燕山大学

数据结构实验报告

Data Structure *Experiment Report*

学生所在学院：信息学院

学生所在班级：2020 级计算机 2 班

姓　　名：董福江

学　　号：201911040033

指导教师：陈子军、何洪豆

教 务 处

2021 年 11 月

实验一 有序表的建立、插入与删除

一、实验目的

1. 了解有序表的顺序存储结构。
2. 掌握有序表元素在内存中是怎样存储的。
3. 在有序表中实现如下操作
4. 插入一个新元素到第 i 个位置，使原来标号为增 1；
5. 删除第 i 个位置的元素；
6. 存一个新元素到第 i 个位置；
7. 读表；
8. 检索表中第 i 个元素；
9. 寻表的长度。

二、实验原理

线性表是最常用的而且也是最简单的一种数据结构，线性表是 N 个数据元素的有限序列。例如 26 个英文元素的字母表：（A，B，C，D…）。其数据结构的描述为：Linear_list=(D,R)其中：D={a_i|a_i属于 D_0,i=1,2,3,···}R={N},N={<a_{i-1},a_i>|

i=2,3,4,···}。本实验是以数组的形式把有序表存放在计算机内存的一个连续的区域内，这样便有：LOC(a_{i+1})=LOC(a_i)+m。其中 m 是存放每个元素所占的内存字数。LOC(a_i)=LO+m·(i-1)。其中 LO 是 a_i 的地址即首地址。

实验程序说明：线性表是最常用的而且也是最简单的一种数据结构，线性表是 N 个数据元素的有限序列。例如 26 个英文元素的字母表：（A，B，C，D…）。其数据结构的描述为：Linear_list=(D,R)其中：D={a_i|a_i属于 D_0,i=1,2,3,···}R={N},N={<a_{i-1},a_i>|i=2,3,4,···}。本实验是以数组的形式把有序表存放在计算机内存的一个连续的区域内，这样便有：LOC(a_{i+1})=LOC(a_i)+m。其中 m 是存放每个元素所占的内存字数。LOC(a_i)=LO+m·(i-1)。其中 LO 是 a_i 的地址即首地址。

实验程序说明：

插入一个新元素到第 i 个位置，既把元素 a_i 向后移一个位置，成为元素 a_{i+1}，把新元素放入到第 i 个位置，其他元素依次后移。

存一新元素到第 i 个位置是把元素 a_i 冲掉后存上新值。

删除第 i 个元素就是把余后的元素依次向前移一个位置。即：以元素 a_{i+1}，a_{i+2}，……，依次取代 a_i，a_{i+1}……。删除后的表长是 n-1（n 是原表长）。

三、实验内容

1. 代码

- **list.h**

```
#ifndef __LIST_LIST_H__
#define __LIST_LIST_H__
#include "../def.h"
#include <stdbool.h>
#include <stdlib.h>

typedef void *linked_list_type;
typedef void *linked_node_type;
/**
* @brief linked list[without header node]
*/
struct linked_list {
struct linked_node *head;
size_t size;
};

/**
* @brief linked list node
*/
struct linked_node {
elem_type elem;
struct linked_node *prev;
struct linked_node *next;
};

/***************linkedList function*****************/
linked_list_type init_linked_list(linked_list_type *p_list);

size_t size_linked_list(linked_list_type list);
bool empty_linked_list(linked_list_type list);
```

```
/**
* @brief add elem to linked list tail
*/
linked_list_type add_linked_list(linked_list_type list, elem_type elem);

/**
* @brief add elem to index back
*/
linked_list_type   add_by_index_linked_list(linked_list_type   list,   size_t
index,
elem_type elem);
linked_list_type foreach_linked_list(linked_list_type list, callback fun);
linked_list_type get_linked_list(linked_list_type list, size_t index,
callback fun);
linked_list_type remove_linked_list(linked_list_type list, size_t index,
callback fun);
linked_list_type clear_linked_list(linked_list_type list);
linked_list_type destory_linked_list(linked_list_type *p_list);
#endif
```

- **List.c**

```
#include "list.h"

static struct linked_node *get_node(linked_list_type list, size_t index);
static void add_node(linked_list_type list, struct linked_node *node,
elem_type elem);

linked_list_type init_linked_list(linked_list_type *p_list) {
if (NULL == p_list || NULL != *p_list)
return p_list;

*p_list = calloc(1, sizeof(struct linked_list));
struct linked_list *list = (struct linked_list *)*p_list;
if (NULL != list) {
list->size = 0;
```

```
list->head = NULL;
}
return p_list;
}

size_t size_linked_list(linked_list_type list) {
if (list == NULL) {
return 0;
}
return ((struct linked_list *)list)->size;
}
bool empty_linked_list(linked_list_type list) {
return size_linked_list(list) <= 0;
}

linked_list_type add_linked_list(linked_list_type list, elem_type elem) {
if (NULL == list) {
return list;
}

add_node(list, get_node(list, 0), elem);
return list;
}

linked_list_type add_by_index_linked_list(linked_list_type list, size_t
index,
elem_type elem) {
if (NULL == list)
return list;

add_node(list, get_node(list, index), elem);
return list;
}

linked_list_type foreach_linked_list(linked_list_type list, callback fun) {

if (NULL == list)
```

```
return list;

struct linked_node *p_cur = ((struct linked_list *)list)->head;
while (p_cur != NULL) {
fun(p_cur->elem);
p_cur = p_cur->next;
}

return list;
}

linked_list_type get_linked_list(linked_list_type list, size_t index,
callback fun) {
if (NULL == list)
return list;
struct linked_node *node = get_node(list, index);
if (NULL != node) {
fun(node->elem);
}
return list;
}
linked_list_type remove_linked_list(linked_list_type list, size_t index,
callback fun) {
if (NULL == list)
return list;

struct linked_node *p_node = get_node(list, index);
if (NULL != p_node) {
struct linked_list *mylist = (struct linked_list *)list;

if (mylist->head == p_node) {
mylist->head = p_node->next;
}

if (NULL != p_node->prev)
p_node->prev->next = p_node->next;
if (NULL != p_node->next)
```

```
p_node->next->prev = p_node->prev;

free(p_node);
--(mylist->size);
}
return list;
}function:
linked_list_type clear_linked_list(linked_list_type list) {
if (NULL == list)
return list;

struct linked_list *mylist = (struct linked_list *)list;
struct linked_node *p_node = mylist->head;

while (NULL != p_node) {
struct linked_node *p_temp = p_node->next;
free(p_node);
p_node = p_temp;
}

mylist->size = 0;
return list;
}
linked_list_type destory_linked_list(linked_list_type *p_list) {
if (NULL == p_list)
return p_list;

clear_linked_list(*p_list);
free(*p_list);
*p_list = NULL;
return p_list;
}

static void add_node(linked_list_type list, struct linked_node *node,
elem_type elem) {
// create a new node
struct linked_node *new_node =
```

```
(struct linked_node *)calloc(1, sizeof(struct linked_node));
if (NULL == new_node)
return;
new_node->elem = elem;

struct linked_list *mylist = (struct linked_list *)list;
if (node == mylist->head || NULL == node) {
if (NULL == node)
node = mylist->head;
mylist->head = new_node;
}
if (NULL != node) {
new_node->next = node;
if (NULL != node->prev) {
new_node->prev = node->prev;
node->prev->next = new_node;
}
node->prev = new_node;
}
++(mylist->size);
}

static struct linked_node *get_node(linked_list_type list, size_t index) {
if (size_linked_list(list) <= index)
return NULL;

struct linked_node *p_cur = ((struct linked_list *)list)->head;
for (; index > 0; --index)
p_cur = p_cur->next;
return p_cur;
}
```

function: test_list

```
void test_list(void) {
size_t size = 10;
struct people *peoples[10] = {0};
```

```
for (int i = 0; i < size; ++i) {
peoples[i] = (struct people *)malloc(sizeof(struct people));
peoples[i]->age = rand() % (size * 10);
}

linked_list_type list = NULL;
init_linked_list(&list);

/** test functions when linked list is empty */
remove_linked_list(list, 0, print);
get_linked_list(list, 0, print);
size_linked_list(list);
empty_linked_list(list);
foreach_linked_list(list, print);
clear_linked_list(list);

/** test at normal case */
printf("%-45s", "add node at posittion 143: ");
add_by_index_linked_list(list, 143, peoples[0]);
foreach_linked_list(list, print);
putchar('\n');

printf("%-45s", "add node at header: ");
add_linked_list(list, peoples[1]);
foreach_linked_list(list, print);
putchar('\n');

printf("%-45s", "update the node's element at position 0: ");
get_linked_list(list, 0, update);
foreach_linked_list(list, print);
putchar('\n');

printf("%-45s", "get the node's element at position 0: ");
get_linked_list(list, 0, print);
putchar('\n');

printf("%-45s", "remove the node at posistion 0: ");
```

```
remove_linked_list(list, 0, NULL);
foreach_linked_list(list, print);
putchar('\n');

printf("%-45s", "clear all nodes and destory the linked list: ");
destory_linked_list(&list);
foreach_linked_list(list, print);
putchar('\n');

for (int i = 0; i < size; ++i) {
free(peoples[i]);
}
}
```

1. 结果

```
testing the linked list......
add node at posittion 143:                    {age: 67}
add node at header:                           {age: 68}  {age: 67}
update the node's element at position 0:      {age: 44444}  {age: 67}
get the node's element at position 0:         {age: 44444}
remove the node at posistion 0:               {age: 67}
clear all nodes and destory the linked list:
```

实验二 中序线索二叉树

一、实验目的

1．理解线索的含义，掌握线索二叉树的算法。

2．了解中序线索及其遍历的实现过程。

二、实验原理

动态线索化二叉树，遍历算法，即在遍历的过程中线索化二叉树，遍历完成后恢复二叉树，可以保证在时间复杂度为 O(N)的同时，空间复杂度为 O(1),而递归遍历，借用栈遍历的空间复杂度为 O(N),可以极大的节省空间。

Morris 算法在遍历的时候避免使用了栈结构，而是让下层到上层有指针，具体是通过底层节点指向 NULL 的空闲指针返回上层的某个节点，从而完成下层到上层的移动。我们知道二叉树有很多空闲的指针，比如某个人节点没有右孩子，我们称这种情况为空闲状态，Morris 算法的遍历就是利用了这些 空闲的指针

原理：模拟中序递归遍历的递归机（在中序递归遍历过程中，每个节点都会被访问两次，该序列即为递归机），前序中序遍历分别为第一次、第二次访问节点的序列，而后序遍历为二叉树所有右边界从左向右的反向遍历。

三、实验内容

tree.h

```
#ifndef __TREE_H__
#define __TREE_H__
```

```
#include "../def.h"
#include <stdlib.h>

struct tree_node {
elem_type elem;
struct tree_node *parent;
struct tree_node *left;
struct tree_node *right;
};

void morris(struct tree_node *node, callback fun);
void morris_in(struct tree_node *node, callback fun);
void morris_post(struct tree_node *node, callback fun);
void morris_pre(struct tree_node *node, callback fun);
#endif
```

tree.c

```
#include "./tree.h"

/**
* @brief morris algorithm template
*
* @param node
*/
void morris(struct tree_node *node, callback fun) {
if (NULL == node) {
return;
}

struct tree_node *most_right = NULL;
while (NULL != node) {
most_right = node->left;
if (NULL != most_right) {
while (NULL != most_right->right && most_right->right != node) {
most_right = most_right->right;
}
```

```
if (NULL == most_right->right) {
// print
fun(node->elem);

// first
most_right->right = node;
node = node->left;
continue;
} else {
// seconed
most_right->right = NULL;
}
} else {
// print
fun(node->elem);
}
// print
fun(node->elem);
node = node->right;
}
}

void morris_pre(struct tree_node *node, callback fun) {

if (NULL == node) {
return;
}

struct tree_node *most_right = NULL;
while (NULL != node) {
most_right = node->left;
if (NULL != most_right) {
while (NULL != most_right->right && most_right->right != node) {
most_right = most_right->right;
}
```

```
if (NULL == most_right->right) {
fun(node->elem);
most_right->right = node;
node = node->left;
continue;
} else {
most_right->right = NULL;
}
} else {
fun(node->elem);
}
node = node->right;
}
}

void morris_in(struct tree_node *node, callback fun) {
if (NULL == node) {
return;
}

struct tree_node *most_right = NULL;
while (NULL != node) {
most_right = node->left;
if (NULL != most_right) {
while (NULL != most_right->right && most_right->right != node) {
most_right = most_right->right;
}

if (NULL == most_right->right) {
most_right->right = node;
node = node->left;
continue;
} else {
most_right->right = NULL;
}
}
```

```
fun(node->elem);
node = node->right;
}
}

/**
* @brief reverse the right edge
*
* @param from
* @return
*/
struct tree_node *reverse_edge(struct tree_node *from) {
struct tree_node *pre = NULL;
struct tree_node *next = NULL;

while (NULL != from) {
next = from->right;
from->right = pre;
pre = from;
from = next;
}
return pre;
}

void foreach_edge(struct tree_node *head, callback fun) {
struct tree_node *tail = reverse_edge(head);
struct tree_node *node = tail;
while (NULL != node) {
fun(node->elem);
node = node->right;
}
reverse_edge(tail);
}

/**
* @brief
*
```

```
* @param node
* @param fun
*/
void morris_post(struct tree_node *node, callback fun) {
if (NULL == node) {
return;
}

struct tree_node *head = node;
struct tree_node *most_right = NULL;
while (NULL != node) {
most_right = node->left;
if (NULL != most_right) {
while (NULL != most_right->right && most_right->right != node) {
most_right = most_right->right;
}

if (NULL == most_right->right) {
most_right->right = node;
node = node->left;
continue;
} else {
most_right->right = NULL;
foreach_edge(node->left, fun);
}
}
node = node->right;
}

foreach_edge(head, fun);
}
```

function: test_morris

```
void test_mirros(void) {
/** create a simple binary tree */
size_t size = 5;
```

```
struct tree_node *nodes[5] = {0};
for (long i = 0; i < size; ++i) {
nodes[i] = (struct tree_node *)malloc(sizeof(struct tree_node *) * size);
nodes[i]->elem = (struct people *)malloc(sizeof(struct people));
((struct people *)nodes[i]->elem)->age = i;
nodes[i]->left = NULL;
nodes[i]->right = NULL;
nodes[i]->parent = NULL;
}

nodes[0]->left = nodes[1];
nodes[0]->right = nodes[2];
nodes[1]->left = nodes[3];
nodes[1]->right = nodes[4];
nodes[1]->parent = nodes[0];
nodes[2]->parent = nodes[0];
nodes[3]->parent = nodes[1];
nodes[4]->parent = nodes[1];

printf("%-40s", "get the binary tree's recursive state: ");
morris(nodes[0], print);
putchar('\n');

printf("%-40s", "traversal through preorder: ");
morris_pre(nodes[0], print);
putchar('\n');
printf("%-40s", "traversal through inorder: ");
morris_in(nodes[0], print);
putchar('\n');
printf("%-40s", "traversal through postorder: ");
morris_post(nodes[0], print);
putchar('\n');

for (int i = 0; i < size; ++i) {
free(nodes[i]);
}
}
```

结果

```
testing the binary tree cueing......
get the binary tree's recursive state:   {age: 0}  {age: 1}  {age: 3}  {age: 3}  {age: 1}  {age: 4}  {age: 4}  {age: 0}  {age: 2}  {age: 2}
traversal through preorder:              {age: 0}  {age: 1}  {age: 3}  {age: 4}  {age: 2}
traversal through inorder:               {age: 3}  {age: 1}  {age: 4}  {age: 0}  {age: 2}
traversal through postorder:             {age: 3}  {age: 4}  {age: 1}  {age: 2}  {age: 0}
```

实验三 快速排序

一、实验目的

掌握快速排序的算法。

二、实验原理

快速排序的基本思想是：通过一趟排序将要排序的数组分割成独立的两部分，其中一部分的所有数据比另一部分所有数据要小，再按这种方法对这两部分数据分别进行快速排序，整个序列过程可以递归进行，使整个数组变成有序序列。这是典型的分治思想，即分治法。

三、实验内容

代码

sort.h

```
#ifndef __SORT_SORT_H__
#define __SORT_SORT_H__

#include "../def.h"
```

```
#include <stdlib.h>
#include <time.h>
#include <memory.h>

void quicksort(elem_type elems[], size_t length, comparator cmp,
callback fun);

#endif
```

sort.c

```
#include "./sort.h"

static void swap(elem_type elems[], size_t first, size_t last) {
elem_type temp = elems[first];
elems[first] = elems[last];
elems[last] = temp;
}

static size_t partition(elem_type elems[], size_t begin, size_t end,
comparator cmp) {
// random position
swap(elems, begin, begin + (size_t)(rand() % (end - begin)));
elem_type povit = elems[begin];

--end;
while (begin < end) {
while (begin < end) {
if (cmp(povit, elems[end]) < 0) {
--end;
} else {
elems[begin++] = elems[end];
break;
}
}

while (begin < end) {
```

```
if (cmp(elems[begin], povit) < 0) {
++begin;
} else {
elems[end--] = elems[begin];
break;
}
}
}
elems[begin] = povit;
return begin;
}

static void qsortRecursion(elem_type elems[], size_t begin, size_t end,
comparator cmp) {
if (end - begin > 1) {
size_t pivot_index = partition(elems, begin, end, cmp);
qsortRecursion(elems, begin, pivot_index, cmp);
qsortRecursion(elems, pivot_index + 1, end, cmp);
}
}

void quicksort(elem_type elems[], size_t length, comparator cmp,
callback fun) {
size_t size = sizeof(elem_type) * length;
elem_type *sorted_elems = (elem_type *)malloc(size);
memcpy(sorted_elems, elems, size);

qsortRecursion(sorted_elems, 0, length, cmp);

for (int i = 0; i < length; ++i) { // fun(elems[]);
fun(sorted_elems[i]);
}
}
```

function: test_quicksort

```
void test_quicksort(void) {
```

```
size_t size = 10;
struct people *peoples[10] = {0};
for (int i = 0; i < size; ++i) {
peoples[i] = (struct people *)malloc(sizeof(struct people));
peoples[i]->age = rand() % (size * 10);
}

// use quick sort
printf("%-20s", "Before sorting: ");
for (int i = 0; i < size; ++i) {
print(peoples[i]);
}
putchar('\n');

printf("%-20s", "quick sort: ");
quicksort((elem_type *)peoples, size, cmp, print);
putchar('\n');

printf("%-20s", "After sorting: ");
for (int i = 0; i < size; ++i) {
print(peoples[i]);
}
putchar('\n');

// free array's elements' memory
for (int i = 0; i < size; ++i) {
free(peoples[i]);
}
}
```

运行结果

```
testing the quick sort......
Before sorting:      {age: 57}  {age: 55}  {age: 53}  {age: 78}  {age: 61}  {age: 21}  {age: 4}  {age: 80}  {age: 6}  {age: 35}
quick sort:          {age: 4}  {age: 6}  {age: 21}  {age: 35}  {age: 53}  {age: 55}  {age: 57}  {age: 61}  {age: 78}  {age: 80}
After sorting:       {age: 57}  {age: 55}  {age: 53}  {age: 78}  {age: 61}  {age: 21}  {age: 4}  {age: 80}  {age: 6}  {age: 35}
```

附录：其他代码

Main.c

```
#include "./list/list.h"
#include "./sort/sort.h"
#include "./tree/tree.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int cmp(elem_type a, elem_type b) {
return ((struct people *)a)->age - ((struct people *)b)->age;
}

void print(elem_type elem) {
printf(" {age: %d} ", ((struct people *)elem)->age);
}

void update(elem_type elem) { ((struct people *)elem)->age = 44444;
}

void test_list(void) {
size_t size = 10;
struct people *peoples[10] = {0};
for (int i = 0; i < size; ++i) {
peoples[i] = (struct people *)malloc(sizeof(struct people));
peoples[i]->age = rand() % (size * 10);
}

linked_list_type list = NULL;
init_linked_list(&list);

/** test functions when linked list is empty */
remove_linked_list(list, 0, print);
get_linked_list(list, 0, print);
```

```
size_linked_list(list);
empty_linked_list(list);
foreach_linked_list(list, print);
clear_linked_list(list);

/** test at normal case */
printf("%-45s", "add node at posittion 143: ");
add_by_index_linked_list(list, 143, peoples[0]);
foreach_linked_list(list, print);
putchar('\n');

printf("%-45s", "add node at header: ");
add_linked_list(list, peoples[1]);
foreach_linked_list(list, print);
putchar('\n');

printf("%-45s", "update the node's element at position 0: ");
get_linked_list(list, 0, update);
foreach_linked_list(list, print);
putchar('\n');

printf("%-45s", "get the node's element at position 0: ");
get_linke#ifndef __DEF_H__
#define __DEF_H__

typedef void *elem_type;
typedef void (*callback)(elem_type);
typedef int (*comparator)(elem_type, elem_type);

struct people {
  int age;
};

#endifd_list(list, 0, print);
putchar('\n');

printf("%-45s", "remove the node at posistion 0: ");
```

```
remove_linked_list(list, 0, NULL);
foreach_linked_list(list, print);
putchar('\n');

printf("%-45s", "clear all nodes and destory the linked list: ");
destory_linked_list(&list);
foreach_linked_list(list, print);
putchar('\n');

for (int i = 0; i < size; ++i) {
free(peoples[i]);
}
}

void test_quicksort(void) {
size_t size = 10;
struct people *peoples[10] = {0};
for (int i = 0; i < size; ++i) {
peoples[i] = (struct people *)malloc(sizeof(struct people));
peoples[i]->age = rand() % (size * 10);
}

// use quick sort
printf("%-20s", "Before sorting: ");
for (int i = 0; i < size; ++i) {
print(peoples[i]);
}
putchar('\n');

printf("%-20s", "quick sort: ");
quicksort((elem_type *)peoples, size, cmp, print);
putchar('\n');

printf("%-20s", "After sorting: ");
for (int i = 0; i < size; ++i) {
print(peoples[i]);
}
```

```
putchar('\n');

// free array's elements' memory
for (int i = 0; i < size; ++i) {
free(peoples[i]);
}
}

void test_mirros(void) {
/** create a simple binary tree */
size_t size = 5;
struct tree_node *nodes[5] = {0};
for (long i = 0; i < size; ++i) {
nodes[i] = (struct tree_node *)malloc(sizeof(struct tree_node *) * size);
nodes[i]->elem = (struct people *)malloc(sizeof(struct people));
((struct people *)nodes[i]->elem)->age = i;
nodes[i]->left = NULL;
nodes[i]->right = NULL;
nodes[i]->parent = NULL;
}

nodes[0]->left = nodes[1];
nodes[0]->right = nodes[2];
nodes[1]->left = nodes[3];
nodes[1]->right = nodes[4];
nodes[1]->parent = nodes[0];
nodes[2]->parent = nodes[0];
nodes[3]->parent = nodes[1];
nodes[4]->parent = nodes[1];

printf("%-40s", "get the binary tree's recursive state: ");
morris(nodes[0], print);
putchar('\n');

printf("%-40s", "traversal through preorder: ");
morris_pre(nodes[0], print);
putchar('\n');
```

```
printf("%-40s", "traversal through inorder: ");
morris_in(nodes[0], print);
putchar('\n');
printf("%-40s", "traversal through postorder: ");
morris_post(nodes[0], print);
putchar('\n');

for (int i = 0; i < size; ++i) {
free(nodes[i]);
}
}

int main(void) {
srand((unsigned)time(NULL));

printf("\n\ntesting the linked list......\n");
test_list();

printf("\n\ntesting the quick sort......\n");
test_quicksort();

printf("\n\ntesting the binary tree cueing......\n");
test_mirros();
return 0;
}
```

def.h

```
#ifndef __DEF_H__
#define __DEF_H__

typedef void *elem_type;
typedef void (*callback)(elem_type);
typedef int (*comparator)(elem_type, elem_type);

struct people {
int age;
```

```
};

#endif
```

封面设计： 贾丽

地 址：中国河北省秦皇岛市河北大街 438 号
邮 编：066004
电 话：0335-8057068
传 真：0335-8057068
网 址：http://jwc.ysu.edu.cn