阅读：12,693　　作者：解学武

广义表的深度和长度（C语言）详解

前面学习了广义表及其对应的存储结构，本节来学习如何计算广义表的深度和长度，以及如何编写相应的 C 语言实现程序。

广义表的长度

广义表的长度，指的是广义表中所包含的数据元素的个数。

由于广义表中可以同时存储原子和子表两种类型的数据，因此在计算广义表的长度时规定，广义表中存储的每个原子算作一个数据，同样每个子表也只算作是一个数据。

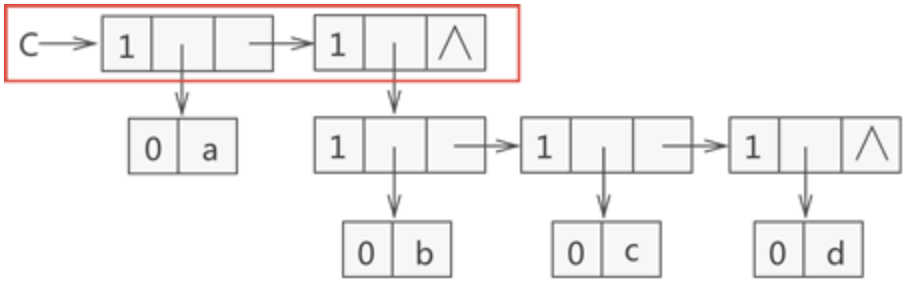
例如，在广义表 {a,{b,c,d}} 中，它包含一个原子和一个子表，因此该广义表的长度为 2。

再比如，广义表 {{a,b,c}} 中只有一个子表 {a,b,c}，因此它的长度为 1。

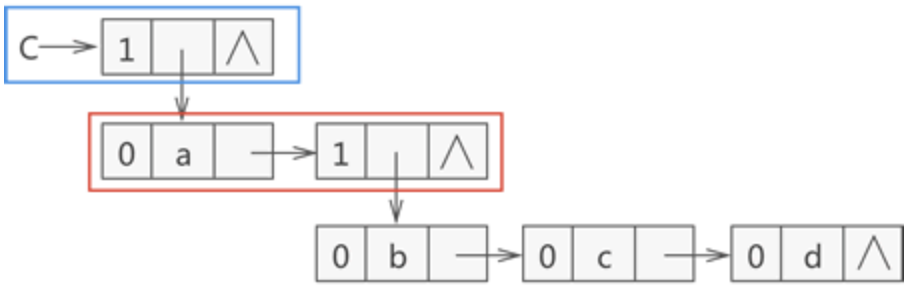
前面我们用 LS={a_1,a_2,...,a_n} 来表示一个广义表，其中每个 a_i 都可用来表示一个原子或子表，其实它还可以表示广义表 LS 的长度为 n。

> 广义表规定，空表 {} 的长度为 0。

在编程实现求广义表长度时，由于广义表的存储使用的是链表结构，且有以下两种方式（如图 1 所示）：



a) 广义表存储结构示意图



b) 另一种广义表存储结构示意图

图 1 存储 {a,{b,c,d}} 的两种方式

对于图 1a) 来说，只需计算最顶层（红色标注）含有的节点数量，即可求的广义表的长度。

同理，对于图 1b) 来说，由于其最顶层（蓝色标注）表示的此广义表，而第二层（红色标注）表示的才是该广义表中包含的数据元素，因此可以通过计算第二层中包含的节点数量，即可求得广义表的长度。

由于两种算法的实现非常简单，这里只给出计算图 1a) 中广义表长度的 C 语言实现代码：

```
#include <stdio.h>
#include <stdlib.h>
typedef struct GLNode{
    int tag;//标志域
    union{
        char atom;//原子结点的值域
        struct{
            struct GLNode * hp,*tp;
        }ptr;//子表结点的指针域，hp指向表头；tp指向表尾
    };
}*Glist;
Glist creatGlist(Glist C){
    //广义表C
    C=(Glist)malloc(sizeof(Glist));
    C->tag=1;
    //表头原子'a'
    C->ptr.hp=(Glist)malloc(sizeof(Glist));
    C->ptr.hp->tag=0;
    C->ptr.hp->atom='a';
```

```
    //表尾子表（b,c,d），是一个整体
    C->ptr.tp=(Glist)malloc(sizeof(Glist));
    C->ptr.tp->tag=1;
    C->ptr.tp->ptr.hp=(Glist)malloc(sizeof(Glist));
    C->ptr.tp->ptr.tp=NULL;
    //开始存放下一个数据元素（b,c,d），表头为‘b’，表尾为（c,d）
    C->ptr.tp->ptr.hp->tag=1;
    C->ptr.tp->ptr.hp->ptr.hp=(Glist)malloc(sizeof(Glist));
    C->ptr.tp->ptr.hp->ptr.hp->tag=0;
    C->ptr.tp->ptr.hp->ptr.hp->atom='b';
    C->ptr.tp->ptr.hp->ptr.tp=(Glist)malloc(sizeof(Glist));
    //存放子表(c,d)，表头为c，表尾为d
    C->ptr.tp->ptr.hp->ptr.tp->tag=1;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.hp=(Glist)malloc(sizeof(Glist));
    C->ptr.tp->ptr.hp->ptr.tp->ptr.hp->tag=0;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.hp->atom='c';
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp=(Glist)malloc(sizeof(Glist));
    //存放表尾d
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->tag=1;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.hp=(Glist)malloc(sizeof(Glist));
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.hp->tag=0;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.hp->atom='d';
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.tp=NULL;
    return C;
}
int GlistLength(Glist C){
    int Number=0;
    Glist P=C;
    while(P){
        Number++;
        P=P->ptr.tp;
    }
    return Number;
}
int main(){
    Glist C = creatGlist(C);
    printf("广义表的长度为：%d",GlistLength(C));
    return 0;
}
```

程序运行结果为：

```
广义表的长度为：2
```

广义表的深度

广义表的深度，可以通过观察该表中所包含括号的层数间接得到。

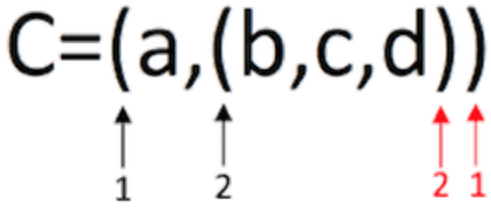


图 2 广义表示意图

从图 2 中可以看到，此广义表从左往右数有两层左括号（从右往左数也是如此），因此该广义表的深度为 2。

再比如，广义表 {{1,2},{3,{4,5}}} 中，子表 {1,2} 和 {3,{4,5}} 位于同层，此广义表中包含 3 层括号，因此深度为 3。

> 以上观察括号的方法需将广义表当做字符串看待，并借助栈存储结构实现，这里不做重点介绍。

编写程序计算广义表的深度时，以图 1a) 中的广义表为例，可以采用递归的方式：

- 依次遍历广义表 C 的每个节点，若当前节点为原子（tag 值为 0），则返回 0；若为空表，则返回 1；反之，则继续遍历该子表中的数据元素。
- 设置一个初始值为 0 的整形变量 max，每次递归过程返回时，令 max 与返回值进行比较，并取较大值。这样，当整个广义表递归结束时，max+1 就是广义表的深度。

> 其实，每次递归返回的值都是当前所在的子表的深度，原子默认深度为 0，空表默认深度为 1。

C 语言实现代码如下：

```
01.  #include <stdio.h>
02.  #include <stdlib.h>
03.
04.  typedef struct GLNode{
05.      int tag;//标志域
06.      union{
07.          char atom;//原子结点的值域
08.          struct{
09.              struct GLNode * hp,*tp;
10.          }ptr;//子表结点的指针域，hp指向表头；tp指向表尾
11.      };
12.  }*Glist,GNode;
13.
14.  Glist creatGlist(Glist C){
15.      //广义表C
16.      C=(Glist)malloc(sizeof(GNode));
17.      C->tag=1;
18.      //表头原子'a'
19.      C->ptr.hp=(Glist)malloc(sizeof(GNode));
```

```
    C->ptr.hp->tag=0;
    C->ptr.hp->atom='a';
    //表尾子表（b,c,d），是一个整体
    C->ptr.tp=(Glist)malloc(sizeof(GNode));
    C->ptr.tp->tag=1;
    C->ptr.tp->ptr.hp=(Glist)malloc(sizeof(GNode));
    C->ptr.tp->ptr.tp=NULL;
    //开始存放下一个数据元素（b,c,d），表头为‘b’，表尾为（c,d）
    C->ptr.tp->ptr.hp->tag=1;
    C->ptr.tp->ptr.hp->ptr.hp=(Glist)malloc(sizeof(GNode));
    C->ptr.tp->ptr.hp->ptr.hp->tag=0;
    C->ptr.tp->ptr.hp->ptr.hp->atom='b';
    C->ptr.tp->ptr.hp->ptr.tp=(Glist)malloc(sizeof(GNode));
    //存放子表(c,d)，表头为c，表尾为d
    C->ptr.tp->ptr.hp->ptr.tp->tag=1;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.hp=(Glist)malloc(sizeof(GNode));
    C->ptr.tp->ptr.hp->ptr.tp->ptr.hp->tag=0;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.hp->atom='c';
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp=(Glist)malloc(sizeof(GNode));
    //存放表尾d
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->tag=1;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.hp=(Glist)malloc(sizeof(GNode));
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.hp->tag=0;
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.hp->atom='d';
    C->ptr.tp->ptr.hp->ptr.tp->ptr.tp->ptr.tp=NULL;
    return C;
}
int GlistDepth(Glist C){
    //如果表C为空表时，直接返回长度1;
    if (!C) {
        return 1;
    }
    //如果表C为原子时，直接返回0;
    if (C->tag==0) {
        return 0;
    }
    int max=0;//设置表C的初始长度为0;
    for (Glist pp=C; pp; pp=pp->ptr.tp) {
        int dep=GlistDepth(pp->ptr.hp);
        if (dep>max) {
            max=dep;//每次找到表中遍历到深度最大的表，并用max记录
        }
    }
    //程序运行至此处，表明广义表不是空表，由于原子返回的是0，而实际长度是1，所以，此处要+1;
    return max+1;
}
int main(int argc, const char * argv[]) {
```

```
67.       Glist C=creatGlist(C);
68.       printf("广义表的深度为：%d",GlistDepth(C));
69.       return 0;
70.   }
```

程序运行结果：

```
广义表的深度为：2
```