系统设计

GWL

（北京理工大学数学与统计学院，北京 100081）

目录

第 1 章 系统概述

该项目全部采用 C 语言编码，模拟了部分操作系统的功能，实现的功能如下：

1. 使用指针申请 100MB 空间模拟操作系统
2. 对空间按照字节分块，能查看指定块的使用情况
3. 查看总体空间使用情况以及空闲块链表
4. 使用自定义指令实现内存分配与回收
5. 使用自定义指令实现链表的建立、使用与回收
6. 使用自定义指令实现数组的建立、使用与回收
7. 使用自定义指令实现堆的建立、使用与回收
8. 使用自定义指令实现栈的建立、使用与回收
9. 使用自定义指令实现队列的建立、使用与回收
10. 使用自定义指令实现树的建立、使用与回收
11. 使用自定义指令实现图的建立、使用与回收

第 2 章 系统架构

项目使用命令行进行操作，用户输入命令后，由自定义解析函数进行解析，然后执行相应操作，直到用户输入退出命令。

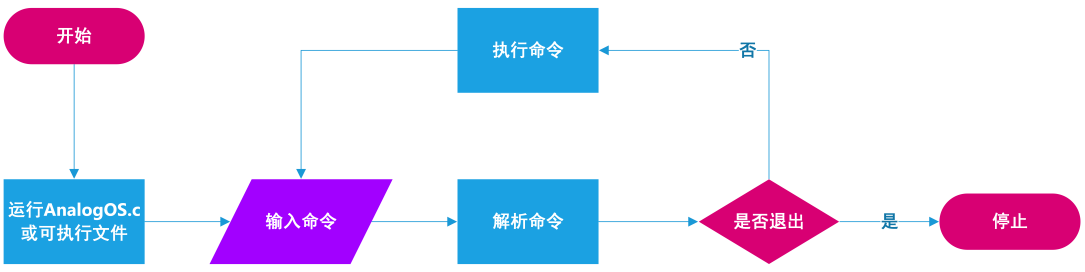

图 1：系统架构

第 3 章 功能设计

3.1 内存管理

3.1.1 内存信息

使用指针申请 100MB 空间模拟操作系统。按照字节进行分块，用字符串存储空间使用信息，已经使用的内存块相应位置的字符为‘1’，未使用的则为‘0’。同时用两个长整型变量存储全局已使用空间和未使用空间，在每次申请内存和释放内存时进行相应的更改，需要时访问即可。

```
// 查看总体空间使用情况
checkSpace()
// 查看指定地址的使用情况
checkBlock(void *address)
// 查看空闲块链表
checkFreeSpaceList()
```

address 为想要查看使用情况的地址。

在本项目中，函数使用不涉及返回值，故所有函数声明均省略函数返回类型。

3.1.2 内存分配

为了尽量减少空间的浪费，减少申请释放内存的消耗时间，项目采用基于**伙伴算法**的存储分配机制。即把内存块分到数个链表中，分别链接具有 1、2、4、8、16、32、64、128……个连续空闲内存块的空间。在内存分配和回收过程中始终保持空闲的连续内存块大小为 2 的幂次方。

```
// 申请 size 个字节的连续空间
myMalloc(size_t size)
// 申请 n × size 个字节的连续空间
myCalloc(size_t n, size_t size)
```

这两个函数的使用类似于库函数 malloc 和 calloc，后者会对申请的空间赋零，而前者不会。

3.1.3 内存回收

在内存分配的时候，用哈希表记录指针指向的空间大小，在释放空间的时候按照哈希表记录的大小进行释放。

```
// 释放指针指向的空间
myFree(void *address)
```

3.2 链表

在设计链表命令之前，先对数据元素以及链表节点进行预定义。

```
typedef long long ELEMENT_TYPE;
// 链表节点
typedef struct listNode
{
    ELEMENT_TYPE val;
    struct listNode *next;
} LIST_NODE;
```

3.2.1 创建链表

```
initList()
```

创建链表也就是创建也就是创建一个链表头节点，不储存元素。创建成功后返回头节点地址，作为链表的地址。

3.2.2 删除链表

```
delAllList(LIST_NODE *head)
```

将包括头节点在内的所有链表节点都释放。

3.2.3 清空链表

```
clearList(LIST_NODE *head)
```

除头节点外，将链表其余节点全部释放。

3.2.4 链表判空

```
listEmpty(LIST_NODE *head)
```

直接判断链表头节点的后继是否为空即可。

3.2.5 链表长度

```
listLength(LIST_NODE *head)
```

遍历一遍链表，每遇到一个节点，计数器加一，就得到了链表长度。

3.2.6 获取链表节点的值

```
getListNodeVal(LIST_NODE *node)
```

直接通过地址访问链表元素即可。

3.2.7 获取链表节点的位序

```
findListNode(LIST_NODE *head, ELEMENT_TYPE val)
```

链表头记为第 0 位，输出节点值等于 val 的第一个元素的位次。

3.2.8 获取链表节点的前驱

```
listPreNode(LIST_NODE *head, ELEMENT_TYPE val)
```

链表头记为第 0 位，输出节点值等于 val 的第一个元素的前驱。

3.2.9 获取链表节点的后继

```
listNextNode(LIST_NODE *head, ELEMENT_TYPE val)
```

链表头记为第 0 位，输出节点值等于 val 的第一个元素的后继。

3.2.10 遍历链表

```
visitList(LIST_NODE *head)
```

遍历链表，输出节点的元素值。

3.2.11 修改链表节点

```
changeListVal(LIST_NODE *head, size_t pos, ELEMENT_TYPE val)
```

头节点位序为 0，将位序为 pos 的链表元素值修改为 val。

3.2.12 插入链表节点

```
addListNode(LIST_NODE *head, size_t pos, ELEMENT_TYPE val)
```

头节点位序为 0，在位序为 pos 的链表节点后插入新节点，值为 val。

3.2.13 删除链表节点

```
delListNode(LIST_NODE *head, size_t pos)
```

头节点位序为 0，将位序为 pos 的链表节点删除。

3.3 数组

3.3.1 创建数组

```
initArray(size_t n)
```

申请一个具有 n 个元素的数组空间。

3.3.2 删除数组

```
delAllArray(ELEMENT_TYPE *arr)
```

将 arr 指向的空间释放掉。

3.3.3 获取数组特定位置的元素值

```
getArrayVal(ELEMENT_TYPE *arr, size_t pos)
```

序号从 0 开始，输出数组第 pos 位的元素值。

3.3.4 修改数组元素的值

```
changeArrayVal(ELEMENT_TYPE *arr, size_t pos, ELEMENT_TYPE
↪ val)
```

序号从 0 开始，将第 pos 位的元素修改为 val。

3.3.5 遍历数组

```
visitArray(ELEMENT_TYPE *arr)
```

按顺序输出数组元素的值。

3.4 堆

用数组来实现堆。在创建堆之前，先对堆的结构进行定义。nums 为用来实现堆的数组，size 为堆中的元素数量，maxSize 为堆的最大容量。

```
typedef struct
{
    ELEMENT_TYPE *nums;
    long long size, maxSize;
} HEAP;
```

3.4.1 创建堆

```
initHeap(size_t maxSize)
```

创建容量为 maxSize 的大根堆。堆有容量限制是因为这里是用数组来实现的大根堆，所以要提前确定堆的容量，以此来申请数组空间。

3.4.2 删除堆

```
delAllHeap(HEAP *heap)
```

释放堆里的数组和堆结构本身。

3.4.3 清空堆

```
clearHeap(HEAP *heap)
```

直接将堆的当前大小置零即可。

3.4.4 入堆

```
heapPush(HEAP *heap, ELEMENT_TYPE val)
```

先将 val 放到堆内最后一个元素的后面，再调整堆，使其满足大根堆特性即可。

3.4.5 出堆

```
heapPop(HEAP *heap)
```

输出堆内第一个元素，再用最后一个元素赋值给第一个元素，堆的大小减一。

3.4.6 获取堆顶元素

```
getHeapTop(HEAP *heap)
```

直接输出堆内第一个元素。

3.4.7 获取堆的元素数量

```
getHeapSize(HEAP *heap)
```

直接输出堆的大小。

3.4.8 输出堆

```
outputHeap(HEAP *heap)
```

依次输出堆中数组的元素，注意输出数量为堆的大小，而不是堆的最大容量。

3.5 栈

用链表来实现栈。在创建栈之前，先对栈的结构进行定义。head 为栈链表的头结点，size 为栈中的元素数量。

```
// 栈
typedef struct
{
    LIST_NODE *head;
    long long size;
} STACK;
```

3.5.1 创建栈

```
1 initStack()
```

申请一个栈空间以及链表头节点。

3.5.2 删除栈

```
1 delAllStack(STACK *stack)
```

释放栈中链表的全部节点以及栈本身。

3.5.3 清空栈

```
1 clearStack(STACK *stack)
```

释放栈中链表除了透节点以外的所有结点。

3.5.4 获取栈顶元素

```
1 getStackTopVal(STACK *stack)
```

输出链表头节点的下一个元素。

3.5.5 获取栈内元素数量

```
1 getStackSize(STACK *stack)
```

输出栈中元素数量。

3.5.6 入栈

```
1 stackPush(STACK *stack, ELEMENT_TYPE val)
```

创建一个新的链表节点，加入到栈中链表头节点的下一个。

3.5.7 出栈

```
1 stackPop(STACK *stack)
```

输出栈中链表头节点的下一个元素，并将其释放。

3.6 队列

用链表来实现队列。在创建队列之前，先对队列的结构进行定义。head 为队列链表的头结点，tail 为队列链表的尾节点，size 为队列中的元素数量。

```
typedef struct
{
    LIST_NODE *head, *tail;
    long long size;
} QUEUE;
```

3.6.1 创建队列

```
initQueue()
```

申请一个队列空间和队列内的链表头节点，将头节点赋值给尾节点。

3.6.2 删除队列

```
delAllQueue(QUEUE *queue)
```

清空并释放队列链表及队列结构本身。

3.6.3 清空队列

```
clearQueue(QUEUE *queue)
```

清空并释放队列链表除头节点以外的节点。

3.6.4 获取队首元素

```
getQueueHeadVal(QUEUE *queue)
```

输出队列链表头节点后的第一个节点。

3.6.5 获取队内元素数量

```
getQueueSize(QUEUE *queue)
```

输出队列元素数量。

3.6.6 入队

```
queuePush(QUEUE *queue, ELEMENT_TYPE val)
```

创建一个值为 val 的链表节点，加入到队列链表尾节点的下一个，同时更新尾结点。

3.6.7 出队

```
queuePop(QUEUE *queue)
```

输出并释放队列链表头节点后的第一个元素。

3.7 树

本项目中的树为带有祖先信息的二叉树。在创建树之前，先对树节点和树的结构进行定义。val 为树节点的值，left、right 和 parent 分别代表节点的左右子树和祖先地址。root 代表树的根节点地址，size 代表树中节点个数。

```
// 树节点
typedef struct treeNode
{
    ELEMENT_TYPE val;
    struct treeNode *left, *right, *parent;
} TREE_NODE;
// 树
typedef struct
{
    TREE_NODE *root;
    long long size;
} TREE;
```

3.7.1 创建树

```
initTree()
```

创建一颗空树。

3.7.2 删除树

```
delAllTree(TREE *tree)
```

释放所有的树节点和树结构本身。

3.7.3 清空树

```
clearTree(TREE *tree)
```

释放所有的树节点。

3.7.4 获取树的节点个数

```
countTreeNode(TREE *tree)
```

输出树的节点个数。

3.7.5 获取树的深度

```
countTreeDeep(TREE *tree)
```

输出树的深度，空树深度为 0，只有根节点的树深度为 1。

3.7.6 获取树的根地址

```
treeRoot(TREE *tree)
```

输出树的根节点。

3.7.7 获取树节点的值

```
getTreeNodeVal(TREE *tree, TREE_NODE *node)
```

输出树中 node 节点的值。

3.7.8 修改树节点的值

```
changeTreeNodeVal(TREE *tree, TREE_NODE *node, ELEMENT_TYPE
↪ val)
```

将树中 node 节点的值修改为 val。

3.7.9 给根节点赋值

```
addTreeRootVal(TREE *tree, ELEMENT_TYPE val)
```

若根节点不存在时，地址为空，无法赋值，因此根节点要专门用一个函数来赋值。

3.7.10 获取树节点的祖先

```
getTreeNodeParent(TREE *tree, TREE_NODE *node)
```

获取树中节点的祖先。

3.7.11 获取树节点的孩子

```
getTreeNodeChildren(TREE *tree, TREE_NODE *node)
```

获取树中节点的孩子。

3.7.12 获取树节点的兄弟

```
getTreeNodeBrother(TREE *tree, TREE_NODE *node)
```

获取树中节点的孩子。

3.7.13 向节点的左子树添加值

```
addTreeNodeInLeft(TREE *tree, TREE_NODE *node, ELEMENT_TYPE
↪ val)
```

向树中节点的左子树添加新节点。

3.7.14 向节点是右子树添加值

```
addTreeNodeInRight(TREE *tree, TREE_NODE *node, ELEMENT_TYPE
↪ val)
```

向树中节点的右子树添加新节点。

3.7.15 删除节点及其子树

```
delTreeNodeAndChildren(TREE *tree, TREE_NODE *root)
```

释放该节点及其子树。

3.7.16 前序遍历树

```
preOrderVisitTree(TREE *tree)
```

输出树的前序遍历。

3.7.17 中序遍历树

```
inOrderVisitTree(TREE *tree)
```

输出树的中序遍历。

3.7.18 后序遍历树

```
postOrderVisitTree(TREE *tree)
```

输出树的后序遍历。

3.7.19 层序遍历树

```
levelOrderVisitTree(TREE *tree)
```

输出树的层序遍历。

3.7.20 可视化树的结构

```
visualTree(TREE *tree)
```

输出将树逆时针旋转 90 度的树结构

3.8 图

在创建图之前，先对图的结构进行定义。图的顶点采用普通链表存储，边采用邻接链表存储。EDGE_NODE 结构中，listHead 代表由顶点 val 指出的顶点链表。MAP 结构中，nodeHead，edgeHead 分别代表顶点链表

的头节点和边链表的头节点。edgeSize 和 nodeSize 分别代表边的数量和顶点的数量。

```
// 边
typedef struct edgeNode
{
    ELEMENT_TYPE val;
    LIST_NODE *listHead;
    struct edgeNode *next;
} EDGE_NODE;
// 图
typedef struct
{
    // 存储顶点
    LIST_NODE *nodeHead;
    // 存储边
    EDGE_NODE *edgeHead;
    long long edgeSize, nodeSize;
} MAP;
```

3.8.1 创建图

```
initMap()
```

创建一个空的图。

3.8.2 删除图

```
delAllMap(MAP *map)
```

释放图的所有顶点、边，以及图结构本身。

3.8.3 清空图

```
clearMap(MAP *map)
```

释放图所有的顶点和边。

3.8.4 添加顶点

```
addMapNode(MAP *map, ELEMENT_TYPE val)
```

向图中添加值为 val 的顶点。

3.8.5 删除顶点

```
delMapNode(MAP *map, ELEMENT_TYPE val)
```

删除图中值为 val 的顶点以及与该顶点相连的边。

3.8.6 添加边

```
addMapEdge(MAP *map, ELEMENT_TYPE start, ELEMENT_TYPE end)
```

向图中添加起点为 start，终点为 end 的边。

3.8.7 删除边

```
delMapEdge(MAP *map, ELEMENT_TYPE start, ELEMENT_TYPE end)
```

删除图中起点为 start，终点为 end 的边。

3.8.8 获取进入某顶点的点

```
inMapNode(MAP *map, ELEMENT_TYPE val)
```

输出进入顶点 val 的所有顶点。

3.8.9 获取离开某顶点的点

```
outMapNode(MAP *map, ELEMENT_TYPE val)
```

输出离开顶点 val 的所有顶点。

3.8.10 获取图的顶点数

```
mapNodeNum(MAP *map)
```

输出图的顶点数。

3.8.11 获取图的边数

```
mapEdgeNum(MAP *map)
```

输出图的边数。

3.8.12 遍历所有顶点

```
visitAllMapNode(MAP *map)
```

输出图中的所有顶点。

3.8.13 遍历所有边

```
visitAllMapEdge(MAP *map)
```

输出图中所有的边。

3.9 解析函数

```
eval(char *command)
```

该函数与 Python 等语言的 eval 函数功能类似，解析输入的命令字符串，解析出字符串中的参数，最后执行相应的函数。

第 4 章 未来优化

如果未来还有更新，可以在以下几个方面进行优化：

- 现有内存管理仅为基础伙伴算法，效率还有提升的空间
- 增加可视化窗口选项，可以让用户选择命令行模式或可视化模式
- 项目现有的容错能力较弱，可以在后期加强对异常输入的处理
- 目前的数据类型较为单一，后续可以增加对多种数据类型的支持
- 可以在现有函数基础上，对数据进行更加精细化的操作
- 可以增加对多种复合数据结构的支持
- ……