为分辨后序遍历时两次进栈的不同返回点，需在指针进栈时同时将一个标志进栈）。

代码实现：

```
1  int PostOrderTraverse(BiTree T) {
2      SqStack S;
3      BiTree p;
4      SElemType e;
5      int StackMark[100] = {0}; // 标记栈，设置各结点访
       问标记（初始化为0）
6      int k;
7      if(T == NULL) return 0;
8      InitStack(&S);
9      p = T;
10     k = -1;
```

```
11      while(TRUE) {
12          while(p) {
13              Push(&S, p);
14              k++;
15              StackMark[k] = 1; // 设置第一次访问的标记
16              p = p->lchild;
17          }
18          // p为空但栈不为空
19          while(!p && !StackEmpty(S)) {
20              GetTop(S, &p);
21
22              // 已访问过一次，当前是第二次访问
23              if(StackMark[k] == 1) {
24                  StackMark[k] = 2;
25                  p = p->rchild;
```

```
26
27                  // 已访问过两次，当前是第三次访问
28              } else {
29                  printf("%c ", p->data);
30                  Pop(&S, &e);
31                  StackMark[k] = 0;
32                  k--;
33                  p = NULL;
34              }
35          }
36          if(StackEmpty(S)) break;
37      }
38      return 1;
39  }
```

6.47(1) 编写按层次顺序（同一层自左至右）遍历二叉树的算法。

解答：代码如下

```
1  void LevelOrderTraverse(BiTree T) {
2      Queue q;
3      if (T == NULL) return; // 树为空，直接返回
4      Initqueue(q);
5      Inqueue(q, T); // 先将根节点入队
6      while (!QueueEmpty(q)){
7          p = Dequeue(q) // 出队保存队头并访问
8          if (p->lchild) // 将出队结点的左子树根入队
9              Inqueue(q, p->lchild);
10         if (p->rchild) // 将出队结点的右子树根入队
11             Inqueue(q, p->rchild);
12     }
13 }
```

6.58(4) 试写一个算法，在中序全线索二叉树的结点 *p 之下，插入一棵以结点 *x 为根、只有左子树的中序全线索二叉树，使 *x 为根的二叉树成为 *p 的左子树。若 *p 原来有左子树，则令它为 *x 的右子树。完成插入之后的二叉树应保持全线索化特性。

```
1  Status Ans6_58(BiThrTree p, BiThrTree x, BiThrTree
       Thrx) {
2      BiThrTree pPre; // 结点 p 的前驱
3
4      BiThrTree xFirst; // 树 x 中序序列的第一个结点
5      BiThrTree xLast; // 树 x 中序序列的最后一个结点
6
7      BiThrTree lt; // p 的左子树
8      BiThrTree ltFirst; // 子树 lt 中序序列的第一个结点
```

```
9
10      if(p==NULL || x==NULL)
11          return ERROR;
12
13      // x结点不允许存在右孩子
14      if(x->RTag==Link)
15          return ERROR;
16      // 获取树x的中序序列的第一个结点和最后一个结点
17      xFirst = Thrx->lchild;
18      // 如果存在左子树，一直向左遍历
19      while(xFirst->LTag==Link){
20          xFirst = xFirst->lchild;
21      }
22      xLast = Thrx->rchild;
```

```
23      if(p->LTag==Thread) { // 若结点 p 无左子树
24          pPre = p->lchild; // 直接获取 p 的前驱
25          p->LTag = Link; // 修改p的左线索为左孩子
26          p->lchild = x; // 插入子树x
27          xFirst->lchild = pPre; // 重置 xFirst 的左线索
28          xLast->rchild = p; // 重置 xLast 的右线索
29      } else { // 若结点 p 有左子树
30          lt = p->lchild; // 指向结点 p 的左子树
31          ltFirst = lt; // 查找子树 lt 中序序列首个结点
32          // 如果存在左孩子，则一直向左遍历
33          while(ltFirst->LTag==Link)
34              ltFirst = ltFirst->lchild;
35          x->RTag = Link; // 将lt插入为x的右子树
36          x->rchild = lt;
```

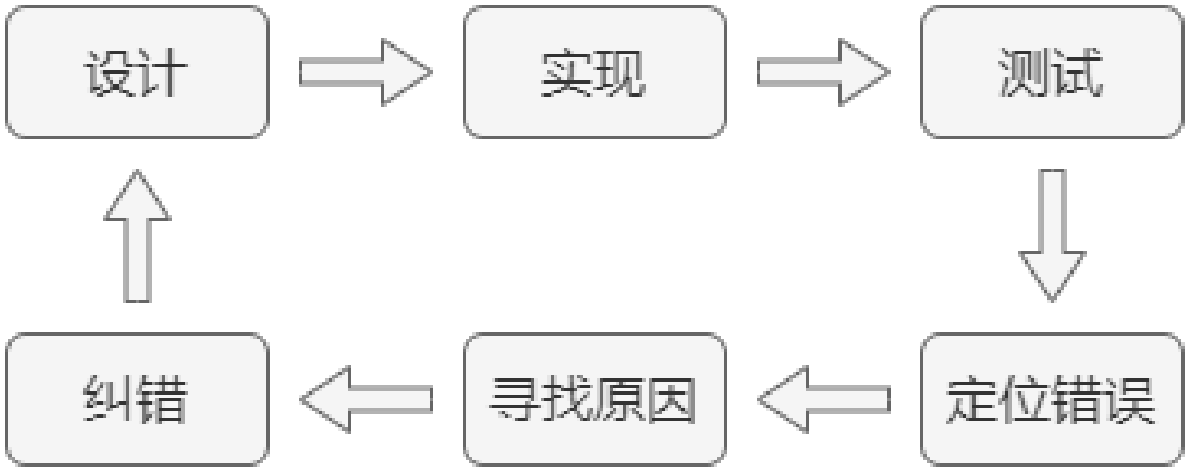
```
37          /*
38           * 这里需要一点:
39           * 左子树lt变成子树x的根结点的右子树时,
40           * lt中序序列上最后一个结点的后继线索不会变
41           */
42          xFirst->lchild = ltFirst->lchild; // 接管子树lt的左线索
43          ltFirst->lchild = x; // 子树lt的左线索指向x
44          p->lchild = x;  // 更新p的左子树
45      }
46      Thrx->lchild = Thrx->rchild = Thrx; // 将x从Thrx上移除
47      return OK;
48  }
```

大纲

附加项

- 【实现可指定的任意基本符号单元大小的 Huffman 压缩/解压缩算法】
  - 当基本符号单元的大小不再是字节的时候，可能会出现原文件本身大小并非基本符号单元大小整数倍的情况，一种可能的解决方案是将原文件填充到合适的大小，并通过为压缩文件的头部增添额外的信息帮助你在解压时去掉填充的部分。
  - 在设计的初期，可以先从较小的测试文件开始，或者自备一些简单的数据进行初步测试
- 【实现可指定的任意多元 Huffman 压缩/解压缩算法】为使该树为满的 n 叉树，可以在基本符号单元集合中增加权重为 0 的额外的占位符，来为你的算法设计带来方便。

我们基于自己的经验和这几次实验的情况，选择了一些易上手的技巧来与同学们分享，为方便起见，这里先把平时做实验敲代码的过程分成几个阶段



【声明!】这仅仅是助教们结合自己编程体验和近几次实验情况的个人经验之谈，为一些不知道该如何 debug 同学聊作参考，与实际开发中成熟的设计-测试流程完全是天壤之别，想系统性地学习、了解这方面知识的同学请务必以相关的权威书籍资料为准!!!

设计

大概没法速成，推荐有意识地去锻炼和改进

- 如非时间急迫，在着手写代码前，最好先把设计思路理顺
  - 模块图、流程图、类图、活动图、状态图
- 有必要的时候，在设计时为自己后续的 debug 留个方便

实现

- 培养良好的代码习惯
  - 命名、注释、可读性
  - 适当使用 IDE 避免错误，保持规范
- 妥善管理文件
  - 使用合适的版本管理工具，如 Git
  - 适当组织文件结构，即时分类

测试

- 寻找边界条件测试例
- 若有必要，可先针对单个模块进行测试

定位错误

- 最小错误示例
  - 减少循环次数，去除不必要的分支
  - 减小数据量
- 结合流程图或者根据程序分支来定位错误
- 检查输入输出、函数的出入口等交界部分

查找原因

- 程序跟踪
  - 善用 GDB
  - 初上手可以借助一些 IDE （自动配置和可视化调试），如 VS（安装即用），VSCode+ 插件（需要配置）
- 寻求其他人的帮助

纠错

- 记录 debug 日志，可以结合 Git

谢谢!