算法导论 Code

前言

这文档是根据《算法导论》第 2 版的伪代码用 C 语言写的代码，因为用到了 c99 的一些特性，如变长数组。需要用 gcc 指定选项-std=c99 编译，不支持 VC。

参考书：

算法导论第 2 版电子版:

http://www.kuaipan.cn/file/id_12008874588516081.html

C 语言程序设计_现代方法(第 2 版)电子版:

http://www.kuaipan.cn/file/id_12008874588516159.html

By 小鹏，QQ:562453464
2011.11.30

第 2 章 算法入门

2.1 插入排序

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
void insertion_sort(void *base, size_t elem_size, size_t n,
                    int (*comp) (const void *, const void *))
{
    char *cbase = base;
    char key[elem_size];
    for (size_t i = 1; i < n; i++) {
        memcpy(key, &cbase[i * elem_size], elem_size);
        /*把 base[i]插入到排好序的 base[0..i-1]中  */
        int j = i - 1;
        while (j >= 0 && comp(&cbase[j * elem_size], key) > 0) {
            memcpy(&cbase[(j + 1) * elem_size],
                   &cbase[j * elem_size], elem_size);
            j--;
        }
```

```
        memcpy(&cbase[(j + 1) * elem_size], key, elem_size);
    }
}

void swap(void *a, void *b, size_t elem_size)
{
    if(a==NULL||b==NULL||a==b)
        return;
    char temp[elem_size];      /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}
void randomized_in_place(void *array, size_t elem_size, int n)
{
    char *carray = array;
    for (int i = 0; i < n; i++) {
        int index = rand() % (n - i) + i;
        swap(&carray[i * elem_size], &carray[index * elem_size],
             elem_size);
    }
}

void print_array(int a[], int n)
{
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

int cmp_int(const void *p1, const void *p2)
{
    const int *pa = p1;
    const int *pb = p2;
    if (*pa < *pb)
        return -1;
    if (*pa == *pb)
        return 0;
    return 1;
}
int main(void)
{
    srand((unsigned)time(NULL));
```

```
    int a[10];
    for (int i = 0; i < 10; i++) {
        a[i] = i;
    }
    randomized_in_place(a, sizeof(int), 10);
    printf("排序前:\n");
    print_array(a, 10);
    insertion_sort(a, sizeof(int), 10, cmp_int);
    printf("排序后:\n");
    print_array(a, 10);
    return 0;
}
```

2.3 归并排序

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
void merge(void *base, size_t elem_size, size_t left, size_t middle,
       size_t right, void *max, int (*comp) (const void *, const void *),
       void *buff)
{
    char *cbase = base;
    char *cbuff = buff;
    size_t left_length = middle - left + 1;
    size_t right_length = right - middle;
    char *left_buff = cbuff;
    char *right_buff = &cbuff[(left_length + 1) * elem_size];
    for (size_t i = 0; i < left_length; i++) {
        memcpy(&left_buff[i * elem_size],
                &cbase[(left + i) * elem_size], elem_size);
    }
    memcpy(&left_buff[left_length * elem_size], max, elem_size);
    for (size_t i = 0; i < right_length; i++) {
        memcpy(&right_buff[i * elem_size],
                &cbase[(middle + 1 + i) * elem_size], elem_size);
    }
    memcpy(&right_buff[right_length * elem_size], max, elem_size);
    for (size_t k = left, i = 0, j = 0; k <= right; k++) {
        if (comp(&left_buff[i * elem_size], &right_buff[j * elem_size])
            <= 0) {
```

```c
                memcpy(&cbase[k * elem_size], &left_buff[i * elem_size],
                        elem_size);
                i++;
            } else {
                memcpy(&cbase[k * elem_size],
                        &right_buff[j * elem_size], elem_size);
                j++;
            }
        }
}

void merge_sort_buff(void *base, size_t elem_size, size_t left, size_t right,
                void *max, int (*comp) (const void *, const void *),
                void *buff)
{
    if (left < right) {
        size_t middle = (left + right) / 2;
        merge_sort_buff(base, elem_size, left, middle, max, comp, buff);
        merge_sort_buff(base, elem_size, middle + 1, right, max, comp,
                buff);
        merge(base, elem_size, left, middle, right, max, comp, buff);
    }
}

void merge_sort(void *base, size_t elem_size, size_t left, size_t right,
        void *max, int (*comp) (const void *, const void *))
{
    if (left >= right)
        return;
    size_t length = right - left + 1; /*数组的长度 */
    char buff[(length + 2) * elem_size];    /*+2 是因为要在缓存保存两个最大值
*/
    merge_sort_buff(base, elem_size, left, right, max, comp, buff);
}

void swap(void *a, void *b, size_t elem_size)
{
    if(a==NULL||b==NULL||a==b)
        return;
    char temp[elem_size];    /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}
```

```c
void randomized_in_place(void *array, size_t elem_size, int n)
{
	char *c_array = array;
	for (int i = 0; i < n; i++) {
		int index = rand() % (n - i) + i;
		swap(&c_array[i * elem_size], &c_array[index * elem_size],
			elem_size);
	}
}

void print_array(int a[], int n)
{
	for (int i = 0; i < n; i++) {
		printf("%d ", a[i]);
	}
	printf("\n");
}

int cmp_int(const void *p1, const void *p2)
{
	const int *pa = p1;
	const int *pb = p2;
	if (*pa < *pb)
		return -1;
	if (*pa == *pb)
		return 0;
	return 1;
}
int main(void)
{
	srand((unsigned)time(NULL));
	int a[10];
	for (int i = 0; i < 10; i++) {
		a[i] = i;
	}
	randomized_in_place(a, sizeof(int), 10);
	printf("排序前:\n");
	print_array(a, 10);
	int max_int = INT_MAX;
	merge_sort(a, sizeof(int), 0, 9, &max_int, cmp_int);
	printf("排序后:\n");
	print_array(a, 10);
	return 0;
}
```

第 5 章 概率分析和随机算法

5.1 雇用问题

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
void hire_assistant(int A[], int n)
{
	int best = 0;
	printf("hire:");
	for (int i = 0; i < n; i++) {
		if (A[i] > best) {
			best = A[i];
			printf("%d ", i);
		}
	}
	printf("\n");
}

int main()
{
	srand((unsigned)time(NULL));
	int a[10];
	for (int i = 0; i < 10; i++) {
		a[i] = rand() % 100;
		printf("%d ", a[i]);
	}
	printf("\n");
	hire_assistant(a, 10);
	return 0;
}
```

5.3 随机算法

5.3.1 通过排序产生随机排列数组

```
#include <stdio.h>
```

```
#include <limits.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
void merge(void *base, size_t elem_size, size_t left, size_t middle,
        size_t right, void *max, int (*comp) (const void *, const void *),
        void *buff)
{
    char *cbase = base;
    char *cbuff = buff;
    size_t left_length = middle - left + 1;
    size_t right_length = right - middle;
    char *left_buff = cbuff;
    char *right_buff = &cbuff[(left_length + 1) * elem_size];
    for (size_t i = 0; i < left_length; i++) {
        memcpy(&left_buff[i * elem_size],
                &cbase[(left + i) * elem_size], elem_size);
    }
    memcpy(&left_buff[left_length * elem_size], max, elem_size);
    for (size_t i = 0; i < right_length; i++) {
        memcpy(&right_buff[i * elem_size],
                &cbase[(middle + 1 + i) * elem_size], elem_size);
    }
    memcpy(&right_buff[right_length * elem_size], max, elem_size);
    for (size_t k = left, i = 0, j = 0; k <= right; k++) {
        if (comp(&left_buff[i * elem_size], &right_buff[j * elem_size])
             <= 0) {
            memcpy(&cbase[k * elem_size], &left_buff[i * elem_size],
                    elem_size);
            i++;
        } else {
            memcpy(&cbase[k * elem_size],
                    &right_buff[j * elem_size], elem_size);
            j++;
        }
    }
}

void merge_sort_buff(void *base, size_t elem_size, size_t left, size_t right,
                void *max, int (*comp) (const void *, const void *),
                void *buff)
{
    if (left < right) {
        size_t middle = (left + right) / 2;
```

```
        merge_sort_buff(base, elem_size, left, middle, max, comp, buff);
        merge_sort_buff(base, elem_size, middle + 1, right, max, comp,
                buff);
        merge(base, elem_size, left, middle, right, max, comp, buff);
    }
}

void merge_sort(void *base, size_t elem_size, size_t left, size_t right,
        void *max, int (*comp) (const void *, const void *))
{
    if (left >= right)
        return;
    size_t length = right - left + 1; /*数组的长度 */
    char buff[(length + 2) * elem_size];    /*+2 是因为要在缓存保存两个最大值
*/
    merge_sort_buff(base, elem_size, left, right, max, comp, buff);
}

struct rand_data {
    int rand_num;
    void *data;        /*捆定了一个数据的指针 */
};
int cmp_data(const void *p1, const void *p2)
{
    const struct rand_data *pa = p1;
    const struct rand_data *pb = p2;
    return pa->rand_num - pb->rand_num;
}

void permute_by_sorting(void *base, size_t elem_size, int length)
{
    char *cbase = base;
    /*把原来的数组复制一份 */
    char data_copy[elem_size * length];
    memcpy(data_copy, base, elem_size * length);
    struct rand_data rand_data_array[length];
    for (int i = 0; i < length; i++) {
        rand_data_array[i].rand_num =
             rand() % (length * length * length);
        rand_data_array[i].data = &data_copy[i * elem_size];
    }
    struct rand_data data_max = { INT_MAX, NULL };
    merge_sort(rand_data_array, sizeof(struct rand_data), 0, length - 1,
           &data_max, cmp_data);
```

```
    /*根据按随机数排序的结果把数据复制回原来的数组 */
    for (int i = 0; i < length; i++) {
        memcpy(&cbase[i * elem_size], rand_data_array[i].data,
               elem_size);
    }
}

void randomized_hire_assistant(int A[], int length)
{
    permute_by_sorting(A, sizeof(int), length);
    printf("hire:");
    int best = 0;
    for (int i = 0; i < length; i++) {
        if (A[i] > best) {
            best = A[i];
            printf("%d ", i);
        }
    }
    printf("\n");
}

int main()
{
    srand((unsigned)time(NULL));
    int a[10];
    for (int i = 0; i < 10; i++) {
        a[i] = i;
    }
    randomized_hire_assistant(a, 10);
    return 0;
}
```

5.3.2 通过原地排列产生随机排列数组

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
void swap(void *a, void *b, size_t elem_size)
{
    if(a==NULL||b==NULL||a==b)
        return;
    char temp[elem_size];    /*变长数组 */
```

```c
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}
void randomized_in_place(void *array, size_t elem_size, int length)
{
    char *carray = array;
    for (int i = 0; i < length; i++) {
        int rand_index = rand() % (length - i) + i;
        swap(&carray[i * elem_size], &carray[rand_index * elem_size],
                elem_size);
    }
}

void randomized_hire_assistant(int A[], int n)
{
    randomized_in_place(A, sizeof(int), n);
    int best = 0;
    printf("hire:");
    for (int i = 0; i < n; i++) {
        if (A[i] > best) {
            best = A[i];
            printf("%d ", i);
        }
    }
    printf("\n");
}

int main()
{
    srand((unsigned)time(NULL));
    int a[10];
    for (int i = 0; i < 10; i++) {
        a[i] = i;
    }
    randomized_hire_assistant(a, 10);
    return 0;
}
```

5.4.4 在线雇用问题

```c
#include <stdio.h>
#include <stdbool.h>
```

```c
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <string.h>
#include <limits.h>
int on_line_maximum(int A[], int n, int k)
{
	int best_score = -INT_MAX;
	for (int i = 0; i < k; i++) {
		if (A[i] > best_score) {
			best_score = A[i];
		}
	}
	for (int i = k; i < n; i++) {
		if (A[i] > best_score) {
			return i;
		}
	}
	return n - 1;
}

int main()
{
	srand((unsigned)time(NULL));
	int a[10];
	for (int i = 0; i < 10; i++) {
		a[i] = rand() % 100;
	}
	int n = on_line_maximum(a, 10, 10 / exp(1.0));
	/*测试 n 是不是最好的*/
	bool flag = true;
	for (int i = 0; i < 10; i++) {
		if (a[i] > a[n]) {
			flag = false;
		}
	}
	printf("%s\n",flag?"true":"false");
	return 0;
}
```

第 6 章 堆排序

6.4 堆排序算法

```
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <stdlib.h>
int parent(int i)
{
    return (i - 1) / 2;
}

int left_child(int i)
{
    return i * 2 + 1;
}

int right_child(int i)
{
    return i * 2 + 2;
}

void swap(void *a, void *b, size_t elem_size)
{
    if(a==NULL||b==NULL||a==b)
        return;
    char temp[elem_size];     /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}
void max_heapify(void *base, size_t elem_size, int i, int heap_size,
          int (*comp) (const void *, const void *))
{
    char *cbase = base;
    int left = left_child(i);
    int right = right_child(i);
    int largest = i;
    if (left < heap_size
        && comp(&cbase[largest * elem_size],
            &cbase[left * elem_size]) < 0) {
```

```c
            largest = left;
        }
        if (right < heap_size
             && comp(&cbase[largest * elem_size],
                     &cbase[right * elem_size]) < 0) {
            largest = right;
        }
        if (largest != i) {
            swap(&cbase[i * elem_size], &cbase[largest * elem_size],
                  elem_size);
            max_heapify(base, elem_size, largest, heap_size, comp);
        }
}

void build_max_heap(void *base, size_t elem_size, int length,
                int (*comp) (const void *, const void *))
{
        int heap_size = length;
        for (int i = parent(length - 1); i >= 0; i--) {
            max_heapify(base, elem_size, i, heap_size, comp);
        }
}

void heap_sort(void *base, size_t elem_size, int length,
               int (*comp) (const void *, const void *))
{
        char *cbase = base;
        build_max_heap(base, elem_size, length, comp);
        int heap_size = length;
        for (int i = length - 1; i > 0; i--) {
            swap(&cbase[i * elem_size], &cbase[0 * elem_size], elem_size);
            --heap_size;
            max_heapify(base, elem_size, 0, heap_size, comp);
        }
}

void randomized_in_place(void *array, size_t elem_size, int length)
{
        char *carray = array;
        for (int i = 0; i < length; i++) {
            int n_rand_index = rand() % (length - i) + i;
            swap(&carray[i * elem_size], &carray[n_rand_index * elem_size],
                  elem_size);
        }
```

```
}

void print_array(int a[], int length)
{
	for (int i = 0; i < length; i++) {
		printf("%d ", a[i]);
	}
	printf("\n");
}

int cmp_int(const void *p1, const void *p2)
{
	const int *pa = p1;
	const int *pb = p2;
	if (*pa < *pb)
		return -1;
	if (*pa == *pb)
		return 0;
	return 1;
}
int main(void)
{
	srand((unsigned)time(NULL));
	int a[10];
	for (int i = 0; i < 10; i++) {
		a[i] = i;
	}
	randomized_in_place(a, sizeof(int), 10);
	printf("排序前:\n");
	print_array(a, 10);
	heap_sort(a, sizeof(int), 10, cmp_int);
	printf("排序后:\n");
	print_array(a, 10);
	return 0;
}
```

6.5 优先级队列

6.5.1 优先级队列

```
#include <stdio.h>
#include <stdbool.h>
```

```
#include <stdlib.h>
#include <string.h>
typedef struct priority_queue_type *priority_queue;
struct priority_queue_type {
	int heap_size;
	void **array;
	int (*comp) (const void *, const void *);
};
int parent(int i)
{
	return (i - 1) / 2;
}

int left_child(int i)
{
	return i * 2 + 1;
}

int right_child(int i)
{
	return i * 2 + 2;
}

void swap(void *a, void *b, size_t elem_size)
{
	if(a==NULL||b==NULL||a==b)
		return;
	char temp[elem_size];	/*变长数组 */
	memcpy(temp, a, elem_size);
	memcpy(a, b, elem_size);
	memcpy(b, temp, elem_size);
}

void heapify(priority_queue pq, int i)
{
	int left = left_child(i);
	int right = right_child(i);
	int largest = i;
	if (left < pq->heap_size
	     && pq->comp(pq->array[largest], pq->array[left]) < 0) {
		largest = left;
	}
	if (right < pq->heap_size
	     && pq->comp(pq->array[largest], pq->array[right]) < 0) {
```

```
		largest = right;
	}
	if (largest != i) {
		swap(&pq->array[i], &pq->array[largest], sizeof(void *));
		heapify(pq, largest);
	}
}

void fix_up(priority_queue pq, int i)
{
	while (i > 0 && pq->comp(pq->array[parent(i)], pq->array[i]) < 0) {
		swap(&pq->array[parent(i)], &pq->array[i], sizeof(void *));
		i = parent(i);
	}
}

priority_queue priority_queue_create(int n_length,
				int (*comp) (const void *, const void *))
{
	priority_queue pq = malloc(sizeof(struct priority_queue_type));
	pq->array = malloc(sizeof(void *) * n_length);
	pq->heap_size = 0;
	pq->comp = comp;
	return pq;
}

void *priority_queue_top(priority_queue pq)
{
	return pq->array[0];
}

/*去掉并返回堆的第一个元素 */
void *priority_queue_extract_top(priority_queue pq)
{
	swap(&pq->array[0], &pq->array[pq->heap_size - 1], sizeof(void *));
	--pq->heap_size;
	heapify(pq, 0);
	return pq->array[pq->heap_size];
}

/*把元素 key 插入队列 */
void priority_queue_insert(priority_queue pq, void *key)
{
	++pq->heap_size;
```

```c
	int i = pq->heap_size - 1;
	memcpy(&pq->array[i], &key, sizeof(void *));
	fix_up(pq, i);
}

bool priority_queue_is_empty(priority_queue pq)
{
	return pq->heap_size == 0;
}

void priority_queue_destroy(priority_queue pq, void (*free_key) (void *))
{
	while (!priority_queue_is_empty(pq)) {
		void *p = priority_queue_extract_top(pq);
		free_key(p);
	}
	free(pq->array);
	free(pq);
}

int cmp_int(const void *p1, const void *p2)
{
	const int *pa = p1;
	const int *pb = p2;
	if (*pa < *pb)
		return -1;
	if (*pa == *pb)
		return 0;
	return 1;
}
int main()
{
	priority_queue pq = priority_queue_create(10, cmp_int);
	for (int i = 0; i < 10; i++) {
		int *p = malloc(sizeof(int));
		*p = i;
		priority_queue_insert(pq, p);
	}
	printf("最大堆结果:\n");
	while (!priority_queue_is_empty(pq)) {
		int *p = priority_queue_extract_top(pq);
		printf("%d ", *p);
		free(p);
	}
```

```
    printf("\n");
    priority_queue_destroy(pq, free);
    return 0;
}
```

6.5.2 基于索引堆的优先队列

```
#include <stdio.h>
#include <limits.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
/*基于索引堆的优先队列*/
typedef struct priority_queue_index_type *priority_queue;
struct priority_queue_index_type {
    int heap_size;
    int *index_array;
    int *index_pos_array;/*这个数组记录了索引在堆中位置 */
    void *data_array;
    size_t elem_size;
    int (*comp) (const void *, const void *);
};
static int parent(int i)
{
    return (i - 1) / 2;
}

static int left_child(int i)
{
    return i * 2 + 1;
}

static int right_child(int i)
{
    return i * 2 + 2;
}

void swap(void *a, void *b, size_t elem_size)
{
    if(a==NULL||b==NULL||a==b)
        return;
    char temp[elem_size];     /*变长数组 */
    memcpy(temp, a, elem_size);
```

```
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}
static void swap_index(priority_queue pq, int i, int j)
{
    swap(&pq->index_pos_array[i], &pq->index_pos_array[j], sizeof(int));
    pq->index_array[pq->index_pos_array[i]] = i;
    pq->index_array[pq->index_pos_array[j]] = j;
}

/*最小堆用的比较函数*/
static bool compare(priority_queue pq, int left, int right)
{
    if (pq->data_array == NULL)
        return false;
    char *pc_array = pq->data_array;
    return pq->comp(&pc_array[left * pq->elem_size],
            &pc_array[right * pq->elem_size]) > 0;
}

static void heapify(priority_queue pq, int i)
{
    int left = left_child(i);
    int right = right_child(i);
    int largest = i;
    if (left < pq->heap_size
         && compare(pq, pq->index_array[largest], pq->index_array[left])) {
        largest = left;
    }
    if (right < pq->heap_size
         && compare(pq, pq->index_array[largest], pq->index_array[right])) {
        largest = right;
    }
    if (largest != i) {
        swap_index(pq, pq->index_array[i], pq->index_array[largest]);
        heapify(pq, largest);
    }
}

static void fix_up(priority_queue pq, int i)
{
    while (i > 0
            && compare(pq, pq->index_array[parent(i)], pq->index_array[i])) {
        swap_index(pq, pq->index_array[parent(i)], pq->index_array[i]);
```

```
        i = parent(i);
    }
}

priority_queue priority_queue_create(void *p_data_array, size_t elem_size,
                        int length, int (*comp) (const void *,
                                    const void *))
{
    priority_queue pq = malloc(sizeof(struct priority_queue_index_type));
    pq->index_array = malloc(sizeof(int) * length);
    pq->index_pos_array = malloc(sizeof(int) * length);
    pq->data_array = p_data_array;
    pq->elem_size = elem_size;
    pq->heap_size = 0;
    pq->comp = comp;
    return pq;
}

void priority_queue_destroy(priority_queue pq)
{
    free(pq->index_array);
    free(pq->index_pos_array);
    free(pq);
}

int priority_queue_top(priority_queue pq)
{
    return pq->index_array[0];
}

/*去掉并返回堆的第一个元素 */
int priority_queue_extract_top(priority_queue pq)
{
    swap_index(pq, pq->index_array[0], pq->index_array[pq->heap_size - 1]);
    --pq->heap_size;
    heapify(pq, 0);
    return pq->index_array[pq->heap_size];
}

/*把元素的索引插入队列 */
void priority_queue_insert(priority_queue pq, int index)
{
    ++pq->heap_size;
    int i = pq->heap_size - 1;
```

```
    pq->index_array[i] = index;
    pq->index_pos_array[index] = i;
    fix_up(pq, i);
}

bool priority_queue_is_empty(priority_queue pq)
{
    return pq->heap_size == 0;
}

/*下标为 index 的数据修改了，调用这个函数来修复索引堆*/
void priority_queue_change_index(priority_queue pq, int index)
{
    fix_up(pq, pq->index_pos_array[index]);
    heapify(pq, pq->index_pos_array[index]);
}

int cmp_int(const void *p1, const void *p2)
{
    const int *pa = p1;
    const int *pb = p2;
    if (*pa < *pb)
        return -1;
    if (*pa == *pb)
        return 0;
    return 1;
}
int g_array[10];
int main()
{
    priority_queue pq =
        priority_queue_create(g_array, sizeof(int), 10, cmp_int);
    for (int i = 0; i < 10; i++) {
        g_array[i] = 100+i;
        /*把索引 i 插入到索引堆 pq */
        priority_queue_insert(pq, i);
    }
    /*测试 chang_index 函数 ,修过索引上的数据，然后修复索引堆*/
    int change_index = 5;
    g_array[change_index] = INT_MAX;
    priority_queue_change_index(pq, change_index);

    printf("最小堆结果:\n");
    while (!priority_queue_is_empty(pq)) {
```

```
        printf("%d ", g_array[priority_queue_extract_top(pq)]);
    }
    printf("\n");
    priority_queue_destroy(pq);
    return 0;
}
```

第 7 章 快速排序

7.1 快速排序的描述

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
void swap(void *a, void *b, size_t elem_size)
{
    if(a==NULL||b==NULL||a==b)
        return;
    char temp[elem_size];    /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}
int partition(void *base, size_t elem_size, int p, int r,
              int (*comp) (const void *, const void *))
{
    char *cbase = base;
    void *key = &cbase[r * elem_size];
    int i = p - 1;
    for (int j = p; j < r; j++) {
        if (comp(&cbase[j * elem_size], key) <= 0) {
            ++i;
            swap(&cbase[i * elem_size], &cbase[j * elem_size], elem_size);
        }
    }
    swap(&cbase[(i + 1) * elem_size], key, elem_size);
    return i + 1;
}

void quick_sort(void *base, size_t elem_size, int p, int r,
```

```
            int (*comp) (const void *, const void *))
{
    if (p < r) {
        int q = partition(base, elem_size, p, r, comp);
        quick_sort(base, elem_size, p, q - 1, comp);
        quick_sort(base, elem_size, q + 1, r, comp);
    }
}

void randomized_in_place(void *array, size_t elem_size, int length)
{
    char *carray = array;
    for (int i = 0; i < length; i++) {
        int rand_index = rand() % (length - i) + i;
        swap(&carray[i * elem_size], &carray[rand_index * elem_size],
              elem_size);
    }
}

void print_array(int a[], int length)
{
    for (int i = 0; i < length; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

int cmp_int(const void *p1, const void *p2)
{
    const int *pa = p1;
    const int *pb = p2;
    if (*pa < *pb)
        return -1;
    if (*pa == *pb)
        return 0;
    return 1;
}
int main(void)
{
    srand((unsigned)time(NULL));
    int a[10];
    for (int i = 0; i < 10; i++) {
        a[i] = i;
    }
```

```
    randomized_in_place(a, sizeof(int), 10);
    printf("排序前:\n");
    print_array(a, 10);
    quick_sort(a, sizeof(int), 0, 9, cmp_int);
    printf("排序后:\n");
    print_array(a, 10);
    return 0;
}
```

7.3 快速排序的随机化版本

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
void swap(void *a, void *b, size_t elem_size)
{
    if(a==NULL||b==NULL||a==b)
        return;
    char temp[elem_size];    /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}
int partition(void *base, size_t elem_size, int p, int r,
           int (*comp) (const void *, const void *))
{
    char *cbase = base;
    void *key = &cbase[r * elem_size];
    int i = p - 1;
    for (int j = p; j < r; j++) {
        if (comp(&cbase[j * elem_size], key) <= 0) {
            ++i;
            swap(&cbase[i * elem_size], &cbase[j * elem_size], elem_size);
        }
    }
    swap(&cbase[(i + 1) * elem_size], key, elem_size);
    return i + 1;
}

int randomized_partition(void *base, size_t elem_size, int p, int r,
            int (*comp) (const void *, const void *))
{
```

```c
    char *cbase = base;
    int i = rand() % (r - p + 1) + p;
    swap(&cbase[r * elem_size], &cbase[i * elem_size], elem_size);
    return partition(base, elem_size, p, r, comp);
}

void randomize_quick_sort(void *base, size_t elem_size, int p, int r,
            int (*comp) (const void *, const void *))
{
    if (p < r) {
        int q = randomized_partition(base, elem_size, p, r, comp);
        randomize_quick_sort(base, elem_size, p, q - 1, comp);
        randomize_quick_sort(base, elem_size, q + 1, r, comp);
    }
}

void randomized_in_place(void *array, size_t elem_size, int length)
{
    char *carray = array;
    for (int i = 0; i < length; i++) {
        int n_rand_index = rand() % (length - i) + i;
        swap(&carray[i * elem_size], &carray[n_rand_index * elem_size],
             elem_size);
    }
}

void print_array(int a[], int length)
{
    for (int i = 0; i < length; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

int cmp_int(const void *p1, const void *p2)
{
    const int *pa = p1;
    const int *pb = p2;
    if (*pa < *pb)
        return -1;
    if (*pa == *pb)
        return 0;
    return 1;
}
```

```
int main(void)
{
	srand((unsigned)time(NULL));
	int a[10];
	for (int i = 0; i < 10; i++) {
		a[i] = i;
	}
	randomized_in_place(a, sizeof(int), 10);
	printf("排序前:\n");
	print_array(a, 10);
	randomize_quick_sort(a, sizeof(int), 0, 9, cmp_int);
	printf("排序后:\n");
	print_array(a, 10);
	return 0;
}
```

第 8 章 线性时间排序

8.2 计数排序

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
void counting_sort(int A[], int n, int k)
{
	int *C = malloc(sizeof(int) * (k + 1));
	for (int i = 0; i <= k; i++) {
		C[i] = 0;
	}
	for (int i = 0; i < n; i++) {
		++C[A[i]];
	}
	for (int i = 1; i <= k; i++) {
		C[i] += C[i - 1];
	}
	int *B = malloc(sizeof(int) * n);
	for (int i = n - 1; i >= 0; i--) {
		B[C[A[i]] - 1] = A[i];
		--C[A[i]];
	}
```

```
    for (int i = 0; i < n; i++) {
        A[i] = B[i];
    }
    free(C);
    free(B);
}

void swap(void *a, void *b, size_t elem_size)
{
    if(a==NULL||b==NULL||a==b)
        return;
    char temp[elem_size];      /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}
void randomized_in_place(void *array, size_t elem_size, int n)
{
    char *c_array = array;
    for (int i = 0; i < n; i++) {
        int index = rand() % (n - i) + i;
        swap(&c_array[i * elem_size], &c_array[index * elem_size],
              elem_size);
    }
}

void print_array(int a[], int n)
{
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

int main(void)
{
    srand((unsigned)time(NULL));
    int a[10];
    for (int i = 0; i < 10; i++) {
        a[i] = rand() % 10;
    }
    randomized_in_place(a, sizeof(int), 10);
    printf("排序前:\n");
    print_array(a, 10);
```

```
    counting_sort(a, 10, 9);
    printf("排序后:\n");
    print_array(a, 10);
    return 0;
}
```

8.3 基数排序

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
int digit(int n, int w)
{
    static int base_array[20];
    if (base_array[0] == 0)     //求进制位的基值
    {
        base_array[0] = 1;
        for (int i = 1; i < 20; i++) {
            base_array[i] = base_array[i - 1] * 10;
        }
    }
    int n_base = base_array[w - 1];
    return n / n_base % 10;
}

void counting_sort(int A[], int n, int k, int w)
{
    int *C = malloc(sizeof(int) * (k + 1));
    for (int i = 0; i <= k; i++) {
        C[i] = 0;
    }
    for (int i = 0; i < n; i++) {
        ++C[digit(A[i], w)];
    }
    for (int i = 1; i <= k; i++) {
        C[i] += C[i - 1];
    }
    int *B = malloc(sizeof(int) * n);
    for (int i = n - 1; i >= 0; i--) {
        B[C[digit(A[i], w)] - 1] = A[i];
        --C[digit(A[i], w)];
```

```
    }
    for (int i = 0; i < n; i++) {
        A[i] = B[i];
    }
    free(C);
    free(B);
}

void radix_sort(int A[], int n, int d)
{
    for (int i = 1; i <= d; i++) {
        counting_sort(A, n, 9, i);
    }
}

void swap(void *a, void *b, size_t elem_size)
{
    if(a==NULL||b==NULL||a==b)
        return;
    char temp[elem_size];    /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}
void randomized_in_place(void *array, size_t elem_size, int n)
{
    char *c_array = array;
    for (int i = 0; i < n; i++) {
        int index = rand() % (n - i) + i;
        swap(&c_array[i * elem_size], &c_array[index * elem_size],
            elem_size);
    }
}

void print_array(int a[], int n)
{
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

int main(void)
{
```

```
    srand((unsigned)time(NULL));
    int a[10];
    for (int i = 0; i < 10; i++) {
        a[i] = rand()%1000;
    }
    randomized_in_place(a, sizeof(int), 10);
    printf("排序前:\n");
    print_array(a, 10);
    radix_sort(a,10, 3);
    printf("排序后:\n");
    print_array(a, 10);
    return 0;
}
```

8.4 桶排序

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
struct node {
    float value;
    struct node *next;
};
void node_ini(struct node *pnode, float value)
{
    pnode->value = value;
    pnode->next = NULL;
}

void insert_node(struct node *head, struct node *pnode)
{
    struct node *p = head;
    while (p->next != NULL) {
        if (p->next->value > pnode->value) {
            break;
        } else {
            p = p->next;
        }
    }
    pnode->next = p->next;
    p->next = pnode;
}
```

```
void bucket_sort(float A[], int n)
{
	struct node *node_array = malloc(sizeof(struct node) * n);
	for(int i=0;i<n;i++)
		node_ini(&node_array[i],0);
	for (int i = 0; i < n; i++) {
		struct node *p = malloc(sizeof(struct node));
		node_ini(p, A[i]);
		insert_node(&node_array[(int)(n * A[i])], p);
	}
	int k = 0;
	for (int i = 0; i < n; i++) {
		for (struct node * p = node_array[i].next; p != NULL;) {
			A[k++] = p->value;
			struct node *del = p;
			p = p->next;
			free(del);
		}
	}
	free(node_array);
}

void swap(void *a, void *b, size_t elem_size)
{
	if (a == NULL || b == NULL || a == b)
		return;
	char temp[elem_size];	/*变长数组 */
	memcpy(temp, a, elem_size);
	memcpy(a, b, elem_size);
	memcpy(b, temp, elem_size);
}

void randomized_in_place(void *array, size_t elem_size, int n)
{
	char *c_array = array;
	for (int i = 0; i < n; i++) {
		int index = rand() % (n - i) + i;
		swap(&c_array[i * elem_size], &c_array[index * elem_size],
		     elem_size);
	}
}

void print_array(float a[], int n)
```

```
{
	for (int i = 0; i < n; i++) {
		printf("%.2f ", a[i]);
	}
	printf("\n");
}

int main(void)
{
	srand((unsigned)time(NULL));
	float a[10];
	for (int i = 0; i < 10; i++) {
		a[i] = rand() / (float)RAND_MAX;
	}
	randomized_in_place(a, sizeof(int), 10);
	printf("排序前:\n");
	print_array(a, 10);
	bucket_sort(a, 10);
	printf("排序后:\n");
	print_array(a, 10);
	return 0;
}
```

第 9 章 中位数和顺序统计学

9.1 最小值和最大值

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
int minimum(int A[], int n)
{
	int min = A[0];
	for (int i = 1; i < n; i++) {
		if (min > A[i]) {
			min = A[i];
		}
	}
	return min;
}
```

```
void min_and_max(int A[], int n, int *min, int *max)
{
	int i;
	if (n % 2 == 1) {
		*min = A[0];
		*max = A[0];
		i = 1;
	} else {
		if (A[0] > A[1]) {
			*max = A[0];
			*min = A[1];
		} else {
			*max = A[1];
			*min = A[0];
		}
		i = 2;
	}
	for (; i < n; i += 2) {
		if (A[i] > A[i + 1]) {
			if (A[i] > *max) {
				*max = A[i];
			}
			if (A[i + 1] < *min) {
				*min = A[i + 1];
			}
		} else {
			if (A[i + 1] > *max) {
				*max = A[i + 1];
			}
			if (A[i] < *min) {
				*min = A[i];
			}
		}
	}
}

void print_array(int a[], int n)
{
	for (int i = 0; i < n; i++) {
		printf("%d ", a[i]);
	}
	printf("\n");
}
```

```
int main()
{
	srand((unsigned)time(NULL));
	int a[10];
	for (int i = 0; i < 10; i++) {
		a[i] = rand() % 100;
	}
	print_array(a,10);
	printf("最小元素是:%d\n",minimum(a,10));
	int min;
	int max;
	min_and_max(a,10,&min,&max);
	printf("最小和最大元素是:%d,%d\n",min,max);
}
```

9.2 以期望线性时间做选择

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
void swap(void *a, void *b, size_t elem_size)
{
	if(a==NULL||b==NULL||a==b)
		return;
	char temp[elem_size];	/*变长数组 */
	memcpy(temp, a, elem_size);
	memcpy(a, b, elem_size);
	memcpy(b, temp, elem_size);
}
int partition(void *base, size_t elem_size, int p, int r,
		int (*comp) (const void *, const void *))
{
	char *cbase = base;
	void *key = &cbase[r * elem_size];
	int i = p - 1;
	for (int j = p; j < r; j++) {
		if (comp(&cbase[j * elem_size], key) <= 0) {
			++i;
			swap(&cbase[i * elem_size], &cbase[j * elem_size],
			     elem_size);
		}
```

```
	}
	swap(&cbase[(i + 1) * elem_size], key, elem_size);
	return i + 1;
}

int randomized_partition(void *base, size_t elem_size, int p, int r,
			int (*comp) (const void *, const void *))
{
	char *cbase = base;
	int i = rand() % (r - p + 1) + p;
	swap(&cbase[r * elem_size], &cbase[i * elem_size], elem_size);
	return partition(base, elem_size, p, r, comp);
}

void *randomized_select(void *base, size_t elem_size, int p, int r, int i,
			int (*comp) (const void *, const void *))
{
	char *cbase = base;
	if (p == r)
		return &cbase[p * elem_size];
	int q = randomized_partition(base, elem_size, p, r, comp);
	int k = q - p + 1;
	if (i == k) {
		return &cbase[q * elem_size];
	} else if (i < k) {
		return randomized_select(base, elem_size, p, q - 1, i, comp);
	} else {
		return randomized_select(base, elem_size, q + 1, r, i - k,
					comp);
	}
}

void print_array(int a[], int length)
{
	for (int i = 0; i < length; i++) {
		printf("%d ", a[i]);
	}
	printf("\n");
}

int cmp_int(const void *p1, const void *p2)
{
	const int *pa = p1;
	const int *pb = p2;
```

```
    if (*pa < *pb)
        return -1;
    if (*pa == *pb)
        return 0;
    return 1;
}
void randomize_quick_sort(void *base, size_t elem_size, int p, int r,
                int (*comp) (const void *, const void *))
{
    if (p < r) {
        int q = randomized_partition(base, elem_size, p, r, comp);
        randomize_quick_sort(base, elem_size, p, q - 1, comp);
        randomize_quick_sort(base, elem_size, q + 1, r, comp);
    }
}

int main()
{
    srand((unsigned)time(NULL));
    int a[10];
    for (int i = 0; i < 10; i++) {
        a[i] = rand() % 100;
    }
    printf("原数组:\n");
    print_array(a, 10);
    int order = 3;
    int *select_value = randomized_select(a, sizeof(int), 0, 9, order, cmp_int);
    randomize_quick_sort(a, sizeof(int),0,9, cmp_int);
    printf("第%d 小的元素是:%d\n", order, *select_value);
    printf("跟排序后的相应位置的值比较:%s\n",
            *select_value == a[order - 1] ? "相等" : "不相等");
    return 0;
}
```

9.3 最坏情况线性时间的选择

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
void insertion_sort(void *base, size_t elem_size, size_t n,
                int (*comp) (const void *, const void *))
{
```

```
    char *cbase = base;
    char key[elem_size];
    for (size_t i = 1; i < n; i++) {
        memcpy(key, &cbase[i * elem_size], elem_size);
        /*把 base[i]插入到排好序的 base[0..i-1]中 */
        int j = i - 1;
        while (j >= 0 && comp(&cbase[j * elem_size], key) > 0) {
            memcpy(&cbase[(j + 1) * elem_size],
                    &cbase[j * elem_size], elem_size);
            j--;
        }
        memcpy(&cbase[(j + 1) * elem_size], key, elem_size);
    }
}

void swap(void *a, void *b, size_t elem_size)
{
    if (a == NULL || b == NULL || a == b)
        return;
    char temp[elem_size];     /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}

int partition(void *base, size_t elem_size, int p, int r, void *pivot,
            int (*comp) (const void *, const void *))
{
    char *cbase = base;
    void *key = pivot;
    int i = p - 1;
    int pivot_pos = p;     /*主元的位置 */
    for (int j = p; j < r; j++) {
        if (comp(&cbase[j * elem_size], key) == 0)
            pivot_pos = j;     /*记录主元的位置 */
        if (comp(&cbase[j * elem_size], key) <= 0) {
            ++i;
            swap(&cbase[i * elem_size], &cbase[j * elem_size],
                  elem_size);
        }
    }
    swap(&cbase[(i + 1) * elem_size], &cbase[pivot_pos * elem_size],
          elem_size);
    return i + 1;
```

```
}

void *select(void *base, size_t elem_size, int p, int r, int order,
            int (*comp) (const void *, const void *))
{
    char *cbase = base;
    if (p == r)
        return &cbase[p * elem_size];
    int n = r - p + 1;
    int array_count = n % 5 == 0 ? n / 5 : n / 5 + 1;
    char array[elem_size * array_count];
    for (int i = 0; i < array_count; i++) {
        int begin = p + i * 5;
        int end = begin + 4 < r ? begin + 4 : r;
        insertion_sort(&cbase[begin * elem_size], elem_size,
                       end - begin + 1, comp);
        int middle=begin+(end-begin)/2;
        memcpy(&array[i * elem_size], &cbase[middle * elem_size],
               elem_size);
    }
    void *x =
         select(array, elem_size, 0, array_count - 1, (array_count + 1) / 2,
            comp);
    /*用求得的划分的元素 x 来划分数组 A，保证对数组的划分是好的划分 */
    int q = partition(base, elem_size, p, r, x, comp);
    int k = q - p + 1;
    if (order == k) {
        return &cbase[q * elem_size];
    } else if (order < k) {
        return select(base, elem_size, p, q - 1, order, comp);
    } else {
        return select(base, elem_size, q + 1, r, order - k, comp);
    }
}

void print_array(int a[], int length)
{
    for (int i = 0; i < length; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
}

int cmp_int(const void *p1, const void *p2)
```

```
{
    const int *pa = p1;
    const int *pb = p2;
    if (*pa < *pb)
        return -1;
    if (*pa == *pb)
        return 0;
    return 1;
}

int main()
{
    srand((unsigned)time(NULL));
    int a[10];
    for (int i = 0; i < 10; i++) {
        a[i] = rand() % 100;
    }
    printf("原数组:\n");
    print_array(a, 10);
    int order = 3;
    int *select_value = select(a, sizeof(int), 0, 9, order, cmp_int);
    insertion_sort(a, sizeof(int), 10, cmp_int);
    printf("第%d 小的元素是:%d\n", order, *select_value);
    printf("跟排序后的相应位置的值比较:%s\n",
            *select_value == a[order - 1] ? "相等" : "不相等");
    return 0;
}
```

第 10 章 基本数据结构

10.1 栈和队列

10.1.1 栈

10.1.1.1 基于数组实现

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
```

```c
typedef struct stack_type *stack;
struct stack_type {
	int top;
	int num;
	void **array;
};
stack stack_create(int num)
{
	stack s = malloc(sizeof(struct stack_type));
	s->top = -1;
	s->num = num;
	s->array = malloc(sizeof(void *) * num);
	return s;
}

bool stack_is_empty(stack s)
{
	return s->top == -1;
}

bool stack_is_full(stack s)
{
	return s->top == s->num - 1;
}

void stack_push(stack s, void *x)
{
	s->array[++s->top] = x;
}

void *stack_pop(stack s)
{
	return s->array[s->top--];
}

void stack_destroy(stack s, void (*free_key) (void *))
{
	while (!stack_is_empty(s)) {
		void *p = stack_pop(s);
		free_key(p);
	}
	free(s->array);
	free(s);
}
```

```
int main()
{
    stack s = stack_create(10);
    for (int i = 0; i < 10; i++) {
        int *p = malloc(sizeof(int));
        *p = i;
        stack_push(s, p);
    }
    printf("stack is full?%s\n", stack_is_full(s) ? "true" : "false");
    while (!stack_is_empty(s)) {
        int *p = stack_pop(s);
        printf("%d ", *p);
        free(p);
    }
    printf("\n");
    stack_destroy(s, free);
    return 0;
}
```

10.1.1.2 基于链表实现

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct stack_type *stack;
struct stack_node {
    void *key;
    struct stack_node *next;
};
struct stack_type {
    struct stack_node *head;
};
void stack_node_ini(struct stack_node *n, void *key)
{
    n->key = key;
    n->next = NULL;
}

stack stack_create()
{
    stack s = malloc(sizeof(struct stack_type));
```

```c
	s->head = NULL;
	return s;
}


bool stack_is_empty(stack s)
{
	return s->head == NULL;
}

void stack_push(stack s, void *x)
{
	struct stack_node *node = malloc(sizeof(struct stack_node));
	stack_node_ini(node, x);
	node->next = s->head;
	s->head = node;
}

void *stack_pop(stack s)
{
	struct stack_node *p = s->head;
	s->head = s->head->next;
	void *key = p->key;
	free(p);
	return key;
}

void stack_destroy(stack s, void (*free_key) (void *))
{
	while (!stack_is_empty(s)) {
		void *p = stack_pop(s);
		free_key(p);
	}
	free(s);
}

int main()
{
	stack s = stack_create();
	for (int i = 0; i < 10; i++) {
		int *p = malloc(sizeof(int));
		*p = i;
		stack_push(s, p);
	}
```

```
	while (!stack_is_empty(s)) {
		int *p = stack_pop(s);
		printf("%d ", *p);
		free(p);
	}
	printf("\n");
	stack_destroy(s, free);
	return 0;
}
```

10.1.2 队列

10.1.2.1 基于数组实现

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct queue_type *queue;
struct queue_type {
	int num;
	int head;
	int tail;
	void **array;
};
queue queue_create(int num)
{
	queue q = malloc(sizeof(struct queue_type));
	/*num 比要存放在队列中的元素个数多一，是为了区分队列满和空的状态 */
	q->num = num + 1;
	q->head = 0;
	q->tail = 0;
	q->array = malloc(sizeof(void *) * num);
	return q;
}

bool queue_is_empty(queue q)
{
	return q->head == q->tail;
}

bool queue_is_full(queue q)
{
```

```
    return q->head == (q->tail + 1) % q->num;
}

void queue_en_queue(queue q, void *x)
{
    q->array[q->tail++] = x;
    q->tail = q->tail % q->num;
}

void *queue_de_queue(queue q)
{
    void *x = q->array[q->head++];
    q->head = q->head % q->num;
    return x;
}

void queue_destroy(queue q,void (*free_key)(void *))
{
    while (!queue_is_empty(q)) {
        void *p = queue_de_queue(q);
        free_key(p);
    }
    free(q->array);
    free(q);
}

int main()
{
    queue q = queue_create(10);
    for (int i = 0; i < 10; i++) {
        int *p = malloc(sizeof(int));
        *p = i;
        queue_en_queue(q, p);
    }
    printf("queue is full?:%s\n", queue_is_full(q) ? "true" : "false");
    while (!queue_is_empty(q)) {
        int *p = queue_de_queue(q);
        printf("%d ", *p);
        free(p);
    }
    printf("\n");
    queue_destroy(q,free);
    return 0;
}
```

10.1.2.2 基于链表实现

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct queue_type *queue;
struct queue_node {
	void *key;
	struct queue_node *next;
};

struct queue_type {
	struct queue_node *head;
	struct queue_node *tail;
};

void queue_node_ini(struct queue_node *node, void *key)
{
	node->key = key;
	node->next = NULL;
}

queue queue_create()
{
	queue q = malloc(sizeof(struct queue_type));
	q->head = NULL;
	q->tail = NULL;
	return q;
}

bool queue_is_empty(queue q)
{
	return q->head == NULL;
}

void queue_en_queue(queue q, void *x)
{
	struct queue_node *p = malloc(sizeof(struct queue_node));
	queue_node_ini(p, x);
	if (q->head == NULL) {
		q->head = p;
```

```
        q->tail = p;
    } else {
        q->tail->next = p;
        q->tail = p;
    }
}

void *queue_de_queue(queue q)
{
    void *key = q->head->key;
    struct queue_node *p = q->head;
    q->head = q->head->next;
    free(p);
    return key;
}

void queue_destroy(queue q, void (*free_key) (void *))
{
    while (!queue_is_empty(q)) {
        void *p = queue_de_queue(q);
        free_key(p);
    }
    free(q);
}

int main()
{
    queue q = queue_create();
    for (int i = 0; i < 10; i++) {
        int *p = malloc(sizeof(int));
        *p = i;
        queue_en_queue(q, p);
    }
    while (!queue_is_empty(q)) {
        int *p = queue_de_queue(q);
        printf("%d ", *p);
        free(p);
    }
    printf("\n");
    queue_destroy(q,free);
    return 0;
}
```

10.2 链表

10.2.1 双链表

```
#include <stdio.h>
#include <stdlib.h>
typedef struct list_type *list;
struct list_node {
	void *key;
	struct list_node *prev;
	struct list_node *next;
};
struct list_type {
	struct list_node *head;
};
void list_node_ini(struct list_node *p, void *key)
{
	p->key = key;
	p->prev = NULL;
	p->next = NULL;
}

list list_create()
{
	list l = malloc(sizeof(struct list_type *));
	l->head = NULL;
	return l;
}

void list_destroy(list l, void (*free_key) (void *))
{
	struct list_node *x = l->head;
	while (x != NULL) {
		struct list_node *del = x;
		x = x->next;
		free_key(del->key);
		free(del);
	}
	free(l);
}

struct list_node *list_search(list l, void *k,
			int (*comp) (const void *, const void *))
```

```
{
        struct list_node *x = l->head;
        while (x != NULL && comp(x->key, k) != 0) {
                x = x->next;
        }
        return x;
}

void list_insert(list l, struct list_node *x)
{
        x->next = l->head;
        if (l->head != NULL) {
                l->head->prev = x;
        }
        l->head = x;
        x->prev = NULL;
}

void list_delete(list l, struct list_node *x)
{
        if (x->prev != NULL) {
                x->prev->next = x->next;
        } else {
                l->head = x->next;
        }
        if (x->next != NULL) {
                x->next->prev = x->prev;
        }
}

void list_display(list l, void (*print_key) (const void *))
{
        struct list_node *x = l->head;
        while (x != NULL) {
                print_key(x->key);
                printf(" ");
                x = x->next;
        }
        printf("\n");
}

int cmp_int(const void *p1, const void *p2)
{
        const int *pa = p1;
```

```c
    const int *pb = p2;
    if (*pa < *pb)
        return -1;
    if (*pa == *pb)
        return 0;
    return 1;
}
void print_key(const void *key)
{
    const int *p = key;
    printf("%d", *p);
}

int main()
{
    list l = list_create();
    for (int i = 0; i < 10; i++) {
        struct list_node *n = malloc(sizeof(struct list_node));
        int *p = malloc(sizeof(int));
        *p = i;
        list_node_ini(n, p);
        list_insert(l, n);
    }
    list_display(l, print_key);
    int k = 0;
    struct list_node *node = list_search(l, &k, cmp_int);
    printf("查找关键字:%d 的结果:%s\n", k,
            node != NULL ? "成功" : "失败");
    printf("删除关键字:%d 的结果:\n", k);
    list_delete(l, node);
    free(node->key);
    free(node);
    list_display(l, print_key);
    list_destroy(l,free);
    return 0;
}
```

10.2.2 带哨兵的环形双向链表

```c
#include <stdio.h>
#include <stdlib.h>
/*带哨兵的环形双向链表*/
typedef struct list_circle_type *list;
```

```
struct list_node {
	void *key;
	struct list_node *prev;
	struct list_node *next;
};
struct list_circle_type {
	struct list_node *nil;  /*哑元结点，用来代替 NULL */
};
void list_node_ini(struct list_node *p, void *key)
{
	p->key = key;
	p->prev = NULL;
	p->next = NULL;
}

list list_create()
{
	list l = malloc(sizeof(struct list_circle_type *));
	l->nil = malloc(sizeof(struct list_node));
	l->nil->prev = l->nil;
	l->nil->next = l->nil;
	return l;
}

void list_destroy(list l, void (*free_key) (void *))
{
	struct list_node *x = l->nil->next;
	while (x != l->nil) {
		struct list_node *del = x;
		x = x->next;
		free_key(del->key);
		free(del);
	}
	free(l->nil);
	free(l);
}

struct list_node *list_search(list l, void *k,
				int (*comp) (const void *, const void *))
{
	struct list_node *x = l->nil->next;
	while (x != l->nil && comp(x->key, k) != 0) {
		x = x->next;
	}
```

```
    return x;
}

void list_insert(list l, struct list_node *x)
{
    x->next = l->nil->next;
    l->nil->next->prev = x;
    l->nil->next = x;
    x->prev = l->nil;
}

void list_delete(list l, struct list_node *x)
{
    if (x == l->nil)
        return;
    x->next->prev = x->prev;
    x->prev->next = x->next;
}

void list_display(list l, void (*print_key) (const void *))
{
    struct list_node *x = l->nil->next;
    while (x != l->nil) {
        print_key(x->key);
        printf(" ");
        x = x->next;
    }
    printf("\n");
}

int cmp_int(const void *p1, const void *p2)
{
    const int *pa = p1;
    const int *pb = p2;
    if (*pa < *pb)
        return -1;
    if (*pa == *pb)
        return 0;
    return 1;
}
void print_key(const void *key)
{
    const int *p = key;
    printf("%d", *p);
```

```
}

int main()
{
	list l = list_create();
	for (int i = 0; i < 10; i++) {
		struct list_node *n = malloc(sizeof(struct list_node));
		int *p = malloc(sizeof(int));
		*p = i;
		list_node_ini(n, p);
		list_insert(l, n);
	}
	list_display(l, print_key);
	int k = 0;
	struct list_node *node = list_search(l, &k, cmp_int);
	printf("查找关键字:%d 的结果:%s\n", k,
	       node != l->nil ? "成功" : "失败");
	printf("删除关键字:%d 的结果:\n", k);
	list_delete(l, node);
	free(node->key);
	free(node);
	list_display(l, print_key);
	list_destroy(l,free);
	return 0;
}
```

10.3 指针和对象的实现

```
#include <stdio.h>
#include <stdlib.h>
typedef struct list_type *list;
enum { NIL = -1 };
struct list_type {
	int num;
	int *key;
	int *next;
	int *prev;
	int free;
	int head;
};
list list_create(int num)
{
	list l = malloc(sizeof(struct list_type));
```

```
    l->num = num;
    l->key = malloc(sizeof(int) * num);
    l->prev = malloc(sizeof(int) * num);
    l->next = malloc(sizeof(int) * num);
    /*自由表是一个单链表，只用到 next 数组 */
    l->free = 0;
    for (int i = 0; i < num - 1; i++) {
        l->next[i] = i + 1;
    }
    l->next[num - 1] = NIL;
    l->head = NIL;
    return l;
}

void list_destroy(list l)
{
    free(l->key);
    free(l->prev);
    free(l->next);
    free(l);
}

int list_allocate_object(list l)
{
    if (l->free == NIL) {
        return NIL;
    } else {
        int x = l->free;
        l->free = l->next[x];
        return x;
    }
}

void list_free_object(list l, int x)
{
    if(x==NIL)
        return;
    l->next[x] = l->free;
    l->free = x;
}

int list_search(list l, int k)
{
    int x = l->head;
```

```
	while (x != NIL && l->key[x] != k) {
		x = l->next[x];
	}
	return x;
}

void list_insert(list l, int x)
{
	l->next[x] = l->head;
	if (l->head != NIL) {
		l->prev[l->head] = x;
	}
	l->head = x;
	l->prev[x] = NIL;
}

void list_delete(list l, int x)
{
	if(x==NIL)
		return;
	if (l->prev[x] != NIL) {
		l->next[l->prev[x]] = l->next[x];
	} else {
		l->head = l->next[x];
	}
	if (l->next[x] != NIL) {
		l->prev[l->next[x]] = l->prev[x];
	}
}

void list_display(list l)
{
	int x = l->head;
	while (x != NIL) {
		printf("%d ", l->key[x]);
		x = l->next[x];
	}
	printf("\n");
}

int main()
{
	list l = list_create(10);
	for (int i = 0; i < 10; i++) {
```

```
        int x=list_allocate_object(l);
        l->key[x]=i;
        list_insert(l, x);
    }
    list_display(l);
    int k = 0;
    int pos = list_search(l, k);
    printf("查找关键字:%d 的结果:%s\n", k,
            pos != NIL ? "成功" : "失败");
    printf("删除关键字:%d 的结果:\n", k);
    list_delete(l,pos );
    list_free_object(l,pos);
    list_display(l);
    list_destroy(l);
    return 0;
}
```

第 11 章 散列表

11.2 散列表

```
#include <stdio.h>
#include <stdlib.h>
/*通过链接法解决碰撞*/
typedef struct hash_chain_type *hash;
typedef struct list_type *list;
struct list_node {
    void *key;
    struct list_node *prev;
    struct list_node *next;
};
struct list_type {
    struct list_node *head;
};
struct hash_chain_type {
    list *list_array;
    int (*get_value) (const void *);
    int num;
};
void list_node_ini(struct list_node *p, void *key)
{
```

```
	p->key = key;
	p->prev = NULL;
	p->next = NULL;
}

list list_create()
{
	list l = malloc(sizeof(struct list_type *));
	l->head = NULL;
	return l;
}

void list_destroy(list l, void (*free_key) (void *))
{
	struct list_node *x = l->head;
	while (x != NULL) {
		struct list_node *del = x;
		x = x->next;
		free_key(del->key);
		free(del);
	}

	free(l);
}

struct list_node *list_search(list l, void *k,
				int (*comp) (const void *, const void *))
{
	struct list_node *x = l->head;
	while (x != NULL && comp(x->key, k) != 0) {
		x = x->next;
	}
	return x;
}

void list_insert(list l, struct list_node *x)
{
	x->next = l->head;
	if (l->head != NULL) {
		l->head->prev = x;
	}
	l->head = x;
	x->prev = NULL;
}
```

```
void list_delete(list l, struct list_node *x)
{
	if (x->prev != NULL) {
		x->prev->next = x->next;
	} else {
		l->head = x->next;
	}
	if (x->next != NULL) {
		x->next->prev = x->prev;
	}
}

hash hash_create(int num, int (*get_value) (const void *))
{
	hash h = malloc(sizeof(struct hash_chain_type));
	h->num = num;
	h->get_value = get_value;
	h->list_array = malloc(sizeof(list) * num);
	for (int i = 0; i < num; i++) {
		h->list_array[i] = list_create();
	}
	return h;
}

void hash_destroy(hash h, void (*free_key) (void *))
{
	for (int i = 0; i < h->num; i++) {
		list_destroy(h->list_array[i], free_key);
	};
	free(h->list_array);
	free(h);
}

int hash_value(hash h, int key)
{
	return key % h->num;
}

void hash_insert(hash h, struct list_node *x)
{
	int key = h->get_value(x);
	list l = h->list_array[hash_value(h, key)];
	list_insert(l, x);
```

```
}

struct list_node *hash_search(hash h, int key,
                               int (*comp) (const void *, const void *))
{
        list l = h->list_array[hash_value(h, key)];
        return list_search(l, &key, comp);
}

void hash_delete(hash h, struct list_node *x)
{
        if (x == NULL)
                return;
        int key = h->get_value(x);
        list l = h->list_array[hash_value(h, key)];
        list_delete(l, x);
}

/*用于 list_node 的 key 成员的比较函数*/
int cmp_int(const void *p1, const void *p2)
{
        const int *pa = p1;
        const int *pb = p2;
        if (*pa < *pb)
                return -1;
        if (*pa == *pb)
                return 0;
        return 1;
}
/*从 list_node 类型指针中取得关键字的整数值*/
int get_value(const void *x)
{
        const struct list_node *p = x;
        const int *ip = p->key;
        return *ip;
}

int main()
{
        hash h = hash_create(10,get_value);
        for (int i = 0; i < 10; i++) {
                struct list_node *x = malloc(sizeof(struct list_node));
                int *p = malloc(sizeof(int));
                *p = i;
```

```
        list_node_ini(x, p);
        printf("%d ", *p);
        hash_insert(h, x);
    }
    printf("\n");
    int k = 0;
    struct list_node *x = hash_search(h, k, cmp_int);
    printf("查找关键字:%d 的结果:%s\n", k,
            x != NULL ? "成功" : "失败");
    if (x != NULL) {
        hash_delete(h, x);
        free(x->key);
        free(x);
        x = hash_search(h, k, cmp_int);
        printf("删除关键字:%d 的结果:%s\n", k,
                x == NULL ? "成功" : "失败");
    }
    hash_destroy(h, free);
    return 0;
}
```

11.4 开放地址法

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct hash_open_addressing_type *hash;
#define NIL      ((void*)-1)
#define DELETED ((void*)-2)
struct hash_open_addressing_type {
    void **array;
    int num;
    int (*hash_fun) (hash, int, int);
    int (*get_value) (const void *);
};
hash hash_create(int num, int (*hash_fun) (hash, int, int),
          int (*get_value) (const void *))
{
    hash h = malloc(sizeof(struct hash_open_addressing_type));
    h->num = num;
    h->array = malloc(sizeof(void *) * num);
    for (int i = 0; i < h->num; i++)
        h->array[i] = NIL;
```

```
    h->hash_fun = hash_fun;
    h->get_value = get_value;
    return h;
}

void hash_destroy(hash h, void (*free_key) (void *))
{
    for (int i = 0; i < h->num; i++) {
        if (h->array[i] != NIL && h->array[i] != DELETED) {
            free_key(h->array[i]);
        }
    }
    free(h->array);
    free(h);
}

int hash_insert(hash h, void *p)
{
    int key = h->get_value(p);
    int i = 0;
    int j = 0;
    do {
        j = h->hash_fun(h, key, i);
        if (h->array[j] == NIL || h->array[j] == DELETED) {
            h->array[j] = p;
            return j;
        } else {
            i++;
        }
    } while (i < h->num);
    return -1;
}

/*返回槽的索引*/
int hash_search(hash h, int key)
{
    int i = 0;
    int j = 0;
    do {
        j = (h->hash_fun) (h, key, i);
        if (h->array[j] != DELETED) {
            int value = h->get_value(h->array[j]);
            if (value == key) {
                return j;
```

```
            }
        }
        ++i;
    } while (h->array[j] != NIL && i < h->num);
    return -1;
}

/*返回槽的索引*/
int hash_delete(hash h, int key)
{
    int i = hash_search(h, key);
    if (i != -1) {
        h->array[i] = DELETED;
    }
    return i;
}

int hash_fun_linear(hash h, int key, int i)
{
    int h1 = key % h->num;
    return (h1 + i) % h->num;
}

int hash_fun_quadratic(hash h, int key, int i)
{
    int h1 = key % h->num;
    return (h1 + i + i * i) % h->num;
}

int hash_fun_double_hash(hash h, int key, int i)
{
    int h1 = key % h->num;
    int h2 = key % h->num + 1;
    return (h1 + i * h2) % h->num;
}

/*从存到 hash 表里的类型指针中取得关键字的整数值*/
int get_value(const void *x)
{
    const int *ip = x;
    return *ip;
}

int main()
```

```
{
	hash h = hash_create(10, hash_fun_double_hash, get_value);
	for (int i = 0; i < 10; i++) {
		int *p = malloc(sizeof(int));
		*p = i;
		printf("%d ", *p);
		hash_insert(h, p);
	}
	printf("\n");
	int k = 0;
	int pos = hash_search(h, k);
	printf("查找关键字:%d 的结果:%s\n", k,
			pos != -1 ? "成功" : "失败");
	int *p = h->array[pos];
	int delete_key = get_value(p);
	hash_delete(h, delete_key);
	free(p);
	pos = hash_search(h, k);
	printf("删除关键字:%d 的结果:%s\n", k,
			pos == -1 ? "成功" : "失败");
	hash_destroy(h, free);
	return 0;
}
```

第 12 章 二叉查找树

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct binary_search_tree_type *tree;
struct tree_node {
	void *key;
	struct tree_node *parent;
	struct tree_node *left;
	struct tree_node *right;
};
struct binary_search_tree_type {
	int (*comp) (const void *, const void *);
	struct tree_node *root;
};
void tree_node_ini(struct tree_node *p, void *key)
{
```

```
    p->key = key;
    p->parent = NULL;
    p->left = NULL;
    p->right = NULL;
}

tree tree_create(int (*comp) (const void *, const void *))
{
    tree t = malloc(sizeof(struct binary_search_tree_type));
    t->comp = comp;
    t->root = NULL;
    return t;
}

void tree_delete_node(tree t, struct tree_node *x, void (*free_key) (void *))
{
    if (x != NULL) {
        tree_delete_node(t, x->left, free_key);
        tree_delete_node(t, x->right, free_key);
        free_key(x->key);
        free(x);
    }
}

void tree_destroy(tree t, void (*free_key) (void *))
{
    tree_delete_node(t, t->root, free_key);
    free(t);
}

void tree_inorder_tree_walk(struct tree_node *x, void (*handle) (const void *))
{
    if (x != NULL) {
        tree_inorder_tree_walk(x->left, handle);
        handle(x->key);
        tree_inorder_tree_walk(x->right, handle);
    }
}

struct tree_node *tree_search(tree t, struct tree_node *x, void *key)
{
    if (x == NULL || t->comp(key, x->key) == 0) {
        return x;
    }
```

```
    if (t->comp(key, x->key) < 0) {
        return tree_search(t, x->left, key);
    } else {
        return tree_search(t, x->right, key);
    }
}

struct tree_node *tree_search_iterative(tree t, struct tree_node *x, void *key)
{
    while (x != NULL && t->comp(key, x->key) != 0) {
        if (t->comp(key, x->key) < 0) {
            x = x->left;
        } else {
            x = x->right;
        }
    }
    return x;
}

struct tree_node *tree_minimum(struct tree_node *x)
{
    while (x != NULL && x->left != NULL) {
        x = x->left;
    }
    return x;
}

struct tree_node *tree_maximum(struct tree_node *x)
{
    while (x != NULL && x->right != NULL) {
        x = x->right;
    }
    return x;
}

struct tree_node *tree_successor(struct tree_node *x)
{
    if (x->right != NULL) {
        return tree_minimum(x->right);
    }
    struct tree_node *y = x->parent;
    while (y != NULL && x == y->right) {
        x = y;
        y = y->parent;
```

```
	}
	return y;
}

struct tree_node *tree_predecessor(struct tree_node *x)
{
	if (x->left != NULL) {
		return tree_maximum(x->left);
	}
	struct tree_node *y = x->parent;
	while (y != NULL && x == y->left) {
		x = y;
		y = y->parent;
	}
	return y;
}

void tree_insert(tree t, struct tree_node *z)
{
	struct tree_node *y = NULL;
	struct tree_node *x = t->root;
	while (x != NULL) {
		y = x;
		if (t->comp(z->key, x->key) < 0) {
			x = x->left;
		} else {
			x = x->right;
		}
	}
	z->parent = y;
	if (y == NULL) {
		t->root = z;
	} else {
		if (t->comp(z->key, y->key) < 0) {
			y->left = z;
		} else {
			y->right = z;
		}
	}
}

void swap(void *a, void *b, size_t elem_size)
{
	if (a == NULL || b == NULL || a == b)
```

```
        return;
    char temp[elem_size];     /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}

struct tree_node *tree_delete(tree t, struct tree_node *z)
{
    struct tree_node *y;
    struct tree_node *x;
    if (z->left == NULL || z->right == NULL) {
        y = z;
    } else {
        y = tree_successor(z);
    }
    if (y->left != NULL) {
        x = y->left;
    } else {
        x = y->right;
    }
    if (x != NULL) {
        x->parent = y->parent;
    }
    if (y->parent == NULL) {
        t->root = x;
    } else {
        if (y == y->parent->left) {
            y->parent->left = x;
        } else {
            y->parent->right = x;
        }
    }
    if (y != z) {
        /*要删除的结点 y 是 z 的后继,交换 z 和 y 结点的内容 */
        swap(&z->key, &y->key, sizeof(void *));
    }
    return y;
}

int cmp_int(const void *p1, const void *p2)
{
    const int *pa = p1;
    const int *pb = p2;
```

```
    if (*pa < *pb)
        return -1;
    if (*pa == *pb)
        return 0;
    return 1;
}
void print_key(const void *key)
{
    const int *p = key;
    printf("%d ", *p);
}

int main()
{
    tree t = tree_create(cmp_int);
    for (int i = 0; i < 10; i++) {
        struct tree_node *node = malloc(sizeof(struct tree_node));
        int *ip = malloc(sizeof(int));
        *ip = i;
        tree_node_ini(node, ip);
        tree_insert(t, node);
    }
    printf("中序遍历结果:\n");
    tree_inorder_tree_walk(t->root, print_key);
    printf("\n");
    struct tree_node *max = tree_maximum(t->root);
    printf("max:%d\n", *(int *)max->key);
    struct tree_node *min = tree_minimum(t->root);
    printf("min:%d\n", *(int *)min->key);
    struct tree_node *success = tree_successor(min);
    printf("%d 的后继:%d\n", *(int *)min->key, *(int *)success->key);
    struct tree_node *predecessor = tree_predecessor(max);
    printf("%d 的前趋:%d\n", *(int *)max->key, *(int *)predecessor->key);
    int k = 0;
    struct tree_node *result = tree_search(t, t->root, &k);
    printf("查找关键字:%d 的结果:%s\n", k,
           result != NULL ? "成功" : "失败");
    struct tree_node *del_node = tree_delete(t, result);
    free(del_node->key);
    free(del_node);
    result = tree_search(t, t->root, &k);
    printf("删除关键字:%d 的结果:%s\n", k,
           result == NULL ? "成功" : "失败");
    printf("中序遍历结果:\n");
```

```
	tree_inorder_tree_walk(t->root, print_key);
	printf("\n");
	tree_destroy(t, free);
	return 0;
}
```

第 13 章 红黑树

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct red_black_tree_type *tree;
enum color_enum {
	color_red,
	color_black
};
struct tree_node {
	void *key;
	enum color_enum color;
	struct tree_node *parent;
	struct tree_node *left;
	struct tree_node *right;
};

struct red_black_tree_type {
	int (*comp) (const void *, const void *);
	struct tree_node *root;
	struct tree_node *nil;
};
void tree_node_ini(struct tree_node *p, void *key)
{
	p->key = key;
	p->parent = NULL;
	p->left = NULL;
	p->right = NULL;
	p->color = color_black;
}

void tree_left_rotate(tree t, struct tree_node *x)
{
	struct tree_node *y = x->right;
	x->right = y->left;
```

```
	if (y->left != t->nil) {
		y->left->parent = x;
	}
	y->parent = x->parent;
	if (x->parent == t->nil) {
		t->root = y;
	} else {
		if (x == x->parent->left) {
			x->parent->left = y;
		} else {
			x->parent->right = y;
		}
	}
	y->left = x;
	x->parent = y;
}

void tree_right_rotate(tree t, struct tree_node *x)
{
	struct tree_node *y = x->left;
	x->left = y->right;
	if (y->right != t->nil) {
		y->right->parent = x;
	}
	y->parent = x->parent;
	if (x->parent == t->nil) {
		t->root = y;
	} else {
		if (x == x->parent->left) {
			x->parent->left = y;
		} else {
			x->parent->right = y;
		}
	}
	y->right = x;
	x->parent = y;
}

tree tree_create(int (*comp) (const void *, const void *))
{
	tree t = malloc(sizeof(struct red_black_tree_type));
	t->comp = comp;
	t->nil = malloc(sizeof(struct tree_node));
	tree_node_ini(t->nil, NULL);
```

```
	t->root = t->nil;
	return t;
}

void tree_destroy_all_node(tree t, struct tree_node *x,
			void (*free_key) (void *))
{
	if (x != t->nil) {
		tree_destroy_all_node(t, x->left, free_key);
		tree_destroy_all_node(t, x->right, free_key);
		free_key(x->key);
		free(x);
	}
}

void tree_destroy(tree t, void (*free_key) (void *))
{
	tree_destroy_all_node(t, t->root, free_key);
	free(t->nil);
	free(t);
}

void tree_inorder_walk(tree t, struct tree_node *x,
			void (*handle) (const void *))
{
	if (x != t->nil) {
		tree_inorder_walk(t, x->left, handle);
		handle(x->key);
		printf(" ");
		tree_inorder_walk(t, x->right, handle);
	}
}

struct tree_node *tree_search(tree t, struct tree_node *x, void *key)
{
	if (x == t->nil || t->comp(key, x->key) == 0) {
		return x;
	}
	if (t->comp(key, x->key) < 0) {
		return tree_search(t, x->left, key);
	} else {
		return tree_search(t, x->right, key);
	}
}
```

```
void tree_count_leaf(tree t, struct tree_node *x,
                     struct tree_node *leaf_array[], int *num)
{
    if (x != t->nil) {
        if (x->left == t->nil && x->right == t->nil) {
            leaf_array[++*num] = x;
        }
        tree_count_leaf(t, x->left, leaf_array, num);
        tree_count_leaf(t, x->right, leaf_array, num);
    }
}

void tree_black_height(tree t, struct tree_node *x,
                       struct tree_node *leaf_node,
                       int *black_height)
{
    if (x->color == color_black) {
        ++*black_height;
    }
    if (x == t->nil) {
        return;
    }
    if (t->comp(leaf_node->key , x->key)<0) {
        tree_black_height(t, x->left, leaf_node, black_height);
    } else {
        tree_black_height(t, x->right, leaf_node, black_height);
    }
}

struct tree_node *tree_minimum(tree t, struct tree_node *x)
{
    while (x != t->nil && x->left != t->nil) {
        x = x->left;
    }
    return x;
}

struct tree_node *tree_successor(tree t, struct tree_node *x)
{
    if (x->right != t->nil) {
        return tree_minimum(t, x->right);
    }
    struct tree_node *y = x->parent;
```

```
	while (y != t->nil && x == y->right) {
		x = y;
		y = y->parent;
	}
	return y;
}

void tree_insert_fixup(tree t, struct tree_node *z)
{
	struct tree_node *y = t->nil;
	while (z->parent->color == color_red) {
		if (z->parent == z->parent->parent->left) {
			y = z->parent->parent->right;
			//情况 1:z 的叔叔 y 是红色的
			if (y->color == color_red) {
				z->parent->color = color_black;
				y->color = color_black;
				z->parent->parent->color = color_red;
				z = z->parent->parent;
			} else {
				//情况 2:z 的叔叔 y 是黑色的,z 是右孩子
				if (z == z->parent->right) {
					z = z->parent;
					tree_left_rotate(t, z);
				}
				//情况 3:z 的叔叔 y 是黑色的,z 是左孩子
				z->parent->color = color_black;
				z->parent->parent->color = color_red;
				tree_right_rotate(t, z->parent->parent);
			}
		} else {
			y = z->parent->parent->left;
			if (y->color == color_red) {
				z->parent->color = color_black;
				y->color = color_black;
				z->parent->parent->color = color_red;
				z = z->parent->parent;
			} else {
				if (z == z->parent->left) {
					z = z->parent;
					tree_right_rotate(t, z);
				}
				z->parent->color = color_black;
				z->parent->parent->color = color_red;
```

```
                tree_left_rotate(t, z->parent->parent);
            }
        }
    }
    t->root->color = color_black;
}

void tree_insert(tree t, struct tree_node *z)
{
    struct tree_node *y = t->nil;
    struct tree_node *x = t->root;
    while (x != t->nil) {
        y = x;
        if (t->comp(z->key, x->key) < 0) {
            x = x->left;
        } else {
            x = x->right;
        }
    }
    z->parent = y;
    if (y == t->nil) {
        t->root = z;
    } else {
        if (t->comp(z->key, y->key) < 0) {
            y->left = z;
        } else {
            y->right = z;
        }
    }
    z->left = t->nil;
    z->right = t->nil;
    z->color = color_red;
    tree_insert_fixup(t, z);
}

void tree_delete_fixup(tree t, struct tree_node *x)
{
    struct tree_node *w = t->nil;
    while (x != t->root && x->color == color_black) {
        if (x == x->parent->left) {
            w = x->parent->right;
            //情况 1:x 的兄弟 w 是红色的
            if (w->color == color_red) {
                w->color = color_black;
```

```
                x->parent->color = color_red;
                tree_left_rotate(t, x->parent);
                w = x->parent->right;
            } else {
                //情况 2:x 的兄弟 w 是黑色的,而且 w 的两个孩子都是黑色的
                if (w->left->color == color_black
                      && w->right->color == color_black) {
                    w->color = color_red;
                    x = x->parent;
                } else {
                    //情况 3:x 的兄弟 w 是黑色的,w 的左孩子是红的，右孩子是
黑的
                    if (w->right->color == color_black) {
                        w->left->color = color_black;
                        w->color = color_red;
                        tree_right_rotate(t, w);
                        w = x->parent->right;
                    }
                    //情况 4:x 的兄弟 w 是黑色的,而且 w 的右孩子是红色的
                    w->color = x->parent->color;
                    x->parent->color = color_black;
                    w->right->color = color_black;
                    tree_left_rotate(t, x->parent);
                    x = t->root;
                }
            }
        } else {
            w = x->parent->left;
            if (w->color == color_red) {
                w->color = color_black;
                x->parent->color = color_red;
                tree_right_rotate(t, x->parent);
                w = x->parent->left;
            } else {
                if (w->right->color == color_black
                      && w->left->color == color_black) {
                    w->color = color_red;
                    x = x->parent;
                } else {
                    if (w->left->color == color_black) {
                        w->right->color = color_black;
                        w->color = color_red;
                        tree_left_rotate(t, w);
                        w = x->parent->left;
```

```
                        }
                        w->color = x->parent->color;
                        x->parent->color = color_black;
                        w->left->color = color_black;
                        tree_right_rotate(t, x->parent);
                        x = t->root;
                    }
                }
            }
        }
        x->color = color_black;
}

void swap(void *a, void *b, size_t elem_size)
{
        if (a == NULL || b == NULL || a == b)
            return;
        char temp[elem_size];       /*变长数组 */
        memcpy(temp, a, elem_size);
        memcpy(a, b, elem_size);
        memcpy(b, temp, elem_size);
}

struct tree_node *tree_delete(tree t, struct tree_node *z)
{
        struct tree_node *y;
        struct tree_node *x;
        if (z->left == t->nil || z->right == t->nil) {
            y = z;
        } else {
            y = tree_successor(t, z);
        }
        if (y->left != t->nil) {
            x = y->left;
        } else {
            x = y->right;
        }
        x->parent = y->parent;
        if (y->parent == t->nil) {
            t->root = x;
        } else {
            if (y == y->parent->left) {
                y->parent->left = x;
            } else {
```

```
            y->parent->right = x;
        }
    }
    if (y != z) {
        /*要删除的结点 y 是 z 的后继,交换 z 和 y 结点的内容 */
        swap(&z->key, &y->key, sizeof(void *));
    }
    if (y->color == color_black) {
        tree_delete_fixup(t, x);
    }
    return y;
}

int cmp_int(const void *p1, const void *p2)
{
    const int *pa = p1;
    const int *pb = p2;
    if (*pa < *pb)
        return -1;
    if (*pa == *pb)
        return 0;
    return 1;
}
void print_key(const void *key)
{
    const int *p = key;
    printf("%-2d", *p);
}

int main()
{
    tree t = tree_create(cmp_int);
    for (int i = 0; i < 10; i++) {
        struct tree_node *node = malloc(sizeof(struct tree_node));
        int *ip = malloc(sizeof(int));
        *ip = i;
        tree_node_ini(node, ip);
        tree_insert(t, node);
    }
    printf("中序遍历结果:\n");
    tree_inorder_walk(t, t->root, print_key);
    printf("\n");
    int k = 0;
    struct tree_node *result = tree_search(t, t->root, &k);
```

```
    printf("查找关键字:%d 的结果:%s\n", k,
            result != t->nil ? "成功" : "失败");
    struct tree_node *del_node = tree_delete(t, result);
    free(del_node->key);
    free(del_node);
    result = tree_search(t, t->root, &k);
    printf("删除关键字:%d 的结果:%s\n", k,
            result == t->nil ? "成功" : "失败");
    printf("查看黑高度:\n");
    struct tree_node *left_array[100];
    int num = -1;
    /*统计叶子结点 */
    tree_count_leaf(t, t->root, left_array, &num);
    for (int i = 0; i < num; i++) {
        /*黑高度不包括自身,所以初始值为-1 */
        int black_height = -1;
        tree_black_height(t, t->root, left_array[i], &black_height);
        /*测试黑高度，从根到叶的路径上，黑结点数是一样的 */
        printf("根到叶子:%d 的黑高度:%d\n",
                *(int *)left_array[i]->key, black_height);
    }
    printf("中序遍历结果:\n");
    tree_inorder_walk(t, t->root, print_key);
    printf("\n");
    /*遍历树，释放结点的 key */
    tree_destroy(t, free);
    return 0;
}
```

第 14 章 数据结构的扩张

14.1 动态顺序统计

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct red_black_tree_type *tree;
enum color_enum {
    color_red,
    color_black
};
```

```
struct tree_node {
	void *key;
	enum color_enum color;
	struct tree_node *parent;
	struct tree_node *left;
	struct tree_node *right;
	int size;
};

struct red_black_tree_type {
	int (*comp)(const void*,const void*);
	struct tree_node *root;
	struct tree_node *nil;
};
void tree_node_ini(struct tree_node *p, void *key)
{
	p->key = key;
	p->parent = NULL;
	p->left = NULL;
	p->right = NULL;
	p->color = color_black;
	p->size = 0;
}

void tree_left_rotate(tree t, struct tree_node *x)
{
	struct tree_node *y = x->right;
	x->right = y->left;
	y->left->parent = x;
	y->parent = x->parent;
	if (x->parent == t->nil) {
		t->root = y;
	} else {
		if (x == x->parent->left) {
			x->parent->left = y;
		} else {
			x->parent->right = y;
		}
	}
	y->left = x;
	x->parent = y;
	y->size = x->size;
	x->size = x->left->size + x->right->size + 1;
}
```

```
void tree_right_rotate(tree t, struct tree_node *x)
{
	struct tree_node *y = x->left;
	x->left = y->right;
	y->right->parent = x;
	y->parent = x->parent;
	if (x->parent == t->nil) {
		t->root = y;
	} else {
		if (x == x->parent->left) {
			x->parent->left = y;
		} else {
			x->parent->right = y;
		}
	}
	y->right = x;
	x->parent = y;
	y->size = x->size;
	x->size = x->left->size + x->right->size + 1;
}

tree tree_create(int (*comp)(const void*,const void*))
{
	tree t = malloc(sizeof(struct red_black_tree_type));
	t->nil = malloc(sizeof(struct tree_node));
	t->comp=comp;
	tree_node_ini(t->nil, NULL);
	t->root = t->nil;
	return t;
}

void tree_destroy_all_node(tree t, struct tree_node *x, void (*free_key) (void *))
{
	if (x != t->nil) {
		tree_destroy_all_node(t, x->left, free_key);
		tree_destroy_all_node(t, x->right, free_key);
		free_key(x->key);
		free(x);
	}
}

void tree_destroy(tree t, void (*free_key) (void *))
{
```

```
	tree_destroy_all_node(t, t->root, free_key);
	free(t->nil);
	free(t);
}

void tree_inorder_tree_walk(tree t, struct tree_node *x,
			void (*handle) (const void *))
{
	if (x != t->nil) {
		tree_inorder_tree_walk(t, x->left, handle);
		handle(x->key);
		tree_inorder_tree_walk(t, x->right, handle);
	}
}

struct tree_node *tree_search(tree t, struct tree_node *x, void *key,
			int (*comp) (const void *, const void *))
{
	if (x == t->nil || comp(key, x->key) == 0) {
		return x;
	}
	if (comp(key, x->key) < 0) {
		return tree_search(t, x->left, key, comp);
	} else {
		return tree_search(t, x->right, key, comp);
	}
}

struct tree_node *tree_minimum(tree t, struct tree_node *x)
{
	while (x != t->nil && x->left != t->nil) {
		x = x->left;
	}
	return x;
}

struct tree_node *tree_successor(tree t, struct tree_node *x)
{
	if (x->right != t->nil) {
		return tree_minimum(t, x->right);
	}
	struct tree_node *y = x->parent;
	while (y != t->nil && x == y->right) {
		x = y;
```

```
        y = y->parent;
    }
    return y;
}

void tree_insert_fixup(tree t, struct tree_node *z)
{
    struct tree_node *y = t->nil;
    while (z->parent->color == color_red) {
        if (z->parent == z->parent->parent->left) {
            y = z->parent->parent->right;
            //情况 1:z 的叔叔 y 是红色的
            if (y->color == color_red) {
                z->parent->color = color_black;
                y->color = color_black;
                z->parent->parent->color = color_red;
                z = z->parent->parent;
            } else {
                //情况 2:z 的叔叔 y 是黑色的,z 是右孩子
                if (z == z->parent->right) {
                    z = z->parent;
                    tree_left_rotate(t, z);
                }
                //情况 3:z 的叔叔 y 是黑色的,z 是左孩子
                z->parent->color = color_black;
                z->parent->parent->color = color_red;
                tree_right_rotate(t, z->parent->parent);
            }
        } else {
            y = z->parent->parent->left;
            if (y->color == color_red) {
                z->parent->color = color_black;
                y->color = color_black;
                z->parent->parent->color = color_red;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    tree_right_rotate(t, z);
                }
                z->parent->color = color_black;
                z->parent->parent->color = color_red;
                tree_left_rotate(t, z->parent->parent);
            }
```

```
        }
    }
    t->root->color = color_black;
}

void tree_insert(tree t, struct tree_node *z)
{
    struct tree_node *y = t->nil;
    struct tree_node *x = t->root;
    while (x != t->nil) {
        y = x;
        ++y->size;
        if (t->comp(z->key, x->key) < 0) {
            x = x->left;
        } else {
            x = x->right;
        }
    }
    z->parent = y;
    if (y == t->nil) {
        t->root = z;
    } else {
        if (t->comp(z->key, y->key) < 0) {
            y->left = z;
        } else {
            y->right = z;
        }
    }
    z->left = t->nil;
    z->right = t->nil;
    z->color = color_red;
    z->size=1;
    tree_insert_fixup(t, z);
}

void tree_delete_fixup(tree t, struct tree_node *x)
{
    struct tree_node *w = t->nil;
    while (x != t->root && x->color == color_black) {
        if (x == x->parent->left) {
            w = x->parent->right;
            //情况 1:x 的兄弟 w 是红色的
            if (w->color == color_red) {
                w->color = color_black;
```

```
                x->parent->color = color_red;
                tree_left_rotate(t, x->parent);
                w = x->parent->right;
            } else {
                //情况 2:x 的兄弟 w 是黑色的,而且 w 的两个孩子都是黑色的
                if (w->left->color == color_black
                     && w->right->color == color_black) {
                    w->color = color_red;
                    x = x->parent;
                } else {
                    //情况 3:x 的兄弟 w 是黑色的,w 的左孩子是红的，右孩子是
黑的
                    if (w->right->color == color_black) {
                        w->left->color = color_black;
                        w->color = color_red;
                        tree_right_rotate(t, w);
                        w = x->parent->right;
                    }
                    //情况 4:x 的兄弟 w 是黑色的,而且 w 的右孩子是红色的
                    w->color = x->parent->color;
                    x->parent->color = color_black;
                    w->right->color = color_black;
                    tree_left_rotate(t, x->parent);
                    x = t->root;
                }
            }
        } else {
            w = x->parent->left;
            if (w->color == color_red) {
                w->color = color_black;
                x->parent->color = color_red;
                tree_right_rotate(t, x->parent);
                w = x->parent->left;
            } else {
                if (w->right->color == color_black
                     && w->left->color == color_black) {
                    w->color = color_red;
                    x = x->parent;
                } else {
                    if (w->left->color == color_black) {
                        w->right->color = color_black;
                        w->color = color_red;
                        tree_left_rotate(t, w);
                        w = x->parent->left;
```

```
                        }
                        w->color = x->parent->color;
                        x->parent->color = color_black;
                        w->left->color = color_black;
                        tree_right_rotate(t, x->parent);
                        x = t->root;
                    }
                }
            }
        }
        x->color = color_black;
}

void swap(void *a, void *b, size_t elem_size)
{
        if (a == NULL || b == NULL || a == b)
            return;
        char temp[elem_size];       /*变长数组 */
        memcpy(temp, a, elem_size);
        memcpy(a, b, elem_size);
        memcpy(b, temp, elem_size);
}

struct tree_node *tree_delete(tree t, struct tree_node *z)
{
        struct tree_node *y;
        struct tree_node *x;
        if (z->left == t->nil || z->right == t->nil) {
            y = z;
        } else {
            y = tree_successor(t, z);
        }
        if (y->left != t->nil) {
            x = y->left;
        } else {
            x = y->right;
        }
        x->parent = y->parent;
        if (y->parent == t->nil) {
            t->root = x;
        } else {
            if (y == y->parent->left) {
                y->parent->left = x;
            } else {
```

```
            y->parent->right = x;
        }
    }
    if (y != z) {
        /*要删除的结点 y 是 z 的后继,交换 z 和 y 结点的内容 */
        swap(&z->key, &y->key, sizeof(void *));
    }
    struct tree_node *p = y;
    do {
        p = p->parent;
        --p->size;
    } while (p != t->root);    /*更新 size */
    if (y->color == color_black) {
        tree_delete_fixup(t, x);
    }
    return y;
}

/*检索具有给定顺序的元素*/
struct tree_node *tree_select(struct tree_node *x, int i)
{
    int r = x->left->size + 1;
    if (i == r) {
        return x;
    } else {
        if (i < r) {
            return tree_select(x->left, i);
        } else {
            return tree_select(x->right, i - r);
        }
    }
}

/*确定一个元素的秩*/
int tree_rank(tree t, struct tree_node *x)
{
    int r = x->left->size + 1;
    struct tree_node *y = x;
    while (y != t->root) {
        if (y == y->parent->right) {
            r = r + y->parent->left->size + 1;
        }
        y = y->parent;
    }
```

```
    return r;
}

void print_key(const void *key)
{
    const int *p = key;
    printf("%d ", *p);
}

int cmp_int(const void *p1, const void *p2)
{
    const int *pa = p1;
    const int *pb = p2;
    if (*pa < *pb)
        return -1;
    if (*pa == *pb)
        return 0;
    return 1;
}
int main()
{
    tree t = tree_create(cmp_int);
    for (int i = 0; i < 10; i++) {
        struct tree_node *node = malloc(sizeof(struct tree_node));
        int *ip = malloc(sizeof(int));
        *ip = i;
        tree_node_ini(node, ip);
        tree_insert(t, node);
    }
    printf("中序遍历结果:\n");
    tree_inorder_tree_walk(t, t->root, print_key);
    printf("\n");
    int rank = tree_rank(t, t->root);
    printf("根结点:%d,排名:%d,Size:%d\n", *(int *)t->root->key, rank,
            t->root->size);
    rank = 6;
    struct tree_node *node = tree_select(t->root, rank);
    printf("第%d 个元素是:%d\n", rank, *(int *)node->key);
    printf("删掉第%d 个元素的结果是:\n", rank);
    struct tree_node *del_node = tree_delete(t, node);
    free(del_node->key);
    free(del_node);
    tree_inorder_tree_walk(t, t->root, print_key);
    printf("\n");
```

```
	rank = tree_rank(t, t->root);
	printf("删除后，根结点:%d,排名:%d,Size:%d\n",
			*(int *)t->root->key, rank, t->root->size);
	/*遍历树，释放结点的 key */
	tree_destroy(t, free);
	return 0;
}
```

14.3 区间树

```
#include <stdio.h>
#include <time.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
#define  MAX(a,b) (((a)>(b))?(a):(b))
typedef struct red_black_tree_type *tree;
enum color_enum {
	color_red,
	color_black
};
struct interval {
	int low;
	int hight;
};

struct tree_node {
	struct interval it;
	enum color_enum color;
	struct tree_node *parent;
	struct tree_node *left;
	struct tree_node *right;
	int interval_max;
};

struct red_black_tree_type {
	struct tree_node *root;
	struct tree_node *nil;
};
void interval_ini(struct interval *it, int low, int hight)
{
	it->low = low;
	it->hight = hight;
```

```
}

bool interval_is_overlap(struct interval *it_a, struct interval *it_b)
{
	if (it_a->low <= it_b->hight && it_b->low <= it_a->hight) {
		return true;
	} else {
		return false;
	}
}

void tree_node_ini(struct tree_node *p, struct interval *it)
{
	p->it = *it;
	p->interval_max = it->hight;
	p->parent = NULL;
	p->left = NULL;
	p->right = NULL;
	p->color = color_black;
}

void tree_left_rotate(tree t, struct tree_node *x)
{
	struct tree_node *y = x->right;
	x->right = y->left;
	y->left->parent = x;
	y->parent = x->parent;
	if (x->parent == t->nil) {
		t->root = y;
	} else {
		if (x == x->parent->left) {
			x->parent->left = y;
		} else {
			x->parent->right = y;
		}
	}
	y->left = x;
	x->parent = y;
	y->interval_max = x->interval_max;
	int max = MAX(x->left->interval_max, x->right->interval_max);
	x->interval_max = MAX(x->it.hight, max);
}

void tree_right_rotate(tree t, struct tree_node *x)
```

```
{
	struct tree_node *y = x->left;
	x->left = y->right;
	y->right->parent = x;
	y->parent = x->parent;
	if (x->parent == t->nil) {
		t->root = y;
	} else {
		if (x == x->parent->left) {
			x->parent->left = y;
		} else {
			x->parent->right = y;
		}
	}
	y->right = x;
	x->parent = y;
	y->interval_max = x->interval_max;
	int max = MAX(x->left->interval_max, x->right->interval_max);
	x->interval_max = MAX(x->it.hight, max);
}

tree tree_create()
{
	tree t = malloc(sizeof(struct red_black_tree_type));
	t->nil = malloc(sizeof(struct tree_node));
	tree_node_ini(t->nil, &(struct interval) {
			0, 0});
	t->root = t->nil;
	return t;
}

void tree_delete_node(tree t, struct tree_node *x)
{
	if (x != t->nil) {
		tree_delete_node(t, x->left);
		tree_delete_node(t, x->right);
		free(x);
	}
}

void tree_destroy(tree t)
{
	tree_delete_node(t, t->root);
	free(t->nil);
```

```
    free(t);
}

void tree_inorder_tree_walk(tree t, struct tree_node *x,
                            void (*handle) (struct interval *))
{
    if (x != t->nil) {
        tree_inorder_tree_walk(t, x->left, handle);
        handle(&x->it);
        tree_inorder_tree_walk(t, x->right, handle);
    }
}

struct tree_node *tree_interval_search(tree t, struct interval *it)
{
    struct tree_node *x = t->root;
    while (x != t->nil && !interval_is_overlap(&x->it, it)) {
        if (x->left != t->nil && x->left->interval_max >= it->low) {
            x = x->left;
        } else {
            x = x->right;
        }
    }
    return x;
}

struct tree_node *tree_minimum(tree t, struct tree_node *x)
{
    while (x != t->nil && x->left != t->nil) {
        x = x->left;
    }
    return x;
}

struct tree_node *tree_successor(tree t, struct tree_node *x)
{
    if (x->right != t->nil) {
        return tree_minimum(t, x->right);
    }
    struct tree_node *y = x->parent;
    while (y != t->nil && x == y->right) {
        x = y;
        y = y->parent;
    }
```

```
    return y;
}

void tree_insert_fixup(tree t, struct tree_node *z)
{
    struct tree_node *y = t->nil;
    while (z->parent->color == color_red) {
        if (z->parent == z->parent->parent->left) {
            y = z->parent->parent->right;
            //情况 1:z 的叔叔 y 是红色的
            if (y->color == color_red) {
                z->parent->color = color_black;
                y->color = color_black;
                z->parent->parent->color = color_red;
                z = z->parent->parent;
            } else {
                //情况 2:z 的叔叔 y 是黑色的,z 是右孩子
                if (z == z->parent->right) {
                    z = z->parent;
                    tree_left_rotate(t, z);
                }
                //情况 3:z 的叔叔 y 是黑色的,z 是左孩子
                z->parent->color = color_black;
                z->parent->parent->color = color_red;
                tree_right_rotate(t, z->parent->parent);
            }
        } else {
            y = z->parent->parent->left;
            if (y->color == color_red) {
                z->parent->color = color_black;
                y->color = color_black;
                z->parent->parent->color = color_red;
                z = z->parent->parent;
            } else {
                if (z == z->parent->left) {
                    z = z->parent;
                    tree_right_rotate(t, z);
                }
                z->parent->color = color_black;
                z->parent->parent->color = color_red;
                tree_left_rotate(t, z->parent->parent);
            }
        }
    }
```

```
	t->root->color = color_black;
}

void tree_insert(tree t, struct tree_node *z)
{
	struct tree_node *y = t->nil;
	struct tree_node *x = t->root;
	while (x != t->nil) {
		y = x;
		y->interval_max = MAX(y->interval_max, z->interval_max);
		if (z->it.low < x->it.low) {
			x = x->left;
		} else {
			x = x->right;
		}
	}
	z->parent = y;
	if (y == t->nil) {
		t->root = z;
	} else {
		if (z->it.low < y->it.low) {
			y->left = z;
		} else {
			y->right = z;
		}
	}
	z->left = t->nil;
	z->right = t->nil;
	z->color = color_red;
	tree_insert_fixup(t, z);
}

void tree_delete_fixup(tree t, struct tree_node *x)
{
	struct tree_node *w = t->nil;
	while (x != t->root && x->color == color_black) {
		if (x == x->parent->left) {
			w = x->parent->right;
			//情况 1:x 的兄弟 w 是红色的
			if (w->color == color_red) {
				w->color = color_black;
				x->parent->color = color_red;
				tree_left_rotate(t, x->parent);
				w = x->parent->right;
```

```
            } else {
                //情况 2:x 的兄弟 w 是黑色的,而且 w 的两个孩子都是黑色的
                if (w->left->color == color_black
                     && w->right->color == color_black) {
                    w->color = color_red;
                    x = x->parent;
                } else {
                    //情况 3:x 的兄弟 w 是黑色的,w 的左孩子是红的，右孩子是
黑的
                    if (w->right->color == color_black) {
                        w->left->color = color_black;
                        w->color = color_red;
                        tree_right_rotate(t, w);
                        w = x->parent->right;
                    }
                    //情况 4:x 的兄弟 w 是黑色的,而且 w 的右孩子是红色的
                    w->color = x->parent->color;
                    x->parent->color = color_black;
                    w->right->color = color_black;
                    tree_left_rotate(t, x->parent);
                    x = t->root;
                }
            }
        } else {
            w = x->parent->left;
            if (w->color == color_red) {
                w->color = color_black;
                x->parent->color = color_red;
                tree_right_rotate(t, x->parent);
                w = x->parent->left;
            } else {
                if (w->right->color == color_black
                     && w->left->color == color_black) {
                    w->color = color_red;
                    x = x->parent;
                } else {
                    if (w->left->color == color_black) {
                        w->right->color = color_black;
                        w->color = color_red;
                        tree_left_rotate(t, w);
                        w = x->parent->left;
                    }
                    w->color = x->parent->color;
                    x->parent->color = color_black;
```

```
                    w->left->color = color_black;
                    tree_right_rotate(t, x->parent);
                    x = t->root;
                }
            }
        }
    }
    x->color = color_black;
}

void swap(void *a, void *b, size_t elem_size)
{
    if (a == NULL || b == NULL || a == b)
        return;
    char temp[elem_size];      /*变长数组  */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}

struct tree_node *tree_delete(tree t, struct tree_node *z)
{
    struct tree_node *y;
    struct tree_node *x;
    if (z->left == t->nil || z->right == t->nil) {
        y = z;
    } else {
        y = tree_successor(t, z);
    }
    if (y->left != t->nil) {
        x = y->left;
    } else {
        x = y->right;
    }
    x->parent = y->parent;
    if (y->parent == t->nil) {
        t->root = x;
    } else {
        if (y == y->parent->left) {
            y->parent->left = x;
        } else {
            y->parent->right = x;
        }
    }
```

```
    if (y != z) {
        /*要删除的结点 y 是 z 的后继,交换 z 和 y 结点的内容 */
        swap(&z->it, &y->it, sizeof(struct interval));
    }
    struct tree_node *p = y;
    do {
        p = p->parent;
        int max = MAX(p->left->interval_max, p->right->interval_max);
        p->interval_max = MAX(p->it.hight, max);
    } while (p != t->root);     /*更新 interval_max */
    if (y->color == color_black) {
        tree_delete_fixup(t, x);
    }
    return y;
}

void print_interval(struct interval *it)
{
    printf("(%d,%d)", it->low, it->hight);
}

int main()
{
    srand((unsigned)time(NULL));
    tree t = tree_create();
    for (int i = 0; i < 10; i++) {
        struct tree_node *node = malloc(sizeof(struct tree_node));
        struct interval it;
        interval_ini(&it, i, i + rand() % 10);
        tree_node_ini(node, &it);
        tree_insert(t, node);
    }
    printf("中序遍历结果:\n");
    tree_inorder_tree_walk(t, t->root, print_interval);
    printf("\n");
    printf("根结点区间:(%d,%d),区间最大值:%d\n", t->root->it.low,
            t->root->it.hight, t->root->interval_max);
    struct interval it = { 10, 15 };
    printf("查找与区间:(%d,%d)重叠的区间:\n", it.low, it.hight);
    struct tree_node *result = tree_interval_search(t, &it);
    printf("查找的结果:%s\n", result != t->nil ? "成功" : "失败");
    if (result != t->nil) {
        printf("区间是:(%d,%d)\n", result->it.low, result->it.hight);
        printf("删除区间:(%d,%d)\n", result->it.low,
```

```
                    result->it.hight);
        struct tree_node *del_node = tree_delete(t, result);
        free(del_node);

        printf("中序遍历结果:\n");
        tree_inorder_tree_walk(t, t->root, print_interval);
        printf("\n");
        printf("删掉后,根结点区间:(%d,%d),区间最大值:%d\n",
                    t->root->it.low, t->root->it.hight,
                    t->root->interval_max);
    }
    /*遍历树，释放结点的 key */
    tree_destroy(t);
    return 0;
}
```

第 15 章 动态规划

15.1 装配线调度

```
#include <stdio.h>
#include <stdlib.h>
enum { NUM = 6 };
void fastest_way(int n,int a[][n], int t[][n - 1],
             int e[], int x[], int f[][n], int l[][n],
             int *fastest_time, int *last_line)
{
    f[0][0] = e[0] + a[0][0];
    f[1][0] = e[1] + a[1][0];
    for (int j = 1; j < n; j++) {
        if (f[0][j - 1] <= f[1][j - 1] + t[1][j - 1]) {
            f[0][j] = f[0][j - 1] + a[0][j];
            l[0][j] = 0;
        } else {
            f[0][j] = f[1][j - 1] + t[1][j - 1] + a[0][j];
            l[0][j] = 1;
        }
        if (f[1][j - 1] <= f[0][j - 1] + t[0][j - 1]) {
            f[1][j] = f[1][j - 1] + a[1][j];
            l[1][j] = 1;
        } else {
```

```
                f[1][j] = f[0][j - 1] + t[0][j - 1] + a[1][j];
                l[1][j] = 0;
            }
        }
        if (f[0][NUM - 1] + x[0] <= f[1][n - 1] + x[1]) {
            *fastest_time = f[0][n - 1] + x[0];
            *last_line = 0;
        } else {
            *fastest_time = f[1][n - 1] + x[1];
            *last_line = 1;
        }
}

void print_stations(int n,int line[][n], int last_line)
{
        int i = last_line;
        printf("line %d, station %d\n", i + 1, n);
        for (int j = n - 1; j > 0; j--) {
            i = line[i][j];
            printf("line %d, station %d\n", i + 1, j);
        }
}

int main()
{
        int n=NUM;
        int f[2][NUM];
        int l[2][NUM];
        int a[2][NUM] = { {7, 9, 3, 4, 8, 4}, {8, 5, 6, 4, 5, 7} };
        int t[2][NUM - 1] = { {2, 3, 1, 3, 4}, {2, 1, 2, 2, 1} };
        int e[2] = { 2, 4 };
        int x[2] = { 3, 2 };
        int fastest_time;
        int last_line;
        fastest_way(n,a, t, e, x, f, l, &fastest_time, &last_line);
        printf("%d %d\n", fastest_time, last_line + 1);
        printf("输出 F 数组:\n");
        for (int i = 0; i < n; ++i) {
            printf("%2d ", f[0][i]);
        }
        printf("\n");
        for (int i = 0; i < n; ++i) {
            printf("%2d ", f[1][i]);
        }
```

```c
	printf("\n");
	printf("输出 L 数组:\n");
	for (int i = 1; i < n; ++i) {
		printf("%2d ", l[0][i] + 1);
	}
	printf("\n");
	for (int i = 1; i < n; ++i) {
		printf("%2d ", l[1][i] + 1);
	}
	printf("\n");
	print_stations(n,l, last_line);
	return 0;
}
```

15.2 矩阵链相乘

15.2.1 矩阵相乘

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct matrix_type *matrix;
struct matrix_type {
	int row;
	int col;
	int **data;
};
matrix matrix_create(int row, int col)
{
	if (row == 0)
		return NULL;
	matrix m = malloc(sizeof(struct matrix_type));
	m->row = row;
	m->col = col;
	m->data = malloc(sizeof(int *) * row);
	for (int i = 0; i < row; i++) {
		m->data[i] = malloc(sizeof(int) * col);
		for (int j = 0; j < col; j++) {
			m->data[i][j] = 0;
		}
	}
	return m;
}
```

```
void matrix_destroy(matrix m)
{
	for (int i = 0; i < m->row; i++)
		free(m->data[i]);
	free(m->data);
	free(m);
}

void matrix_display(matrix m)
{
	for (int i = 0; i < m->row; ++i) {
		for (int j = 0; j < m->col; ++j) {
			printf("%2d ", m->data[i][j]);
		}
		printf("\n");
	}
}

void matrix_multiply(matrix A, matrix B, matrix C)
{
	if (A->col != B->row) {
		return;
	}
	for (int i = 0; i < A->row; ++i) {
		for (int j = 0; j < B->col; ++j) {
			C->data[i][j] = 0;
			for (int k = 0; k < A->col; ++k) {
				C->data[i][j] += A->data[i][k] * B->data[k][j];
			}
		}
	}
}

void matrix_copy(matrix mdst, matrix msrc)
{
	if (mdst->row != msrc->row || mdst->col != msrc->col) {
		matrix_destroy(mdst);
		mdst->row = msrc->row;
		mdst->col = msrc->col;
		mdst->data = malloc(sizeof(int *) * mdst->row);
		for (int i = 0; i < mdst->row; i++) {
			mdst->data[i] = malloc(sizeof(int) * mdst->col);
		}
```

```
	}
	for (int i = 0; i < mdst->row; i++) {
		for (int j = 0; j < mdst->col; j++) {
			mdst->data[i][j] = msrc->data[i][j];
		}
	}
}

int main()
{
	matrix A = matrix_create(2, 4);
	matrix B = matrix_create(4, 3);
	for (int i = 0; i < A->row; ++i) {
		for (int j = 0; j < A->col; ++j) {
			A->data[i][j] = 1;	/*全部是 1，为了测试方便随便设置的值 */
		}
	}
	printf("输出 A 矩阵的值:\n");
	matrix_display(A);
	for (int i = 0; i < B->row; ++i) {
		for (int j = 0; j < B->col; ++j) {
			B->data[i][j] = 2;/*全部是 2，为了测试方便随便设置的值 */
		}
	}
	printf("输出 B 矩阵的值:\n");
	matrix_display(B);
	matrix C = matrix_create(A->row, B->col);
	matrix_multiply(A, B, C);
	printf("输出 C 矩阵的值:\n");
	matrix_display(C);
	matrix D = matrix_create(C->row, C->col);
	matrix_copy(D, C);
	printf("输出 D 矩阵的值:\n");
	matrix_display(D);
	matrix_destroy(A);
	matrix_destroy(B);
	matrix_destroy(C);
	matrix_destroy(D);
	return 0;
}
```

15.2.2 求矩阵链的最优加全部括号

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
enum { NUM = 7 };
void matrix_chain_order(int n,int p[], int m[][n], int s[][n])
{
	for (int i = 1; i < n; ++i) {
		m[i][i] = 0;
	}
	for (int l = 2; l < n; ++l) {
		for (int i = 1; i < n - l + 1; ++i) {
			int j = i + l - 1;
			m[i][j] = INT_MAX;
			for (int k = i; k <= j - 1; ++k) {
				int q =
				    m[i][k] + m[k + 1][j] + p[i -
				                        1] * p[k] * p[j];
				if (q < m[i][j]) {
					m[i][j] = q;
					s[i][j] = k;
				}
			}
		}
	}
}

void print_optimal_matrix(int n,int s[][n], int i, int j)
{
	if (i == j) {
		printf("A%d", i);
	} else {
		printf("(");
		print_optimal_matrix(n,s, i, s[i][j]);
		print_optimal_matrix(n,s, s[i][j] + 1, j);
		printf(")");
	}
}

int main()
{
	int n=NUM;
```

```
    int p[NUM] = { 30, 35, 15, 5, 10, 20, 25 };
    int m[NUM][NUM] = { {0} };
    int s[NUM][NUM] = { {0} };
    matrix_chain_order(n,p, m, s);
    printf("最优加全部括号\n");
    print_optimal_matrix(n,s, 1, 6);
    printf("\n");
    printf("输出 m 数组的值\n");
    for (int i = 1; i < n; ++i) {
        for (int j = 1; j < n; ++j) {
            printf("%5d ", m[i][j]);
        }
        printf("\n");
    }
    printf("输出 s 数组的值\n");
    for (int i = 1; i < n; ++i) {
        for (int j = 1; j < n; ++j) {
            printf("%5d ", s[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

15.3 动态规划基础

15.3.1 递归求矩阵链的最优加全部括号

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
enum { NUM = 7 };
int recursive_matrix_chain(int n,int p[], int i, int j, int m[][n], int s[][n])
{
    if (i == j) {
        return 0;
    }
    m[i][j] = INT_MAX;
    for (int k = i; k <= j - 1; k++) {
        int q = recursive_matrix_chain(n,p, i, k, m, s)
            + recursive_matrix_chain(n,p, k + 1, j, m, s)
            + p[i - 1] * p[k] * p[j];
```

```
            if (q < m[i][j]) {
                m[i][j] = q;
                s[i][j] = k;
            }
        }
        return m[i][j];
}

void print_optimal_matrix(int n,int s[][n], int i, int j)
{
        if (i == j) {
            printf("A%d", i);
        } else {
            printf("(");
            print_optimal_matrix(n,s, i, s[i][j]);
            print_optimal_matrix(n,s, s[i][j] + 1, j);
            printf(")");
        }
}

int main()
{
        int n=NUM;
        int p[NUM] = { 30, 35, 15, 5, 10, 20, 25 };
        int m[NUM][NUM] = { {0} };
        int s[NUM][NUM] = { {0} };
        recursive_matrix_chain(n,p, 1, 6, m, s);
        printf("最优加全部括号\n");
        print_optimal_matrix(n,s, 1, 6);
        printf("\n");
        printf("输出 m 数组的值\n");
        for (int i = 1; i < n; ++i) {
            for (int j = 1; j < n; ++j) {
                printf("%5d ", m[i][j]);
            }
            printf("\n");
        }
        printf("输出 s 数组的值\n");
        for (int i = 1; i < n; ++i) {
            for (int j = 1; j < n; ++j) {
                printf("%5d ", s[i][j]);
            }
            printf("\n");
        }
```

```
    return 0;
}
```

15.3.2 加了备忘的递归求矩阵链的最优加全部括号

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
enum { NUM = 7 };
int lookup_chain(int n, int p[], int i, int j, int m[][n], int s[][n])
{
    if (m[i][j] < INT_MAX) {
        return m[i][j];
    }
    if (i == j) {
        m[i][j] = 0;
    } else {
        for (int k = i; k <= j - 1; ++k) {
            int q = lookup_chain(n, p, i, k, m, s)
                + lookup_chain(n, p, k + 1, j, m, s)
                + p[i - 1] * p[k] * p[j];
            if (q < m[i][j]) {
                m[i][j] = q;
                s[i][j] = k;
            }
        }
    }
    return m[i][j];
}

int memoized_matrix_chain(int n, int p[], int m[][n], int s[][n])
{
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            m[i][j] = INT_MAX;
        }
    }
    return lookup_chain(n, p, 1, n - 1, m, s);
}

void print_optimal_matrix(int n, int s[][n], int i, int j)
{
    if (i == j) {
```

```
            printf("A%d", i);
        } else {
            printf("(");
            print_optimal_matrix(n, s, i, s[i][j]);
            print_optimal_matrix(n, s, s[i][j] + 1, j);
            printf(")");
        }
}

int main()
{
        int n = NUM;
        int p[NUM] = { 30, 35, 15, 5, 10, 20, 25 };
        int m[NUM][NUM] = { {0} };
        int s[NUM][NUM] = { {0} };
        memoized_matrix_chain(n, p, m, s);
        printf("最优加全部括号\n");
        print_optimal_matrix(n, s, 1, 6);
        printf("\n");
        printf("输出 m 数组的值\n");
        for (int i = 1; i < n; ++i) {
            for (int j = 1; j < n; ++j) {
                printf("%5d ", m[i][j] == INT_MAX ? 0 : m[i][j]);
            }
            printf("\n");
        }
        printf("输出 s 数组的值\n");
        for (int i = 1; i < n; ++i) {
            for (int j = 1; j < n; ++j) {
                printf("%5d ", s[i][j]);
            }
            printf("\n");
        }
        return 0;
}
```

15.4 最长公共子串

15.4.1 求最长公共子串

```
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>
enum direction_enum {
	direction_up,
	direction_left,
	direction_left_up
};
void lcs_length(char *X, char *Y, int m, int n, int c[][n], int b[][n])
{
	for (int i = 0; i < m; ++i) {
		c[i][0] = 0;
	}
	for (int j = 0; j < n; ++j) {
		c[0][j] = 0;
	}
	for (int i = 1; i < m; ++i) {
		for (int j = 1; j < n; ++j) {
			if (X[i] == Y[j]) {
				c[i][j] = c[i - 1][j - 1] + 1;
				b[i][j] = direction_left_up;
			} else {
				if (c[i - 1][j] >= c[i][j - 1]) {
					c[i][j] = c[i - 1][j];
					b[i][j] = direction_up;
				} else {
					c[i][j] = c[i][j - 1];
					b[i][j] = direction_left;
				}
			}
		}
	}
}

void print_lcs(int n,char *X, int b[][n], int i, int j)
{
	if (i == 0 || j == 0)
		return;
	if (b[i][j] == direction_left_up) {
		print_lcs(n,X, b, i - 1, j - 1);
		printf("%c", X[i]);
	} else {
		if (b[i][j] == direction_up) {
			print_lcs(n,X, b, i - 1, j);
		} else {
			print_lcs(n,X, b, i, j - 1);
```

```
        }
    }
}

int main()
{
    char X[] = "0ABCBDAB";    //X,Y 的有效字符的位置从 1 开始，前面的 0 是用来填充
    char Y[] = "0BDCABA";
    int m=strlen(X);
    int n=strlen(Y);
    int c[m][n];
    int b[m][n];
    lcs_length(X, Y, m, n, c, b);
    print_lcs(n,X, b, m-1, n-1);
    printf("\n");
    printf("输出 C 数组:\n");
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            printf("%d ", c[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

15.4.2 求最长公共子串,不使用 b 数组

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
enum direction_enum {
    direction_up,
    direction_left,
    direction_left_up
};
void lcs_length(char *X, char *Y, int m, int n, int c[][n])
{
    for (int i = 0; i < m; ++i) {
        c[i][0] = 0;
    }
    for (int j = 0; j < n; ++j) {
        c[0][j] = 0;
```

```
    }
    for (int i = 1; i < m; ++i) {
        for (int j = 1; j < n; ++j) {
            if (X[i] == Y[j]) {
                c[i][j] = c[i - 1][j - 1] + 1;
            } else {
                if (c[i - 1][j] >= c[i][j - 1]) {
                    c[i][j] = c[i - 1][j];
                } else {
                    c[i][j] = c[i][j - 1];
                }
            }
        }
    }
}

void print_lcs(int n,char *X, int c[][n], int i, int j)
{
    if (i == 0 || j == 0)
        return;
    if (c[i][j] == c[i - 1][j - 1] + 1) {
        print_lcs(n,X, c, i - 1, j - 1);
        printf("%c", X[i]);
    } else {
        if (c[i][j] == c[i - 1][j]) {
            print_lcs(n,X, c, i - 1, j);
        } else {
            print_lcs(n,X, c, i, j - 1);
        }
    }
}

int main()
{
    char X[] = "0ABCBDAB";    //X,Y 的有效字符的位置从 1 开始，前面的 0 是
用来填充
    char Y[] = "0BDCABA";
    int m=strlen(X);
    int n=strlen(Y);
    int c[m][n];
    lcs_length(X, Y, m, n, c);
    print_lcs(n,X, c, m-1, n-1);
    printf("\n");
    printf("输出 C 数组:\n");
```

```
	for (int i = 0; i < m; ++i) {
		for (int j = 0; j < n; ++j) {
			printf("%d ", c[i][j]);
		}
		printf("\n");
	}
	return 0;
}
```

15.4.3 求最长公共子串,X,Y 字符串有效字符从 0 开始算

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
enum direction_enum {
	direction_up,
	direction_left,
	direction_left_up
};
void lcs_length(char *X, char *Y, int m, int n, int c[][n])
{
	for (int i = 0; i < m; ++i) {
		for (int j = 0; j < n; ++j) {
			if (X[i] == Y[j]) {
				if (i > 0 && j > 0) {
					c[i][j] = c[i - 1][j - 1] + 1;
				} else {
					c[i][j] = 1;
				}
			} else {
				int a = i > 0 ? c[i - 1][j] : 0;
				int b = j > 0 ? c[i][j - 1] : 0;
				if (a >= b) {
					c[i][j] = a;
				} else {
					c[i][j] = b;
				}
			}
		}
	}
}

void print_lcs(int n,char *X, int c[][n], int i, int j)
```

```
{
	if (i == 0 || j == 0)
		return;
	if (c[i][j] == c[i - 1][j - 1] + 1) {
		print_lcs(n,X, c, i - 1, j - 1);
		printf("%c", X[i]);
	} else {
		if (c[i][j] == c[i - 1][j]) {
			print_lcs(n,X, c, i - 1, j);
		} else {
			print_lcs(n,X, c, i, j - 1);
		}
	}
}

int main()
{
	char X[] = "ABCBDAB";
	char Y[] = "BDCABA";
	int m=strlen(X);
	int n=strlen(Y);
	int c[m][n];
	lcs_length(X, Y, 7, 6, c);
	print_lcs(n,X, c, m-1, n-1);
	printf("\n");
	printf("输出 C 数组:\n");
	for (int i = 0; i < m; ++i) {
		for (int j = 0; j < n; ++j) {
			printf("%d ", c[i][j]);
		}
		printf("\n");
	}
	return 0;
}
```

15.5 最优二叉查找树

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
void optimal_bst(float p[], float q[], int n, float e[][n], float w[][n],
		int root[][n])
{
```

```
    for (int i = 1; i < n + 1; ++i) {
        e[i][i - 1] = q[i - 1];
        w[i][i - 1] = q[i - 1];
    }
    for (int l = 1; l < n; ++l) {
        for (int i = 1; i < n - l + 1; ++i) {
            int j = i + l - 1;
            e[i][j] = INT_MAX;
            w[i][j] = w[i][j - 1] + p[j] + q[j];
            for (int r = i; r <= j; ++r) {
                float t = e[i][r - 1] + e[r + 1][j] + w[i][j];
                if (t < e[i][j]) {
                    e[i][j] = t;
                    root[i][j] = r;
                }
            }
        }
    }
}

int main()
{
    float p[] = { INT_MAX, 0.15, 0.10, 0.05, 0.10, 0.20 };  // 忽略第一个，
INT_MAX 是随便设置的值
    float q[] = { 0.05, 0.10, 0.05, 0.05, 0.05, 0.10 };
    int n=sizeof(p)/sizeof(p[0]);
    float e[n+1][n];
    float w[n+1][n];
    int root[n+1][n];
    optimal_bst(p, q, n, e, w, root);
    printf("输出 e 数组的值:\n");
    for (int i = 1; i < n+1; i++) {
        for (int j = 0; j < n; j++) {
            if (j == i - 1 || i <= j) {
                printf("%-4.2f ",e[i][j]);
            }
        }
        printf("\n");
    }
    printf("输出 w 数组的值:\n");
    for (int i = 1; i < n+1; i++) {
        for (int j = 0; j < n; j++) {
            if (j == i - 1 || i <= j) {
                printf("%-4.2f ",w[i][j]);
```

```
                }
            }
            printf("\n");
        }
        printf("输出 root 数组的值:\n");
        for (int i = 1; i < n+1; i++) {
            for (int j = 1; j < n; j++) {
                if (i <= j) {
                    printf("%4d ",root[i][j]);
                }
            }
            printf("\n");
        }
        return 0;
}
```

第 16 章 贪心算法

16.1 活动选择问题

```
#include <stdio.h>
void recursive_activity_select(int s[], int f[], int i, int j,
                              int select_set[], int *select_num)
{
        int m = i + 1;
        while (m < j && s[m] < f[i]) {
            ++m;
        }
        if (m < j) {
            select_set[(*select_num)++] = m;
            recursive_activity_select(s, f, m, j, select_set, select_num);
        }
}

void greedy_activity_select(int s[], int f[], int n, int select_set[],
                        int *select_num)
{
        int i = 1;
        select_set[(*select_num)++] = 1;
        for (int m = 2; m <= n; ++m) {
            if (s[m] >= f[i]) {
```

```
            select_set[(*select_num)++] = m;
            i = m;
        }
    }
}

int main()
{
    /*前面的 0 是添加的，有效数组从下标 1 开始*/
    int s[] = { 0, 1, 3, 0, 5, 3, 5, 6, 8, 8, 2, 12 };
    int f[] = { 0, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 };
    int num=sizeof(s)/sizeof(s[0])-1;
    int select_set[num];
    int select_num = 0;
    //recursive_activity_select(s, f, 0, num+1, select_set, &select_num);
    greedy_activity_select(s,f,num,select_set,&select_num);
    printf("最大相互兼容活动子集:\n");
    for (int i = 0; i < select_num; i++) {
        printf("%d ",select_set[i]);
    }
    printf("\n");
    return 0;
}
```

16.3 赫夫曼编码

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
struct tree_node {
    int frequency;
    char c;
    struct tree_node *left;
    struct tree_node *right;
};
typedef struct priority_queue_type *priority_queue;
struct priority_queue_type {
    int heap_size;
    void **array;
    int (*comp) (const void *, const void *);
};
```

```
int parent(int i)
{
	return (i - 1) / 2;
}

int left_child(int i)
{
	return i * 2 + 1;
}

int right_child(int i)
{
	return i * 2 + 2;
}

void swap(void *a, void *b, size_t elem_size)
{
	if (a == NULL || b == NULL || a == b)
		return;
	char temp[elem_size];	/*变长数组 */
	memcpy(temp, a, elem_size);
	memcpy(a, b, elem_size);
	memcpy(b, temp, elem_size);
}

void heapify(priority_queue pq, int i)
{
	int left = left_child(i);
	int right = right_child(i);
	int largest = i;
	if (left < pq->heap_size
		 && pq->comp(pq->array[largest], pq->array[left]) < 0) {
		largest = left;
	}
	if (right < pq->heap_size
		 && pq->comp(pq->array[largest], pq->array[right]) < 0) {
		largest = right;
	}
	if (largest != i) {
		swap(&pq->array[i], &pq->array[largest], sizeof(void *));
		heapify(pq, largest);
	}
}
```

```
void fix_up(priority_queue pq, int i)
{
	while (i > 0 && pq->comp(pq->array[parent(i)], pq->array[i]) < 0) {
		swap(&pq->array[parent(i)], &pq->array[i], sizeof(void *));
		i = parent(i);
	}
}

priority_queue priority_queue_create(int n_length,
				int (*comp) (const void *, const void *))
{
	priority_queue pq = malloc(sizeof(struct priority_queue_type));
	pq->array = malloc(sizeof(void *) * n_length);
	pq->heap_size = 0;
	pq->comp = comp;
	return pq;
}

void *priority_queue_top(priority_queue pq)
{
	return pq->array[0];
}

/*去掉并返回堆的第一个元素 */
void *priority_queue_extract_top(priority_queue pq)
{
	swap(&pq->array[0], &pq->array[pq->heap_size - 1], sizeof(void *));
	--pq->heap_size;
	heapify(pq, 0);
	return pq->array[pq->heap_size];
}

/*把元素 key 插入队列 */
void priority_queue_insert(priority_queue pq, void *key)
{
	++pq->heap_size;
	int i = pq->heap_size - 1;
	memcpy(&pq->array[i], &key, sizeof(void *));
	fix_up(pq, i);
}

bool priority_queue_is_empty(priority_queue pq)
{
	return pq->heap_size == 0;
```

```
}

void priority_queue_destroy(priority_queue pq, void (*free_key) (void *))
{
	while (!priority_queue_is_empty(pq)) {
		void *p = priority_queue_extract_top(pq);
		free_key(p);
	}
	free(pq->array);
	free(pq);
}

void tree_node_ini(struct tree_node *t, char c, int frequency)
{
	t->c = c;
	t->frequency = frequency;
	t->left = NULL;
	t->right = NULL;
}

/*最小堆的比较函数*/
int cmp_node(const void *pa, const void *pb)
{
	const struct tree_node *pleft = pa;
	const struct tree_node *pright = pb;
	return pright->frequency - pleft->frequency;
}

struct tree_node *huffman(int n, char char_array[], int frequency_array[])
{
	priority_queue pq = priority_queue_create(n, cmp_node);
	for (int i = 0; i < n; i++) {
		struct tree_node *node = malloc(sizeof(struct tree_node));
		tree_node_ini(node, char_array[i], frequency_array[i]);
		priority_queue_insert(pq, node);
	}
	for (int i = 0; i < n - 1; i++) {
		struct tree_node *z = malloc(sizeof(struct tree_node));
		tree_node_ini(z, 0, 0);
		struct tree_node *x = priority_queue_extract_top(pq);
		struct tree_node *y = priority_queue_extract_top(pq);
		z->left = x;
		z->right = y;
		z->frequency = x->frequency + y->frequency;
```

```
        priority_queue_insert(pq, z);
    }
    struct tree_node *root = priority_queue_extract_top(pq);
    priority_queue_destroy(pq, free);
    return root;
}

void create_huffman_code_table(struct tree_node *node, char *str_code, int n,
                               char huffman_code_table[][n])
{
    if (node == NULL) {
        return;
    }
    if (node->left == NULL && node->right == NULL) {
        strcpy(huffman_code_table[(int)node->c], str_code);
        return;
    }
    if (node->left != NULL) {
        char str[n];
        strcpy(str, str_code);
        strcat(str, "0");
        create_huffman_code_table(node->left, str, n,
                                  huffman_code_table);
    }
    if (node->right != NULL) {
        char str[n];
        strcpy(str, str_code);
        strcat(str, "1");
        create_huffman_code_table(node->right, str, n,
                                  huffman_code_table);
    }
}

void encode_huffman_code(char *str, char *result,
                         int n, char huffman_code_table[][n])
{
    strcpy(result, "");
    int len = strlen(str);
    for (int i = 0; i < len; ++i) {
        strcat(result, huffman_code_table[(int)str[i]]);
    }
}

int get_huffman_char(char *str, int i, char *result, struct tree_node *node)
```

```
{
    if (node->left == NULL && node->right == NULL) {
        int len = strlen(result);
        result[len] = node->c;
        result[len + 1] = '\0';
        return i; //返回当前解码的位置
    }
    if (str[i] == '0')    //继续解码
    {
        return get_huffman_char(str, i + 1, result, node->left);
    } else {
        return get_huffman_char(str, i + 1, result, node->right);
    }
}

void decode_huffman_code(char *str, char *result, struct tree_node *node)
{
    int len = strlen(str);
    for (int i = 0; i < len;) {
        i = get_huffman_char(str, i, result, node);
    }
}

void tree_delete_node(struct tree_node *x, void (*free_key) (void *))
{
    if (x != NULL) {
        tree_delete_node(x->left, free_key);
        tree_delete_node(x->right, free_key);
        free(x);
    }
}

int main()
{
    char char_array[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
    int num = sizeof(char_array) / sizeof(char_array[0]);
    int frequency_array[] = { 45, 13, 12, 16, 9, 5 };
    int code_len = 10;
    char huffman_code_table[256][code_len];
    struct tree_node *root = huffman(num, char_array, frequency_array);
    create_huffman_code_table(root, "", code_len, huffman_code_table);
    for (int i = 0; i < 6; ++i) {
        printf("%c:%s\n", char_array[i],
                huffman_code_table[(int)char_array[i]]);
```

```
	}
	char str[] = "aabe";
	char result[256] = { 0 };
	encode_huffman_code(str, result, code_len, huffman_code_table);
	printf("%s 的 huffman 编码是:%s\n", str, result);
	strcpy(str, "");
	decode_huffman_code(result, str, root);
	printf("%s 的 huffman 编码解码的结果是:%s\n", result, str);
	tree_delete_node(root, free);
	return 0;
}
```

第 18 章 B 树

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
/*两个约定:(1)B 树的根结点始终在主存中，因而无需对根做 DISK_READ,
  但是根结点被改变后，都需要对根结点做一次 DISK_WRITE
(2)任何被当作参数的结点被传递之前，要先对它们做一次 DISK_READ*/
#define DISK_READ(x)
#define DISK_WRITE(x)
/*B 树的最小度数*/
enum { tree_degree = 3 };
typedef struct tree_type *tree;
struct tree_node {
	int num;
	void **key;
	struct tree_node **child;
	bool leaf;
};
struct tree_type {
	int (*comp) (const void *, const void *);
	struct tree_node *root;
};
void tree_node_ini(struct tree_node *n, int num, bool leaf)
{
	n->num = num;
	n->leaf = leaf;
	int full_key_num = 2 * tree_degree - 1;
```

```
    int full_child_num = full_key_num + 1;
    n->key = malloc(sizeof(void *) * full_key_num);
    memset(n->key, 0, sizeof(void *) * full_key_num);
    n->child = malloc(sizeof(struct tree_node *) * full_child_num);
    memset(n->child, 0, sizeof(struct tree_node *) * (full_child_num));
}

void tree_node_delete_key_child(struct tree_node *x, int key_pos, int child_pos)
{
    memmove(&x->key[key_pos], &x->key[key_pos + 1],
        sizeof(void *) * (x->num - key_pos - 1));
    memmove(&x->child[child_pos], &x->child[child_pos + 1],
        sizeof(struct tree_node *) * (x->num - child_pos));
    --x->num;
}

void tree_node_insert_key_child(struct tree_node *x, void *key, int key_pos,
                struct tree_node *child, int child_pos)
{
    memmove(&x->key[key_pos], &x->key[key_pos + 1],
        sizeof(void *) * (x->num - key_pos));
    x->key[key_pos] = key;
    memmove(&x->child[child_pos], &x->child[child_pos + 1],
        sizeof(struct tree_node *) * (x->num + 1 - child_pos));
    x->child[child_pos] = child;
    ++x->num;
}

tree tree_create(int (*comp) (const void *, const void *))
{
    tree t = malloc(sizeof(struct tree_type));
    t->comp = comp;
    struct tree_node *x = malloc(sizeof(struct tree_node));
    tree_node_ini(x, 0, true);
    DISK_WRITE(x);
    t->root = x;
    return t;
}

void tree_node_destroy(struct tree_node *x)
{
    free(x->key);
    free(x->child);
    free(x);
```

```c
}

void tree_destroy_all_node(struct tree_node *x, void (*free_key) (void *))
{
	if (x == NULL)
		return;
	for (int i = 0; i < x->num + 1; ++i) {
		tree_destroy_all_node(x->child[i], free_key);
	}
	for (int i = 0; i < x->num; i++) {
		free_key(x->key[i]);
	}
	free(x->key);
	free(x->child);
	free(x);
}

void tree_destroy(tree t, void (*free_key) (void *))
{
	tree_destroy_all_node(t->root, free_key);
	free(t);
}

struct tree_node *tree_search(tree t, struct tree_node *x, void *key,
			      int *index)
{
	int i = 0;
	while (i < x->num && t->comp(key, x->key[i]) > 0) {
		++i;
	}
	if (i < x->num && t->comp(key, x->key[i]) == 0) {
		*index = i;
		return x;
	}
	if (x->leaf) {
		return NULL;
	} else {
		DISK_READ(x->child[i]);
		return tree_search(t, x->child[i], key, index);
	}
}

//前序遍历
void tree_preorder_walk(struct tree_node *x, int depth,
```

```
                void (*handle) (const void *))
{
        if (x != NULL) {
                printf("depth:%d ", depth);
                printf("key:(");
                for (int i = 0; i < x->num; i++) {
                        handle(x->key[i]);
                        if (i < x->num - 1) {
                                printf(",");
                        }
                }
                printf(")\n");
                for (int i = 0; i < x->num + 1; ++i) {
                        tree_preorder_walk(x->child[i], depth + 1, handle);
                }
        }
}

void tree_split_child(struct tree_node *x, int i, struct tree_node *y)
{
        struct tree_node *z = malloc(sizeof(struct tree_node));
        tree_node_ini(z, tree_degree - 1, y->leaf);
        memcpy(z->key, &y->key[tree_degree],
                sizeof(void *) * (tree_degree - 1));
        if (!y->leaf) {
                memcpy(z->child, &y->child[tree_degree],
                        sizeof(struct tree_node *) * tree_degree);
        }
        y->num = tree_degree - 1;
        tree_node_insert_key_child(x, y->key[tree_degree - 1], i, z, i + 1);
        DISK_WRITE(y);
        DISK_WRITE(z);
        DISK_WRITE(x);
}

void tree_union_child(tree t, struct tree_node *x, int i, struct tree_node *y,
                   struct tree_node *z)
{
        void *key = x->key[i];
        y->key[y->num] = key;
        memcpy(&y->key[y->num + 1], z->key, sizeof(void *) * z->num);
        memcpy(&y->child[y->num + 1], z->child, sizeof(void *) * (z->num + 1));
        y->num += z->num + 1;
        tree_node_delete_key_child(x, i, i + 1);
```

```
    tree_node_destroy(z);       //把 z 释放掉
    DISK_WRITE(y);
    DISK_WRITE(x);
    if (x == t->root && x->num == 0) //如果 x 是根，并没有元素了
    {
        t->root = y;
        tree_node_destroy(x);
    }
}

void tree_insert_not_full(tree t, struct tree_node *x, void *key)
{
    int i = x->num - 1;
    if (x->leaf) {
        while (i >= 0 && t->comp(key, x->key[i]) < 0) {
            x->key[i + 1] = x->key[i];
            --i;
        }
        x->key[i + 1] = key;
        ++x->num;
        DISK_WRITE(x);
        return;
    }
    while (i >= 0 && t->comp(key, x->key[i]) < 0) {
        --i;
    }
    ++i;
    DISK_READ(x->child[i]);
    if (x->child[i]->num == 2 * tree_degree - 1) {
        tree_split_child(x, i, x->child[i]);
        if (t->comp(key, x->key[i]) > 0) {
            ++i;
        }
    }
    tree_insert_not_full(t, x->child[i], key);
}

void tree_insert(tree t, void *key)
{
    struct tree_node *r = t->root;
    if (r->num == 2 * tree_degree - 1) {
        struct tree_node *s = malloc(sizeof(struct tree_node));
        tree_node_ini(s, 0, false);
        t->root = s;
```

```c
		s->child[0] = r;
		tree_split_child(s, 0, r);
		tree_insert_not_full(t, s, key);
	} else {
		tree_insert_not_full(t, r, key);
	}
}

struct tree_node *tree_successor(struct tree_node *x)
{
	while (x != NULL && x->child[0] != NULL) {
		x = x->child[0];
	}
	return x;
}

struct tree_node *tree_predecessor(struct tree_node *x)
{
	while (x != NULL && x->child[x->num] != NULL) {
		x = x->child[x->num];
	}
	return x;
}

void swap(void *a, void *b, size_t elem_size)
{
	if (a == NULL || b == NULL || a == b)
		return;
	char temp[elem_size];	/*变长数组 */
	memcpy(temp, a, elem_size);
	memcpy(a, b, elem_size);
	memcpy(b, temp, elem_size);
}

struct tree_node *tree_delete(tree t, struct tree_node *x, void *key, int *i);
//情况 1,如果关键字 k 在结点 x 中而且 x 是个叶结点
struct tree_node *tree_delete_from_leaf(struct tree_node *x, int *i)
{
	void *key = x->key[*i];
	tree_node_delete_key_child(x, *i, *i);
	x->key[x->num] = key;
	*i = x->num;		//i 保存了删掉的 key 的位置
	return x;
}
```

```
//情况 2,如果关键字 k 在结点 x 中而且 x 是个内结点
struct tree_node *tree_delete_from_node(tree t, struct tree_node *x, void *key,
                    int *i)
{
    struct tree_node *y = x->child[*i];
    //情况 2a,结点 x 中前于 k 的子结点 y 包含至少 tree_degree 个关键字
    if (y->num >= tree_degree) {
        struct tree_node *predecessor = tree_predecessor(y);
        swap(&x->key[*i], &predecessor->key[predecessor->num - 1],
             sizeof(void *));
        *i = predecessor->num - 1;
        return tree_delete_from_leaf(predecessor, i);
    }
    struct tree_node *z = x->child[*i + 1];
    //情况 2b,结点 x 中位于 k 之后的子结点包含至少 tree_degree 个关键字
    if (z->num >= tree_degree) {
        struct tree_node *successor = tree_successor(y);
        swap(&x->key[*i], &successor->key[0], sizeof(void *));
        *i = 0;
        return tree_delete_from_leaf(successor, i);
    }
    //情况 2c,y 和 z 都只有 tree_degree-1 个关键字
    tree_union_child(t, x, *i, y, z);
    return tree_delete(t, y, key, i);
}

//如果关键字 k 不在内结点 x 中,则确定必包含 k 的正确的子树的根 c
struct tree_node *tree_delete_from_child(tree t, struct tree_node *x, void *key,
                     int *i)
{
    DISK_READ(x->child[i]);
    struct tree_node *p_child = x->child[*i];
    if (p_child->num >= tree_degree) {
        return tree_delete(t, p_child, key, i);
    }
    //情况 3a，p_child 只包含 tree_degree-1 个关键字
    struct tree_node *y = NULL;
    //p_child 不是最左子结点，则有左兄弟
    if (*i > 0) {
        DISK_READ(x->child[*i - 1]);
        y = x->child[*i - 1];
    }
    if (y != NULL && y->num >= tree_degree) {
```

```
        tree_node_insert_key_child(p_child, x->key[*i - 1], 0,
                                   y->child[y->num], 0);
        x->key[*i - 1] = y->key[y->num - 1];
        tree_node_delete_key_child(y, y->num - 1, y->num);
        return tree_delete(t, p_child, key, i);
    }
    struct tree_node *z = NULL;
    //p_child 不是最右子结点，则有右兄弟
    if (*i < x->num) {
        DISK_READ(x->child[*i + 1]);
        z = x->child[*i + 1];
    }
    if (z != NULL && z->num >= tree_degree) {
        tree_node_insert_key_child(p_child, x->key[*i],
                                   p_child->num, z->child[0],
                                   p_child->num + 1);
        x->key[*i] = z->key[0];
        tree_node_delete_key_child(z, 0, 0);
        return tree_delete(t, p_child, key, i);
    }
    //情况 3b,p_child 及其兄弟都包含 tree_degree-1 个关键字，p_chaild 合并进左
兄弟
    if (y != NULL) {
        tree_union_child(t, x, *i - 1, y, p_child);
        return tree_delete(t, y, key, i);
    }
    //情况 3b,p_child 及其兄弟都包含 tree_degree-1 个关键字，右兄弟合并进
p_child
    if (z != NULL) {
        tree_union_child(t, x, *i, p_child, z);
        return tree_delete(t, p_child, key, i);
    }
    return NULL;
}

struct tree_node *tree_delete(tree t, struct tree_node *x, void *key, int *i)
{
    *i = 0;
    while (*i < x->num && t->comp(key, x->key[*i]) > 0) {
        ++*i;
    }
    if (*i < x->num && t->comp(key, x->key[*i]) == 0) {
        if (x->leaf) {
            return tree_delete_from_leaf(x, i);
```

```
		} else {
			return tree_delete_from_node(t, x, key, i);
		}
	}
	return tree_delete_from_child(t, x, key, i);
}

int cmp_int(const void *p1, const void *p2)
{
	const int *pa = p1;
	const int *pb = p2;
	if (*pa < *pb)
		return -1;
	if (*pa == *pb)
		return 0;
	return 1;
}
void print_key(const void *key)
{
	const int *p = key;
	printf("%d", *p);
}

int main()
{
	printf("minimum degree of the B-tree:%d\n", tree_degree);
	tree t = tree_create(cmp_int);
	for (int i = 0; i < 20; i++) {
		int *p = malloc(sizeof(int));
		*p = i;
		tree_insert(t, p);
	}
	printf("前序遍历:\n");
	tree_preorder_walk(t->root, 0, print_key);
	int index;
	int key = 0;
	struct tree_node *p = tree_search(t, t->root, &key, &index);
	if (p != NULL) {
		printf("查找关键字:%d 成功\n", key);
		printf("删除关键字:%d\n", key);
		struct tree_node *del = tree_delete(t, t->root, &key, &index);
		if (del != NULL) {
			free(del->key[index]);
		}
```

```
		p = tree_search(t, t->root, &key, &index);
		if (p == NULL) {
			printf("删除关键字:%d 成功\n", key);
		}
		printf("删除后,前序遍历:\n");
		tree_preorder_walk(t->root, 0, print_key);
	}
	tree_destroy(t, free);
	return 0;
}
```

第 19 章 二项堆

```
#include <stdio.h>
#include <limits.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>
typedef struct binomial_heap *heap;
struct heap_node {
	void *key;
	int degree;
	struct heap_node *child;
	struct heap_node *sibling;
	struct heap_node *parent;
};
struct binomial_heap {
	int (*comp) (const void *, const void *);
	//这个函数是用于结点交换时通知调用
	void (*on_swap) (struct heap_node *, struct heap_node *);
	struct heap_node *head;
};
void swap(void *a, void *b, size_t elem_size)
{
	if (a == NULL || b == NULL || a == b)
		return;
	char temp[elem_size];	/*变长数组 */
	memcpy(temp, a, elem_size);
	memcpy(a, b, elem_size);
	memcpy(b, temp, elem_size);
}
```

```
void heap_node_ini(struct heap_node *x, void *key)
{
    x->key = key;
    x->degree = 0;
    x->parent = NULL;
    x->child = NULL;
    x->sibling = NULL;
}

heap heap_create(int (*comp) (const void *, const void *),
         void (*on_swap) (struct heap_node *, struct heap_node *))
{
    heap h = malloc(sizeof(struct binomial_heap));
    h->comp = comp;
    h->on_swap = on_swap;
    h->head = NULL;
    return h;
}

//返回一个指针，它指向包含 n 个结点的二项堆 H 中具有最小关键字的结点
struct heap_node *heap_minimum(heap h)
{
    struct heap_node *y = NULL;
    struct heap_node *x = h->head;
    void *min;
    bool first = true;
    while (x != NULL) {
        if (first || h->comp(x->key, min) < 0) {
            first = false;
            min = x->key;
            y = x;
        }
        x = x->sibling;
    }
    return y;
}

bool heap_is_empty(heap h)
{
    return h->head == NULL;
}

//将结点 y 为根和 z 为根的树连接过来，使 z 成为 y 的父结点
void link(struct heap_node *y, struct heap_node *z)
```

```
{
    y->parent = z;
    y->sibling = z->child;
    z->child = y;
    z->degree = z->degree + 1;
}

void heap_destroy(heap h);
//将 ha 和 hb 合并成一个按度数的单调递增次序排列的链表
struct heap_node *heap_merge(heap ha, heap hb)
{
    struct heap_node *pa = ha->head;
    struct heap_node *pb = hb->head;
    struct heap_node *head = NULL;
    struct heap_node *tail = NULL;
    while (pa != NULL && pb != NULL) {
        if (pa->degree <= pb->degree) {
            if (head == NULL) {
                head = pa;
                tail = pa;
                pa = pa->sibling;
                tail->sibling = NULL;
            } else {
                tail->sibling = pa;
                pa = pa->sibling;
                tail = tail->sibling;
                tail->sibling = NULL;
            }
        } else {
            if (head == NULL) {
                head = pb;
                tail = pb;
                pb = pb->sibling;
                tail->sibling = NULL;
            } else {
                tail->sibling = pb;
                pb = pb->sibling;
                tail = tail->sibling;
                tail->sibling = NULL;
            }
        }
    }
    if (pa != NULL && pb == NULL) {
        if (head == NULL) {
```

```
            head = pa;
            tail = pa;
        } else {
            tail->sibling = pa;
        }
    }
    if (pa == NULL && pb != NULL) {
        if (head == NULL) {
            head = pb;
            tail = pb;
        } else {
            tail->sibling = pb;
        }
    }
    hb->head = NULL;
    heap_destroy(hb);
    return head;
}

//将 hb 合并到 ha 中
void heap_union(heap ha, heap hb)
{
    //将 ha 和 hb 的根表合并成一个按度数的单调递增次序排列的链表
    ha->head = heap_merge(ha, hb);
    if (ha->head == NULL) {
        return;
    }
    struct heap_node *prev = NULL;
    struct heap_node *x = ha->head;
    struct heap_node *next = x->sibling;
    while (next != NULL) {
        //情况 1:x->degree!=next->degree
        //情况 2:x->degree==next->degree==next->sibling->degree
        if ((x->degree != next->degree) ||
             (next->sibling != NULL
              && next->sibling->degree == x->degree)) {
            prev = x;
            x = next;
        } else if (ha->comp(x->key, next->key) <= 0) {
            //                                  情                                  况
3:x->degree==next->degree!=next->sibling->degree,x->key<=next->key
            x->sibling = next->sibling;
            link(next, x);
        } else {
```

```
            //                                  情                                  况
4:x->degree==next->degree!=next->sibling->degree,next->key<=x->key
            if (prev == NULL) {
                ha->head = next;
            } else {
                prev->sibling = next;
            }
            link(x, next);
            x = next;
        }
        next = x->sibling;
    }
}

//反转 x 的孩子，随便把 x 的孩子的父结点置为空
void reverse_children(struct heap_node *x)
{
    if (x == NULL || x->child == NULL)
        return;
    struct heap_node *prev = x->child;
    struct heap_node *current = prev->sibling;
    struct heap_node *next = NULL;
    while (current != NULL) {
        next = current->sibling;
        current->sibling = prev;
        current->parent = NULL;
        prev = current;
        current = next;
    }
    x->child->sibling = NULL;
    x->child->parent = NULL;
    x->child = prev;
}

//下面的过程将结点 x 插入二项堆中，假定结点 x 已被分配，且 key[x]也已填有
内容
void heap_insert(heap h, struct heap_node *x)
{
    heap hb = heap_create(h->comp, h->on_swap);
    hb->head = x;
    heap_union(h, hb);
}

struct heap_node *heap_remove_minimum(heap h)
```

```
{
	struct heap_node *x = h->head;
	if (x == NULL)
		return NULL;
	struct heap_node *prev = NULL;
	struct heap_node *min_prev = NULL;
	void *min;
	bool first = true;
	while (x != NULL) {
		if (first || h->comp(x->key, min) < 0) {
			first = false;
			min = x->key;
			min_prev = prev;
		}
		prev = x;
		x = x->sibling;
	}
	//删除结点 x
	if (min_prev == NULL) {
		x = h->head;
		h->head = x->sibling;
	} else {
		x = min_prev->sibling;
		min_prev->sibling = x->sibling;
	}
	return x;
}

//抽取具有最小关键字的结点，并返回一个指向该结点的指针
struct heap_node *heap_extract_min(heap h)
{
	struct heap_node *x = heap_remove_minimum(h);
	if (x == NULL)
		return NULL;
	reverse_children(x);
	heap hb = heap_create(h->comp, h->on_swap);
	hb->head = x->child;
	heap_union(h, hb);
	return x;
}

//将二项堆中的某一结点 x 的关键字减少为一个新值 k，如果 k 大于 x 的当前关键字值，直接返回
void heap_decrease_key(heap h, struct heap_node *x)
```

```
{
    struct heap_node *y = x;
    struct heap_node *z = y->parent;
    while (z != NULL && h->comp(y->key, z->key) < 0) {
        swap(&y->key, &z->key, sizeof(void *));
        if (h->on_swap != NULL) {
            h->on_swap(y, z);
        }
        y = z;
        z = y->parent;
    }
}

void display_node(struct heap_node *x, void (*print_key) (const void *))
{
    print_key(x->key);
    printf(" ");
    if (x->child != NULL) {
        display_node(x->child, print_key);
    }
    if (x->sibling != NULL) {
        display_node(x->sibling, print_key);
    }
}

void heap_display(heap h, void (*print_key) (const void *))
{
    display_node(h->head, print_key);
    printf("\n");
}

void heap_destroy(heap h)
{
    while (!heap_is_empty(h)) {
        struct heap_node *x = heap_extract_min(h);
        free(x->key);
        free(x);
    }
    free(h);
}

void on_swap(struct heap_node *left, struct heap_node *right)
{
    printf("%d 和%d 发生了交换\n", *(int *)left->key,
```

```
            *(int *)right->key);
}

int cmp_int(const void *p1, const void *p2)
{
    const int *pa = p1;
    const int *pb = p2;
    if (*pa < *pb)
        return -1;
    if (*pa == *pb)
        return 0;
    return 1;
}

void print_key(const void *key)
{
    const int *p = key;
    printf("%d", *p);
}

int main()
{
    heap h = heap_create(cmp_int, on_swap);
    struct heap_node *x = NULL;
    struct heap_node *parray[10];
    for (int i = 0; i < 10; i++) {
        struct heap_node *x = malloc(sizeof(struct heap_node));
        int *ip = malloc(sizeof(int));
        *ip = i;
        heap_node_ini(x, ip);
        heap_insert(h, x);
        parray[i] = x;
    }
    printf("原始数据:\n");
    heap_display(h, print_key);
    int change_index = 5;
    *(int*)parray[change_index]->key=INT_MIN;
    heap_decrease_key(h, parray[change_index]);
    printf("修改了第%d 个结点的数据:\n", change_index);
    heap_display(h, print_key);
    heap hb = heap_create(cmp_int, on_swap);
    for (int i = 10; i < 20; i++) {
        struct heap_node *x = malloc(sizeof(struct heap_node));
        int *ip = malloc(sizeof(int));
```

```
        *ip = i;
        heap_node_ini(x, ip);
        heap_insert(hb, x);
    }
    heap_union(h, hb);
    printf("合并了之后的数据:\n");
    heap_display(h, print_key);
    printf("按从小到大的顺序输出:\n");
    while (!heap_is_empty(h)) {
        x = heap_extract_min(h);
        print_key(x->key);
        printf(" ");
        free(x->key);
        free(x);
    }
    printf("\n");
    heap_destroy(h);
    return 0;
}
```

第 20 章 斐波那契堆

```
#include <stdio.h>
#include <limits.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>
#include <stdlib.h>
typedef struct fib_heap *heap;
struct heap_node {
    void *key;
    int degree;
    bool mark;
    struct heap_node *child;
    struct heap_node *left;
    struct heap_node *right;
    struct heap_node *parent;
};
struct fib_heap {
    int (*comp) (const void *, const void *);
    struct heap_node *min;
    int num;
```

```
};
void heap_node_ini(struct heap_node *x, void *key)
{
	x->key = key;
	x->degree = 0;
	x->mark = false;
	x->parent = NULL;
	x->child = NULL;
	x->left = x;
	x->right = x;
}

void swap(void *a, void *b, size_t elem_size)
{
	if (a == NULL || b == NULL || a == b)
		return;
	char temp[elem_size];	/*变长数组 */
	memcpy(temp, a, elem_size);
	memcpy(a, b, elem_size);
	memcpy(b, temp, elem_size);
}

heap heap_create(int (*comp) (const void *, const void *))
{
	heap h = malloc(sizeof(struct fib_heap));
	h->comp = comp;
	h->num = 0;
	h->min = NULL;
	return h;
}

//删除结点，如果只有 x 一个结点的话，这个函数无效
void list_delete(struct heap_node **pos, struct heap_node *x)
{
	if (x->right == x)//只有一个结点
	{
		*pos = NULL;
		return;
	}
	x->left->right = x->right;
	x->right->left = x->left;
	if (*pos == x) {
		*pos = x->right;
	}
```

```
}

//插入结点 x 到 pos 的左边,如果 pos 为空，pos＝x
void list_insert(struct heap_node **pos, struct heap_node *x)
{
	if (*pos == NULL) {
		*pos = x;
		x->left = x;
		x->right = x;
	} else {
		x->left = (*pos)->left;
		(*pos)->left->right = x;
		x->right = (*pos);
		(*pos)->left = x;
	}
}

void add_root(heap h, struct heap_node *x)
{
	list_insert(&h->min, x);
	x->parent = NULL;
	x->mark = false;
	if (h->comp(x->key, h->min->key) < 0) {
		h->min = x;
	}
}

//下面的过程将结点 x 插入斐波那契堆中，假定结点 x 已被分配，且 key[x]也已填有内容
void heap_insert(heap h, struct heap_node *x)
{
	x->degree = 0;
	x->parent = NULL;
	x->child = NULL;
	x->left = x;
	x->right = x;
	add_root(h, x);
	++h->num;
}

//最小结点
struct heap_node *heap_minimum(heap h)
{
	return h->min;
```

```
}

void heap_destroy(heap h);
//将另一个斐波那契堆合并到当前堆，另一堆合并到当前最小结点的右边
void heap_union(heap ha, heap hb)
{
	if (hb == NULL || hb->min == NULL) {
		return;
	}
	if (ha->min == NULL) {
		ha->min = hb->min;
	} else {
		//最小结点的右边结点
		struct heap_node *ha_min_right = ha->min->right;
		ha->min->right = hb->min;
		//另一个堆最小结点的左结点，即最后一个结点
		struct heap_node *hb_min_left = hb->min->left;
		hb->min->left = ha->min;
		hb_min_left->right = ha_min_right;
		ha_min_right->left = hb_min_left;
	}
	if (ha->min == NULL
		 || (hb->min != NULL && ha->comp(hb->min->key, ha->min->key) < 0)) {
		ha->min = hb->min;
	}
	ha->num += hb->num;
	hb->min = NULL;
	heap_destroy(hb);
}

void link(heap h, struct heap_node *y, struct heap_node *x)
{
	list_delete(&h->min, y);
	list_insert(&x->child, y);
	y->parent = x;
	y->mark = false;
	++x->degree;
}

//合并根表
void consolidate(heap h)
{
	if (h->min == NULL)
		return;
```

```
    int D = floor(log(h->num) / log(1.618));     //计算 D 值
    struct heap_node *A[D];
    for (int i = 0; i < D; i++) {
        A[i] = NULL;
    }
    struct heap_node *x = NULL;
    struct heap_node *y = NULL;
    int d;
    struct heap_node *w = h->min;
    struct heap_node *end = h->min->left;
    bool loop_flag = true;
    while (loop_flag) {
        x = w;
        if (w != end) {
            w = w->right;
        } else {
            loop_flag = false;      //w 到达最后一个结点，循环结束
        }
        d = x->degree;
        while (A[d] != NULL) {
            y = A[d];
            if (h->comp(x->key, y->key) > 0) {
                swap(&x, &y, sizeof(struct heap_node *));
            }
            link(h, y, x);
            A[d] = NULL;
            ++d;
        }
        A[d] = x;
    }
    h->min = NULL;
    for (int i = 0; i < D; ++i) {
        if (A[i] != NULL) {
            add_root(h, A[i]);
        }
    }
}

//抽取具有最小关键字的结点，并返回一个指向该结点的指针
struct heap_node *heap_extract_min(heap h)
{
    struct heap_node *z = h->min;
    if (z == NULL)
        return NULL;
```

```
	struct heap_node *x = NULL;
	while (z->degree > 0) {
		x = z->child;
		if (x->right == x) {
			z->child = NULL;
		} else {
			z->child = z->child->right;
		}
		list_delete(&z->child, x);
		add_root(h, x);
		--z->degree;
	}
	if (z == z->right) {
		list_delete(&h->min, z);
	} else {
		list_delete(&h->min, z);
		consolidate(h);
	}
	--h->num;
	return z;
}

void cut(heap h, struct heap_node *x, struct heap_node *y)
{
	list_delete(&y->child, x);
	add_root(h, x);
	--y->degree;
}

void cascading_cut(heap h, struct heap_node *y)
{
	struct heap_node *z = y->parent;
	if (z == NULL)
		return;
	if (y->mark == false) {
		y->mark = true;
	} else {
		cut(h, y, z);
		cascading_cut(h, z);
	}
}

//将斐波那契堆中的某一结点 x 的关键字减少为一个新值 k
void heap_decrease_key(heap h, struct heap_node *x)
```

```
{
	struct heap_node *y = x->parent;
	if (y != NULL && h->comp(x->key, y->key) < 0) {
		cut(h, x, y);
		cascading_cut(h, y);
	}
	if (h->comp(x->key, h->min->key) < 0) {
		h->min = x;
	}
}

bool heap_is_empty(heap h)
{
	return h->min == NULL;
}

void heap_destroy(heap h)
{
	while (!heap_is_empty(h)) {
		struct heap_node *x = heap_extract_min(h);
		free(x->key);
		free(x);
	}
	free(h);
}

void heap_display(heap h, void (*print_key) (const void *))
{
	if (h->min == NULL)
		return;
	struct heap_node *x = h->min;
	bool loop_flag = true;
	struct heap_node *end = h->min->left;
	while (loop_flag) {
		print_key(x->key);
		printf(" ");
		if (x != end) {
			x = x->right;
		} else {
			loop_flag = false;
		}
	}
	printf("\n");
}
```

```
int cmp_int(const void *p1, const void *p2)
{
	const int *pa = p1;
	const int *pb = p2;
	if (*pa < *pb)
		return -1;
	if (*pa == *pb)
		return 0;
	return 1;
}

void print_key(const void *key)
{
	const int *p = key;
	printf("%d", *p);
}

int main()
{
	heap h = heap_create(cmp_int);
	struct heap_node *x = NULL;
	struct heap_node *parray[10];
	for (int i = 0; i < 10; i++) {
		struct heap_node *x = malloc(sizeof(struct heap_node));
		int *ip = malloc(sizeof(int));
		*ip = i;
		heap_node_ini(x, ip);
		heap_insert(h, x);
		parray[i] = x;
	}
	printf("原始数据:\n");
	heap_display(h, print_key);
	int change_index = 5;
	*(int*)parray[change_index]->key=INT_MIN;
	heap_decrease_key(h, parray[change_index]);
	printf("修改了第%d 个结点的数据:\n", change_index);
	heap_display(h, print_key);
	heap hb = heap_create(cmp_int);
	for (int i = 10; i < 20; i++) {
		struct heap_node *x = malloc(sizeof(struct heap_node));
		int *ip = malloc(sizeof(int));
		*ip = i;
		heap_node_ini(x, ip);
```

```
		heap_insert(hb, x);
	}
	heap_union(h, hb);
	printf("合并了之后的数据:\n");
	heap_display(h, print_key);
	printf("按从小到大的顺序输出:\n");
	while (!heap_is_empty(h)) {
		x = heap_extract_min(h);
		print_key(x->key);
		printf(" ");
		free(x->key);
		free(x);
	}
	printf("\n");
	heap_destroy(h);
	return 0;
}
```

第 21 章 用于不相交集合的数据结构

21.2 不相交集体的链表表示

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct set_type *set;
struct set_node {
	void *key;
	struct set_node *next;
	struct set_node *representative;	//指向代表的集合元素
};

struct set_type {
	struct set_node *head;
	struct set_node *tail;
	int num;
};
void set_node_ini(struct set_node *x, void *key)
{
	x->key = key;
	x->next = NULL;
```

```
    x->representative = NULL;
}

set set_create(void *key)
{
    set s = malloc(sizeof(struct set_type));
    struct set_node *x = malloc(sizeof(struct set_node));
    set_node_ini(x, key);
    s->head = x;
    s->tail = x;
    s->num = 1;
    x->representative = x;
    return s;
}

struct set_node *find_set(set s)
{
    return s->head->representative;
}

void update_representative(struct set_node *head,
                struct set_node *representative)
{
    struct set_node *p = head;
    while (p != NULL) {
        p->representative = representative;
        p = p->next;
    }
}

//把较短的链表拼到较长的链表上，更新短链表的每个结点指向代表指针
void set_union(set sa, set sb)
{
    if (sa->num < sb->num) {
        update_representative(sa->head, sb->head);
        sb->tail->next = sa->head;
        sa->head = sb->head;
    } else {
        update_representative(sb->head, sa->head);
        sa->tail->next = sb->head;
        sa->tail = sb->tail;
    }
    sa->num += sb->num;
}
```

```
void set_destroy(set s, void (*free_key) (void *))
{
	free_key(s->head->key);
	free(s->head);
	free(s);
}

struct edge {
	char u;
	char v;
};
int main()
{
	//数据根据书上图 21-1
	char vertex[] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' };
	set s[256] = { NULL };
	struct edge edge_array[] = { {'b', 'd'}, {'e', 'g'}, {'a', 'c'},
	{'h', 'i'}, {'a', 'b'}, {'e', 'f'}, {'b', 'c'}
	};
	int vertex_num = sizeof(vertex) / sizeof(vertex[0]);
	for (int i = 0; i < vertex_num; i++) {
		char *c = malloc(sizeof(char));
		*c = vertex[i];
		s[(int)vertex[i]] = set_create(c);
	}
	//计算连通子图
	for (unsigned i = 0; i < sizeof(edge_array) / sizeof(edge_array[0]);
	     i++) {
		set su = s[(int)edge_array[i].u];
		set sv = s[(int)edge_array[i].v];
		if (find_set(su) != find_set(sv)) {
			set_union(su, sv);
		}
	}
	//输出连通子图
	char str_set[256][256] = { {0} };
	for (int i = 0; i < vertex_num; i++) {
		char *pc = find_set(s[(int)vertex[i]])->key;
		int len = strlen(str_set[(int)*pc]);
		str_set[(int)*pc][len] = vertex[i];
	}
	printf("输出不相交集合组:\n");
	for (int i = 0; i < vertex_num; i++) {
```

```
			if (strcmp(str_set[(int)vertex[i]], "") != 0) {
				printf("%s\n", str_set[(int)vertex[i]]);
			}
	}
	for (int i = 0; i < vertex_num; i++) {
		set sv = s[(int)vertex[i]];
		set_destroy(sv, free);
	}
	return 0;
}
```

21.3 不相交集合森林

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct set_type *set;
struct set_node {
	void *key;
	int rank;
	struct set_node *parent;
};

void set_node_ini(struct set_node *x, void *key)
{
	x->key = key;
	x->rank = 0;
	x->parent = NULL;
}

struct set_type {
	struct set_node *root;
};

set set_create(void *key)
{
	set s = malloc(sizeof(struct set_type));
	s->root = malloc(sizeof(struct set_node));
	set_node_ini(s->root, key);
	s->root->parent = s->root;
	s->root->rank = 0;
	return s;
}
```

```
void link(struct set_node *x, struct set_node *y)
{
	if (x->rank > y->rank) {
		y->parent = x;
	} else {
		x->parent = y;
		if (x->rank == y->rank) {
			++y->rank;
		}
	}
}

struct set_node *find_set_path_compression(struct set_node *x)
{
	if (x != x->parent) {
		x->parent = find_set_path_compression(x->parent);
	}
	return x->parent;
}

struct set_node *find_set(set s)
{
	return find_set_path_compression(s->root);
}

void set_destroy(set s, void (*free_key) (void *))
{
	free_key(s->root->key);
	free(s->root);
	free(s);
}

void set_union(set sa, set sb)
{
	link(find_set(sa), find_set(sb));
}

struct edge {
	char u;
	char v;
};
int main()
{
```

```
    //数据根据书上图 21-1
    char vertex[] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' };
    set s[256] = { NULL };
    struct edge edge_array[] = { {'b', 'd'}, {'e', 'g'}, {'a', 'c'},
    {'h', 'i'}, {'a', 'b'}, {'e', 'f'}, {'b', 'c'}
    };
    int vertex_num = sizeof(vertex) / sizeof(vertex[0]);
    for (int i = 0; i < vertex_num; i++) {
        char *c = malloc(sizeof(char));
        *c = vertex[i];
        s[(int)vertex[i]] = set_create(c);
    }
    //计算连通子图
    for (unsigned i = 0; i < sizeof(edge_array) / sizeof(edge_array[0]);
         i++) {
        set su = s[(int)edge_array[i].u];
        set sv = s[(int)edge_array[i].v];
        if (find_set(su) != find_set(sv)) {
            set_union(su, sv);
        }
    }
    //输出连通子图
    char str_set[256][256] = { {0} };
    for (int i = 0; i < vertex_num; i++) {
        char *pc = find_set(s[(int)vertex[i]])->key;
        int len = strlen(str_set[(int)*pc]);
        str_set[(int)*pc][len] = vertex[i];
    }
    printf("输出不相交集合组:\n");
    for (int i = 0; i < vertex_num; i++) {
        if (strcmp(str_set[(int)vertex[i]], "") != 0) {
            printf("%s\n", str_set[(int)vertex[i]]);
        }
    }
    for (int i = 0; i < vertex_num; i++) {
        set sv = s[(int)vertex[i]];
        set_destroy(sv, free);
    }
    return 0;
}
```

第 22 章 图的基本算法

22.1 图的表示

22.1.1 邻接表表示法

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct graph_type *graph;
struct edge {
	int u;
	int v;
};
struct graph_node {
	int key;
	struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key)
{
	x->key = key;
	x->next = NULL;
}

struct vertex {
	char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
	strcpy(v->str_vertex, "");
}

struct graph_type {
	struct graph_node **adj;
	struct vertex *vertex_array;
	int v_num;
	int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
	graph g = malloc(sizeof(struct graph_type));
```

```
    g->v_num = v_num;
    g->e_num = 0;
    g->adj = malloc(sizeof(struct graph_node *) * v_num);
    g->vertex_array = malloc(sizeof(struct vertex) * v_num);
    for (int i = 0; i < v_num; i++) {
        g->adj[i] = NULL;
        strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
    }
    return g;
}

void graph_destroy(graph g)
{
    for (int i = 0; i < g->v_num; i++) {
        for (struct graph_node * x = g->adj[i]; x != NULL;) {
            struct graph_node *del=x;
            x=x->next;
            free(del);
        }
    }
    free(g->adj);
    free(g->vertex_array);
    free(g);
}

void graph_insert_edge(graph g, struct edge edge)
{
    struct graph_node *u = malloc(sizeof(struct graph_node));
    graph_node_ini(u, edge.u);
    struct graph_node *v = malloc(sizeof(struct graph_node));
    graph_node_ini(v, edge.v);
    //从表头插入,将 v 插入到表头 u
    v->next = g->adj[edge.u];
    g->adj[edge.u] = v;
    //从表头插入,将 u 插入到表头 v
    u->next = g->adj[edge.v];
    g->adj[edge.v] = u;
    ++g->e_num;
}

void graph_display(graph g)
{
    printf("%d vertices,%d edges\n", g->v_num, g->e_num);
    for (int i = 0; i < g->v_num; i++) {
```

```
        printf("%s: ", g->vertex_array[i].str_vertex);
        for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
            printf("%s ", g->vertex_array[x->key].str_vertex);
        }
        printf("\n");
    }
}

int main()
{
    //数据根据书上的图 22-1
    char *str_vertex[5] = { "1", "2", "3", "4", "5" };
    graph g = graph_create(5, str_vertex);
    struct edge edges[] =
         { {0, 1}, {0, 4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {3, 4} };
    for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
        graph_insert_edge(g, edges[i]);
    }
    graph_display(g);
    graph_destroy(g);
    return 0;
}
```

21.1.2 邻接矩阵表示法

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct graph_type *graph;
struct edge {
    int u;
    int v;
};
struct vertex {
    char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
    strcpy(v->str_vertex, "");
}

struct graph_type {
    int **adj;
```

```
	struct vertex *vertex_array;
	int v_num;
	int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
	graph g = malloc(sizeof(struct graph_type));
	g->v_num = v_num;
	g->e_num = 0;
	g->adj = malloc(sizeof(int *) * v_num);
	 for (int i = 0; i < v_num; i++) {
		g->adj[i] = malloc(sizeof(int) * v_num);
		 for (int j = 0; j < v_num; j++) {
			g->adj[i][j] = 0;
		}
	}
	g->vertex_array = malloc(sizeof(struct vertex) * v_num);
	for (int i = 0; i < v_num; i++) {
		strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
	}
	return g;
}

void graph_destroy(graph g)
{
	for (int i = 0; i < g->v_num; i++) {
		free(g->adj[i]);
	}
	free(g->adj);
	free(g->vertex_array);
	free(g);
}

void graph_insert_edge(graph g, struct edge edge)
{
	g->adj[edge.u][edge.v] = 1;
	g->adj[edge.v][edge.u] = 1;
	++g->e_num;
}

void graph_display(graph g)
{
	printf("%d vertices,%d edges\n", g->v_num, g->e_num);
```

```
	for (int i = 0; i < g->v_num; i++) {
		printf("%s: ", g->vertex_array[i].str_vertex);
		for (int j = 0; j < g->v_num; j++) {
			if (g->adj[i][j] != 0) {
				printf("%s ", g->vertex_array[j].str_vertex);
			}
		}
		printf("\n");
	}
}

int main()
{
	//数据根据书上的图 22-1
	char *str_vertex[5] = { "1", "2", "3", "4", "5" };
	graph g = graph_create(5, str_vertex);
	struct edge edges[] =
		{ {0, 1}, {0, 4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {3, 4} };
	for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
		graph_insert_edge(g, edges[i]);
	}
	graph_display(g);
	graph_destroy(g);
	return 0;
}
```

22.2 广度优先搜索

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
typedef struct queue_type *queue;
struct queue_node {
	void *key;
	struct queue_node *next;
};

struct queue_type {
	struct queue_node *head;
	struct queue_node *tail;
};
```

```
void queue_node_ini(struct queue_node *node, void *key)
{
	node->key = key;
	node->next = NULL;
}

queue queue_create()
{
	queue q = malloc(sizeof(struct queue_type));
	q->head = NULL;
	q->tail = NULL;
	return q;
}

bool queue_is_empty(queue q)
{
	return q->head == NULL;
}

void queue_en_queue(queue q, void *x)
{
	struct queue_node *p = malloc(sizeof(struct queue_node));
	queue_node_ini(p, x);
	if (q->head == NULL) {
		q->head = p;
		q->tail = p;
	} else {
		q->tail->next = p;
		q->tail = p;
	}
}

void *queue_de_queue(queue q)
{
	void *key = q->head->key;
	struct queue_node *p = q->head;
	q->head = q->head->next;
	free(p);
	return key;
}

void queue_destroy(queue q, void (*free_key) (void *))
{
```

```
	while (!queue_is_empty(q)) {
		void *p = queue_de_queue(q);
		free_key(p);
	}
	free(q);
}

enum color_enum {
	color_white,
	color_gray,
	color_black
};
typedef struct graph_type *graph;
struct edge {
	int u;
	int v;
};
struct graph_node {
	int key;
	struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key)
{
	x->key = key;
	x->next = NULL;
}

struct vertex {
	enum color_enum color;
	int dis;
	int parent;
	char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
	v->color = color_white;
	v->dis = 0;
	v->parent = -1;		//顶点编号是从 0 开始，-1 表示一个不存在的结点
	strcpy(v->str_vertex, "");
}

struct graph_type {
	struct graph_node **adj;
	struct vertex *vertex_array;
```

```
    int v_num;
    int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
    graph g = malloc(sizeof(struct graph_type));
    g->v_num = v_num;
    g->e_num = 0;
    g->adj = malloc(sizeof(struct graph_node *) * v_num);
    g->vertex_array = malloc(sizeof(struct vertex) * v_num);
    for (int i = 0; i < v_num; i++) {
        g->adj[i] = NULL;
        strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
    }
    return g;
}

void graph_destroy(graph g)
{
    for (int i = 0; i < g->v_num; i++) {
        for (struct graph_node * x = g->adj[i]; x != NULL;) {
            struct graph_node *del=x;
            x=x->next;
            free(del);
        }
    }
    free(g->adj);
    free(g->vertex_array);
    free(g);
}

void graph_insert_edge(graph g, struct edge edge)
{
    struct graph_node *u = malloc(sizeof(struct graph_node));
    graph_node_ini(u, edge.u);
    struct graph_node *v = malloc(sizeof(struct graph_node));
    graph_node_ini(v, edge.v);
    //从表头插入,将 v 插入到表头 u
    v->next = g->adj[edge.u];
    g->adj[edge.u] = v;
    //从表头插入,将 u 插入到表头 v
    u->next = g->adj[edge.v];
    g->adj[edge.v] = u;
```

```
    ++g->e_num;
}

void graph_display(graph g)
{
    printf("%d vertices,%d edges\n", g->v_num, g->e_num);
    for (int i = 0; i < g->v_num; i++) {
        printf("%s: ", g->vertex_array[i].str_vertex);
        for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
            printf("%s ", g->vertex_array[x->key].str_vertex);
        }
        printf("\n");
    }
}

void graph_display_vertex(graph g)
{
    printf("各个顶点的数据:\n");
    for (int i = 0; i < g->v_num; i++) {
        printf("%s:%d\n", g->vertex_array[i].str_vertex,
                g->vertex_array[i].dis);
    }
}

void graph_bfs(graph g, int s)
{
    for (int i = 0; i < g->v_num; i++) {
        if (i != s) {
            g->vertex_array[i].color = color_white;
            g->vertex_array[i].dis = INT_MAX;
            g->vertex_array[i].parent = -1;
        }
    }
    g->vertex_array[s].color = color_gray;
    g->vertex_array[s].dis = 0;
    g->vertex_array[s].parent = -1;
    queue q = queue_create();
    int *p = malloc(sizeof(int));
    *p = s;
    queue_en_queue(q, p);
    while (!queue_is_empty(q)) {
        p = queue_de_queue(q);
        int u = *p;
        free(p);
```

```
        for (struct graph_node * x = g->adj[u]; x != NULL; x = x->next) {
            int v = x->key;
            if (g->vertex_array[v].color == color_white) {
                g->vertex_array[v].color = color_gray;
                g->vertex_array[v].dis =
                    g->vertex_array[u].dis + 1;
                g->vertex_array[v].parent = u;
                p = malloc(sizeof(int));
                *p = v;
                queue_en_queue(q, p);
            }
        }
        g->vertex_array[u].color = color_black;
    }
    queue_destroy(q, free);
}

void graph_print_path(graph g, int s, int v)
{
    if (v == s) {
        printf("%s ", g->vertex_array[s].str_vertex);
    } else {
        if (g->vertex_array[v].parent == -1) {
            printf("no path from %s to %s exist\n",
                    g->vertex_array[s].str_vertex,
                    g->vertex_array[v].str_vertex);
        } else {
            graph_print_path(g, s, g->vertex_array[v].parent);
            printf("%s ", g->vertex_array[v].str_vertex);
        }
    }
}

int main()
{
    //数据根据书上的图 22-3
    char *str_vertex[8] = {
        "r", "s", "t", "u", "v", "w", "x", "y"
    };
    graph g = graph_create(8, str_vertex);
    struct edge edges[] = {
        {0, 1}, {0, 4}, {1, 5}, {2, 3}, {2, 5}, {2, 6}, {3, 6}, {3, 7},
        {5, 6}, {6, 7}
    };
```

```
	for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
		graph_insert_edge(g, edges[i]);
	}
	graph_display(g);
	int s = 1;
	int v = 7;
	graph_bfs(g, s);
	graph_display_vertex(g);
	printf("path from %s to %s\n", str_vertex[s], str_vertex[v]);
	graph_print_path(g, s, v);
	printf("\n");
	graph_destroy(g);
}
```

22.3 深度优先搜索

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef struct graph_type *graph;
enum color_enum {
	color_white,
	color_gray,
	color_black
};
struct edge {
	int u;
	int v;
};
struct graph_node {
	int key;
	struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key)
{
	x->key = key;
	x->next = NULL;
}

struct vertex {
	enum color_enum color;
	int parent;
	int discovered_time;
```

```
    int finish_time;
    char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
    v->color = color_white;
    v->parent = -1;         //顶点编号是从 0 开始，-1 表示一个不存在的结点
    v->discovered_time = 0;
    v->finish_time = 0;
    strcpy(v->str_vertex, "");
}

struct graph_type {
    struct graph_node **adj;
    int time;
    struct vertex *vertex_array;
    int v_num;
    int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
    graph g = malloc(sizeof(struct graph_type));
    g->v_num = v_num;
    g->e_num = 0;
    g->time = 0;
    g->adj = malloc(sizeof(struct graph_node *) * v_num);
    g->vertex_array = malloc(sizeof(struct vertex) * v_num);
    for (int i = 0; i < v_num; i++) {
        g->adj[i] = NULL;
        strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
    }
    return g;
}

void graph_destroy(graph g)
{
    for (int i = 0; i < g->v_num; i++) {
        for (struct graph_node * x = g->adj[i]; x != NULL;) {
            struct graph_node *del=x;
            x=x->next;
            free(del);
        }
    }
```

```
    free(g->adj);
    free(g->vertex_array);
    free(g);
}

void graph_insert_edge(graph g, struct edge edge)
{
    struct graph_node *v = malloc(sizeof(struct graph_node));
    graph_node_ini(v, edge.v);
    //从表头插入,将 v 插入到表头 u
    v->next = g->adj[edge.u];
    g->adj[edge.u] = v;
    ++g->e_num;
}

void graph_display(graph g)
{
    printf("%d vertices,%d edges\n", g->v_num, g->e_num);
    for (int i = 0; i < g->v_num; i++) {
        printf("%s: ", g->vertex_array[i].str_vertex);
        for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
            printf("%s ", g->vertex_array[x->key].str_vertex);
        }
        printf("\n");
    }
}

void graph_dfs_visit(graph g, int u)
{
    g->vertex_array[u].color = color_gray;
    g->vertex_array[u].discovered_time = ++g->time;
    for (struct graph_node * x = g->adj[u]; x != NULL; x = x->next) {
        int v = x->key;
        if (g->vertex_array[v].color == color_white) {
            g->vertex_array[v].parent = u;
            graph_dfs_visit(g,v);
        }
    }
    g->vertex_array[u].color = color_black;
    g->vertex_array[u].finish_time = ++g->time;
}

void graph_depth_first_search(graph g)
{
```

```
	for (int i = 0; i < g->v_num; i++) {
		g->vertex_array[i].color = color_white;
		g->vertex_array[i].parent = -1;
	}
	for (int i = 0; i < g->v_num; i++) {
		if (g->vertex_array[i].color == color_white) {
			graph_dfs_visit(g,i);
		}
	}
}

void graph_display_vertex(graph g)
{
	printf("各个顶点的数据:\n");
	for (int i = 0; i < g->v_num; i++) {
		printf("%s: %d/%d\n", g->vertex_array[i].str_vertex,
				g->vertex_array[i].discovered_time,
				g->vertex_array[i].finish_time);
	}
}

int main()
{
	//数据根据书上的图 22-4
	char *str_vertex[6] = { "u", "v", "w", "x", "y", "z" };
	graph g=graph_create(6, str_vertex);
	struct edge edges[] =
		{ {0, 3}, {0, 1}, {1, 4}, {2, 4}, {2, 5}, {3, 1}, {4, 3}, {5, 5} };
	for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
		graph_insert_edge(g,edges[i]);
	}
	graph_display(g);
	graph_depth_first_search(g);
	graph_display_vertex(g);
	graph_destroy(g);
	return 0;
}
```

22.4 拓扑排序

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```c
#include <stdbool.h>
typedef struct stack_type *stack;
struct stack_node {
	void *key;
	struct stack_node *next;
};
struct stack_type {
	struct stack_node *head;
};
void stack_node_ini(struct stack_node *n, void *key)
{
	n->key = key;
	n->next = NULL;
}

stack stack_create()
{
	stack s = malloc(sizeof(struct stack_type));
	s->head = NULL;
	return s;
}

bool stack_is_empty(stack s)
{
	return s->head == NULL;
}

void stack_push(stack s, void *x)
{
	struct stack_node *node = malloc(sizeof(struct stack_node));
	stack_node_ini(node, x);
	node->next = s->head;
	s->head = node;
}

void *stack_pop(stack s)
{
	struct stack_node *p = s->head;
	s->head = s->head->next;
	void *key = p->key;
	free(p);
	return key;
}
```

```
void stack_destroy(stack s, void (*free_key) (void *))
{
	while (!stack_is_empty(s)) {
		void *p = stack_pop(s);
		free_key(p);
	}
	free(s);
}

typedef struct graph_type *graph;
enum color_enum {
	color_white,
	color_gray,
	color_black
};
struct edge {
	int u;
	int v;
};
struct graph_node {
	int key;
	struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key)
{
	x->key = key;
	x->next = NULL;
}

struct vertex {
	enum color_enum color;
	int parent;
	int discovered_time;
	int finish_time;
	char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
	v->color = color_white;
	v->parent = -1;		//顶点编号是从 0 开始，-1 表示一个不存在的结点
	v->discovered_time = 0;
	v->finish_time = 0;
	strcpy(v->str_vertex, "");
}
```

```c
struct graph_type {
    struct graph_node **adj;
    int time;
    struct vertex *vertex_array;
    int v_num;
    int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
    graph g = malloc(sizeof(struct graph_type));
    g->v_num = v_num;
    g->e_num = 0;
    g->time = 0;
    g->adj = malloc(sizeof(struct graph_node *) * v_num);
    g->vertex_array = malloc(sizeof(struct vertex) * v_num);
    for (int i = 0; i < v_num; i++) {
        g->adj[i] = NULL;
        strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
    }
    return g;
}

void graph_destroy(graph g)
{
    for (int i = 0; i < g->v_num; i++) {
        for (struct graph_node * x = g->adj[i]; x != NULL;) {
            struct graph_node *del=x;
            x=x->next;
            free(del);
        }
    }
    free(g->adj);
    free(g->vertex_array);
    free(g);
}

void graph_insert_edge(graph g, struct edge edge)
{
    struct graph_node *v = malloc(sizeof(struct graph_node));
    graph_node_ini(v, edge.v);
    //从表头插入,将 v 插入到表头 u
    v->next = g->adj[edge.u];
```

```
    g->adj[edge.u] = v;
    ++g->e_num;
}

void graph_display(graph g)
{
    printf("%d vertices,%d edges\n", g->v_num, g->e_num);
    for (int i = 0; i < g->v_num; i++) {
        printf("%s: ", g->vertex_array[i].str_vertex);
        for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
            printf("%s ", g->vertex_array[x->key].str_vertex);
        }
        printf("\n");
    }
}

void graph_display_vertex(graph g)
{
    printf("各个顶点的数据:\n");
    for (int i = 0; i < g->v_num; i++) {
        printf("%s: %d/%d\n", g->vertex_array[i].str_vertex,
                g->vertex_array[i].discovered_time,
                g->vertex_array[i].finish_time);
    }
}

void graph_dfs_visit(graph g, int u, stack s)
{
    g->vertex_array[u].color = color_gray;
    g->vertex_array[u].discovered_time = ++g->time;
    for (struct graph_node * x = g->adj[u]; x != NULL; x = x->next) {
        int v = x->key;
        if (g->vertex_array[v].color == color_white) {
            g->vertex_array[v].parent = u;
            graph_dfs_visit(g, v, s);
        }
    }
    g->vertex_array[u].color = color_black;
    g->vertex_array[u].finish_time = ++g->time;
    int *p=malloc(sizeof(int));
    *p=u;
    stack_push(s, p);
}
```

```c
//拓扑排序，把结果放到堆栈 s,修改自 depth_first_search 函数
void graph_topological_sort(graph g,stack s)
{
	for (int i = 0; i < g->v_num; i++) {
		g->vertex_array[i].color = color_white;
		g->vertex_array[i].parent = -1;
	}
	for (int i = 0; i < g->v_num; i++) {
		if (g->vertex_array[i].color == color_white) {
			graph_dfs_visit(g,i, s);
		}
	}
}

int main()
{
	//数据根据书上的图 22-7
	char *str_vertex[9] =
		 { "shirt", "tie", "jacket", "belt", "watch", "undershorts", "pants",
		"shoes", "socks"
	};
	graph g=graph_create(9, str_vertex);
	struct edge edges[] =
		 { {0, 3}, {0, 1}, {1, 2}, {3, 2}, {5, 7}, {5, 6}, {6, 7}, {6, 3},
	{8, 7}
	};
	for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
		graph_insert_edge(g,edges[i]);
	}
	graph_display(g);
	stack s=stack_create();
	graph_topological_sort(g,s);
	graph_display_vertex(g);
	while (!stack_is_empty(s)) {
		int *p=stack_pop(s);
		printf("%s ",str_vertex[*p]);
		free(p);
	}
	printf("\n");
	stack_destroy(s,free);
	graph_destroy(g);
	return 0;
}
```

22.5 强连通分支

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct stack_type *stack;
struct stack_node {
	void *key;
	struct stack_node *next;
};
struct stack_type {
	struct stack_node *head;
};
void stack_node_ini(struct stack_node *n, void *key)
{
	n->key = key;
	n->next = NULL;
}

stack stack_create()
{
	stack s = malloc(sizeof(struct stack_type));
	s->head = NULL;
	return s;
}

bool stack_is_empty(stack s)
{
	return s->head == NULL;
}

void stack_push(stack s, void *x)
{
	struct stack_node *node = malloc(sizeof(struct stack_node));
	stack_node_ini(node, x);
	node->next = s->head;
	s->head = node;
}

void *stack_pop(stack s)
{
	struct stack_node *p = s->head;
```

```
    s->head = s->head->next;
    void *key = p->key;
    free(p);
    return key;
}

void stack_destroy(stack s, void (*free_key) (void *))
{
    while (!stack_is_empty(s)) {
        void *p = stack_pop(s);
        free_key(p);
    }
    free(s);
}

typedef struct graph_type *graph;
enum color_enum {
    color_white,
    color_gray,
    color_black
};
struct edge {
    int u;
    int v;
};
struct graph_node {
    int key;
    struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key)
{
    x->key = key;
    x->next = NULL;
}

struct vertex {
    enum color_enum color;
    int parent;
    int discovered_time;
    int finish_time;
    char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
```

```
    v->color = color_white;
    v->parent = -1;        //顶点编号是从 0 开始，-1 表示一个不存在的结点
    v->discovered_time = 0;
    v->finish_time = 0;
    strcpy(v->str_vertex, "");
}

struct graph_type {
    struct graph_node **adj;
    int time;
    struct vertex *vertex_array;
    int v_num;
    int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
    graph g = malloc(sizeof(struct graph_type));
    g->v_num = v_num;
    g->e_num = 0;
    g->time = 0;
    g->adj = malloc(sizeof(struct graph_node *) * v_num);
    g->vertex_array = malloc(sizeof(struct vertex) * v_num);
    for (int i = 0; i < v_num; i++) {
        g->adj[i] = NULL;
        strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
    }
    return g;
}

void graph_destroy(graph g)
{
    for (int i = 0; i < g->v_num; i++) {
        for (struct graph_node * x = g->adj[i]; x != NULL;) {
            struct graph_node *del=x;
            x=x->next;
            free(del);
        }
    }
    free(g->adj);
    free(g->vertex_array);
    free(g);
}
```

```
void graph_insert_edge(graph g, struct edge edge)
{
	struct graph_node *v = malloc(sizeof(struct graph_node));
	graph_node_ini(v, edge.v);
	//从表头插入,将 v 插入到表头 u
	v->next = g->adj[edge.u];
	g->adj[edge.u] = v;
	++g->e_num;
}

void graph_display(graph g)
{
	printf("%d vertices,%d edges\n", g->v_num, g->e_num);
	for (int i = 0; i < g->v_num; i++) {
		printf("%s: ", g->vertex_array[i].str_vertex);
		for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
			printf("%s ", g->vertex_array[x->key].str_vertex);
		}
		printf("\n");
	}
}

void graph_dfs_visit(graph g, int u, stack s)
{
	g->vertex_array[u].color = color_gray;
	g->vertex_array[u].discovered_time = ++g->time;
	for (struct graph_node * x = g->adj[u]; x != NULL; x = x->next) {
		int v = x->key;
		if (g->vertex_array[v].color == color_white) {
			g->vertex_array[v].parent = u;
			graph_dfs_visit(g, v, s);
		}
	}
	g->vertex_array[u].color = color_black;
	g->vertex_array[u].finish_time = ++g->time;
	int *p = malloc(sizeof(int));
	*p = u;
	stack_push(s, p);
}

//拓扑排序，把结果放到堆栈 s,修改自 depth_first_search 函数
void graph_topological_sort(graph g, stack s)
{
	for (int i = 0; i < g->v_num; i++) {
```

```
        g->vertex_array[i].color = color_white;
        g->vertex_array[i].parent = -1;
    }
    for (int i = 0; i < g->v_num; i++) {
        if (g->vertex_array[i].color == color_white) {
            graph_dfs_visit(g, i, s);
        }
    }
}

void graph_reverse_graph(graph g, graph gr)
{
    for (int i = 0; i < g->v_num; i++) {
        int u = i;
        for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
            int v = x->key;
            struct edge e = { v, u };
            graph_insert_edge(gr, e);
        }
    }
}

void strongly_connected_components(graph g, char *str_vertex[])
{
    stack s = stack_create();
    //第二次深度优先搜索是按拓扑排序的顺序来访问顶点,所以第一次深度优先
搜索改成求拓扑排序
    graph_topological_sort(g, s);
    graph gr = graph_create(g->v_num, str_vertex);
    graph_reverse_graph(g, gr);    //根据原图构造转置图
    stack sr = stack_create();
    for (int i = 0; i < g->v_num; i++) {
        gr->vertex_array[i].color = color_white;
        gr->vertex_array[i].parent = -1;
    }
    printf("图的强连通分支如下:\n");
    int i = 0;
    while (!stack_is_empty(s)) {
        int *p = stack_pop(s);
        int u = *p;
        free(p);
        if (gr->vertex_array[u].color == color_white) {
            graph_dfs_visit(gr, u, sr); //sr 记录了一个连通分支的所有结点
            printf("第%d 个连通分支:\n", i + 1);
```

```
			while (!stack_is_empty(sr)) {
				int *p = stack_pop(sr);
				printf("%s ", gr->vertex_array[*p].str_vertex);
				free(p);
			}
			printf("\n");
			++i;
		}
	}
	stack_destroy(s,free);
	stack_destroy(sr,free);
	graph_destroy(gr);
}

int main()
{
	//数据根据书上的图 22-9
	char *str_vertex[8] = { "c", "d", "h", "g", "b", "f", "e", "a" };
	graph g = graph_create(8, str_vertex);
	struct edge edges[] = {
		{0, 1}, {0, 3}, {1, 0}, {1, 2}, {2, 2}, {3, 2}, {3, 5}, {4, 0},
		{4, 5}, {4, 6}, {5, 3}, {6, 5}, {6, 7}, {7, 4}
	};
	for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
		graph_insert_edge(g, edges[i]);
	}
	graph_display(g);
	strongly_connected_components(g, str_vertex);
	graph_destroy(g);
}
```

第 23 章 最小生成树

22.2 Kruskal 算法和 Prim 算法

22.2.1 Kruskal 算法

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

```
#include <string.h>
typedef struct graph_type *graph;
struct edge {
	int u;
	int v;
	int w;
};
struct graph_node {
	int key;
	int w;
	struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key, int w)
{
	x->key = key;
	x->w = w;
	x->next = NULL;
}

struct vertex {
	char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
	strcpy(v->str_vertex, "");
}

struct graph_type {
	struct graph_node **adj;
	struct vertex *vertex_array;
	int v_num;
	int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
	graph g = malloc(sizeof(struct graph_type));
	g->v_num = v_num;
	g->e_num = 0;
	g->adj = malloc(sizeof(struct graph_node *) * v_num);
	g->vertex_array = malloc(sizeof(struct vertex) * v_num);
	for (int i = 0; i < v_num; i++) {
		g->adj[i] = NULL;
		strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
```

```
    }
    return g;
}

void graph_destroy(graph g)
{
    for (int i = 0; i < g->v_num; i++) {
        for (struct graph_node * x = g->adj[i]; x != NULL;) {
            struct graph_node *del=x;
            x=x->next;
            free(del);
        }
    }
    free(g->adj);
    free(g->vertex_array);
    free(g);
}

void graph_insert_edge(graph g, struct edge e)
{
    struct graph_node *u = malloc(sizeof(struct graph_node));
    graph_node_ini(u, e.u, e.w);
    struct graph_node *v = malloc(sizeof(struct graph_node));
    graph_node_ini(v, e.v, e.w);
    //从表头插入,将 v 插入到表头 u
    v->next = g->adj[e.u];
    g->adj[e.u] = v;
    //从表头插入,将 u 插入到表头 v
    u->next = g->adj[e.v];
    g->adj[e.v] = u;
    ++g->e_num;
}

void graph_display(graph g)
{
    printf("%d vertices,%d edges\n", g->v_num, g->e_num);
    for (int i = 0; i < g->v_num; i++) {
        printf("%s: ", g->vertex_array[i].str_vertex);
        for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
            printf("%s,%d ", g->vertex_array[x->key].str_vertex,x->w);
        }
        printf("\n");
    }
}
```

```
void swap(void *a, void *b, size_t elem_size)
{
	if (a == NULL || b == NULL || a == b)
		return;
	char temp[elem_size];	/*变长数组 */
	memcpy(temp, a, elem_size);
	memcpy(a, b, elem_size);
	memcpy(b, temp, elem_size);
}

int partition(void *base, size_t elem_size, int p, int r,
		int (*comp) (const void *, const void *))
{
	char *cbase = base;
	void *key = &cbase[r * elem_size];
	int i = p - 1;
	for (int j = p; j < r; j++) {
		if (comp(&cbase[j * elem_size], key) <= 0) {
			++i;
			swap(&cbase[i * elem_size], &cbase[j * elem_size],
				elem_size);
		}
	}
	swap(&cbase[(i + 1) * elem_size], key, elem_size);
	return i + 1;
}

void quick_sort(void *base, size_t elem_size, int p, int r,
		int (*comp) (const void *, const void *))
{
	if (p < r) {
		int q = partition(base, elem_size, p, r, comp);
		quick_sort(base, elem_size, p, q - 1, comp);
		quick_sort(base, elem_size, q + 1, r, comp);
	}
}

typedef struct set_type *set;
struct set_node {
	void *key;
	int rank;
	struct set_node *parent;
};
```

```
void set_node_ini(struct set_node *x, void *key)
{
	x->key = key;
	x->rank = 0;
	x->parent = NULL;
}

struct set_type {
	struct set_node *root;
};

set set_create(void *key)
{
	set s = malloc(sizeof(struct set_type));
	s->root = malloc(sizeof(struct set_node));
	set_node_ini(s->root, key);
	s->root->parent = s->root;
	s->root->rank = 0;
	return s;
}

void link(struct set_node *x, struct set_node *y)
{
	if (x->rank > y->rank) {
		y->parent = x;
	} else {
		x->parent = y;
		if (x->rank == y->rank) {
			++y->rank;
		}
	}
}

struct set_node *find_set_path_compression(struct set_node *x)
{
	if (x != x->parent) {
		x->parent = find_set_path_compression(x->parent);
	}
	return x->parent;
}

struct set_node *find_set(set s)
{
```

```
    return find_set_path_compression(s->root);
}

void set_destroy(set s, void (*free_key) (void *))
{
    free_key(s->root->key);
    free(s->root);
    free(s);
}

void set_union(set sa, set sb)
{
    link(find_set(sa), find_set(sb));
}

void graph_get_edges(graph g, struct edge edges[], int *edge_num)
{
    *edge_num = 0;
    for (int i = 0; i < g->v_num; i++) {
        int u = i;
        for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
            int v = x->key;
            if (u <= v) {
                struct edge edge = { u, v, x->w };
                edges[(*edge_num)++] = edge;
            }
        }
    }
}

int cmp_edge(const void *p1, const void *p2)
{
    const struct edge *pa = p1;
    const struct edge *pb = p2;
    if (pa->w < pb->w)
        return -1;
    if (pa->w == pb->w)
        return 0;
    return 1;
}

void graph_mst_kruskal(graph g, struct edge tree_edges[], int *tree_edge_num)
{
    set set_array[g->v_num];
```

```
	for (int i = 0; i < g->v_num; i++) {
		int *p = malloc(sizeof(int));
		*p = i;
		set_array[i] = set_create(p);
	}
	struct edge edges[g->e_num];
	int edge_num = 0;
	graph_get_edges(g,edges, &edge_num);
	quick_sort(edges, sizeof(struct edge), 0, edge_num - 1, cmp_edge);
	*tree_edge_num = 0;
	for (int i = 0; i < edge_num; i++) {
		struct edge edge = edges[i];
		if (find_set(set_array[edge.u]) !=
		    find_set(set_array[edge.v])) {
			tree_edges[(*tree_edge_num)++] = edge;
			set_union(set_array[edge.u], set_array[edge.v]);
		}
	}
	for(int i=0;i<g->v_num;i++)
	{
		set_destroy(set_array[i],free);
	}
}

int main()
{
	//数据根据书上的图 23-1
	char *str_vertex[9] = { "a", "b", "c", "d", "e", "f", "g", "h", "i" };
	graph g = graph_create(9, str_vertex);
	struct edge edges[] = {
		{0, 1, 4}, {0, 7, 8}, {1, 7, 11},
		{1, 2, 8}, {2, 8, 2}, {2, 5, 4}, {2, 3, 7},
		{3, 4, 9}, {3, 5, 14}, {4, 5, 10}, {5, 6, 2},
		{6, 7, 1}, {6, 8, 6}, {7, 8, 7}
	};
	for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
		graph_insert_edge(g,edges[i]);
	}
	printf("图信息:\n");
	graph_display(g);
	struct edge tree_edges[sizeof(edges) / sizeof(edges[0])];
	int edge_tree_num;
	printf("最小生成树的边集是:\n");
	graph_mst_kruskal(g,tree_edges, &edge_tree_num);
```

```
    int weight_sum = 0;
    for (int i = 0; i < edge_tree_num; i++) {
        struct edge e = tree_edges[i];
        weight_sum += e.w;
        printf("%s %s %d\n", str_vertex[e.u],str_vertex[e.v],e.w);
    }
    printf("最小生成树的权值之和是:%d\n",weight_sum);
    graph_destroy(g);
    return 0;
}
```

22.2.2 Prim 算法

22.2.2.1 Prim 算法,使用最小优先级队列实现

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
typedef struct graph_type *graph;
struct edge {
    int u;
    int v;
    int w;
};
struct graph_node {
    int key;
    int w;
    struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key, int w)
{
    x->key = key;
    x->w = w;
    x->next = NULL;
}

struct vertex {
    int dis;
    int parent;
    bool in_queue;          //是否在队列里面
```

```
	char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
	v->dis = INT_MAX;
	v->parent = -1;
	v->in_queue = false;
	strcpy(v->str_vertex, "");
}

struct graph_type {
	struct graph_node **adj;
	struct vertex *vertex_array;
	int v_num;
	int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
	graph g = malloc(sizeof(struct graph_type));
	g->v_num = v_num;
	g->e_num = 0;
	g->adj = malloc(sizeof(struct graph_node *) * v_num);
	g->vertex_array = malloc(sizeof(struct vertex) * v_num);
	for (int i = 0; i < v_num; i++) {
		g->adj[i] = NULL;
		strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
	}
	return g;
}

void graph_destroy(graph g)
{
	for (int i = 0; i < g->v_num; i++) {
		for (struct graph_node * x = g->adj[i]; x != NULL;) {
			struct graph_node *del = x;
			x = x->next;
			free(del);
		}
	}
	free(g->adj);
	free(g->vertex_array);
	free(g);
}
```

```
void graph_insert_edge(graph g, struct edge e)
{
    struct graph_node *u = malloc(sizeof(struct graph_node));
    graph_node_ini(u, e.u, e.w);
    struct graph_node *v = malloc(sizeof(struct graph_node));
    graph_node_ini(v, e.v, e.w);
    //从表头插入,将 v 插入到表头 u
    v->next = g->adj[e.u];
    g->adj[e.u] = v;
    //从表头插入,将 u 插入到表头 v
    u->next = g->adj[e.v];
    g->adj[e.v] = u;
    ++g->e_num;
}

void graph_display(graph g)
{
    printf("%d vertices,%d edges\n", g->v_num, g->e_num);
    for (int i = 0; i < g->v_num; i++) {
        printf("%s: ", g->vertex_array[i].str_vertex);
        for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
            printf("%s,%d ", g->vertex_array[x->key].str_vertex,
                    x->w);
        }
        printf("\n");
    }
}

void swap(void *a, void *b, size_t elem_size)
{
    if (a == NULL || b == NULL || a == b)
        return;
    char temp[elem_size];     /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}

/*基于索引堆的优先队列*/
typedef struct priority_queue_index_type *priority_queue;
struct priority_queue_index_type {
    int heap_size;
    int *index_array;
```

```
    int *index_pos_array;/*这个数组记录了索引在堆中位置 */
    void *data_array;
    size_t elem_size;
    int (*comp) (const void *, const void *);
};
int parent(int i)
{
    return (i - 1) / 2;
}

int left_child(int i)
{
    return i * 2 + 1;
}

int right_child(int i)
{
    return i * 2 + 2;
}

void swap_index(priority_queue pq, int i, int j)
{
    swap(&pq->index_pos_array[i], &pq->index_pos_array[j], sizeof(int));
    pq->index_array[pq->index_pos_array[i]] = i;
    pq->index_array[pq->index_pos_array[j]] = j;
}

/*最小堆用的比较函数*/
bool compare(priority_queue pq, int left, int right)
{
    if (pq->data_array == NULL)
        return false;
    char *pc_array = pq->data_array;
    return pq->comp(&pc_array[left * pq->elem_size],
            &pc_array[right * pq->elem_size]) > 0;
}

void heapify(priority_queue pq, int i)
{
    int left = left_child(i);
    int right = right_child(i);
    int largest = i;
    if (left < pq->heap_size
        && compare(pq, pq->index_array[largest], pq->index_array[left])) {
```

```
            largest = left;
        }
        if (right < pq->heap_size
             && compare(pq, pq->index_array[largest], pq->index_array[right])) {
            largest = right;
        }
        if (largest != i) {
            swap_index(pq, pq->index_array[i], pq->index_array[largest]);
            heapify(pq, largest);
        }
}

void fix_up(priority_queue pq, int i)
{
        while (i > 0
                    && compare(pq, pq->index_array[parent(i)], pq->index_array[i])) {
            swap_index(pq, pq->index_array[parent(i)], pq->index_array[i]);
            i = parent(i);
        }
}

priority_queue priority_queue_create(void *p_data_array, size_t elem_size,
                                     int length, int (*comp) (const void *,
                                                         const void *))
{
        priority_queue pq = malloc(sizeof(struct priority_queue_index_type));
        pq->index_array = malloc(sizeof(int) * length);
        pq->index_pos_array = malloc(sizeof(int) * length);
        pq->data_array = p_data_array;
        pq->elem_size = elem_size;
        pq->heap_size = 0;
        pq->comp = comp;
        return pq;
}

void priority_queue_destroy(priority_queue pq)
{
        free(pq->index_array);
        free(pq->index_pos_array);
        free(pq);
}

int priority_queue_top(priority_queue pq)
{
```

```
	return pq->index_array[0];
}

/*去掉并返回堆的第一个元素 */
int priority_queue_extract_top(priority_queue pq)
{
	swap_index(pq, pq->index_array[0], pq->index_array[pq->heap_size - 1]);
	--pq->heap_size;
	heapify(pq, 0);
	return pq->index_array[pq->heap_size];
}

/*把元素的索引插入队列 */
void priority_queue_insert(priority_queue pq, int index)
{
	++pq->heap_size;
	int i = pq->heap_size - 1;
	pq->index_array[i] = index;
	pq->index_pos_array[index] = i;
	fix_up(pq, i);
}

bool priority_queue_is_empty(priority_queue pq)
{
	return pq->heap_size == 0;
}

/*下标为 index 的数据修改了，调用这个函数来修复索引堆*/
void priority_queue_change_index(priority_queue pq, int index)
{
	fix_up(pq, pq->index_pos_array[index]);
	heapify(pq, pq->index_pos_array[index]);
}

int cmp_vertex(const void *p1, const void *p2)
{
	const struct vertex *pa = p1;
	const struct vertex *pb = p2;
	if (pa->dis < pb->dis)
		return -1;
	if (pa->dis == pb->dis)
		return 0;
	return 1;
}
```

```
void graph_mst_prim(graph g, int r, struct edge tree_edges[],
                int *tree_edge_num)
{
    priority_queue pq =
        priority_queue_create(g->vertex_array, sizeof(struct vertex),
                    g->v_num, cmp_vertex);
    for (int i = 0; i < g->v_num; i++) {
        g->vertex_array[i].dis = INT_MAX;
        g->vertex_array[i].parent = -1;
        g->vertex_array[i].in_queue = true;
        priority_queue_insert(pq, i);
    }
    g->vertex_array[r].dis = 0;
    priority_queue_change_index(pq, r);
    *tree_edge_num = 0;
    while (!priority_queue_is_empty(pq)) {
        int u = priority_queue_extract_top(pq);
        if (u != r) {
            struct edge edge = { g->vertex_array[u].parent, u,
                g->vertex_array[u].dis
            };
            tree_edges[(*tree_edge_num)++] = edge;
        }
        g->vertex_array[u].in_queue = false;    //表示已经出队
        for (struct graph_node * x = g->adj[u]; x != NULL; x = x->next) {
            int v = x->key;
            //在队列中
            if (g->vertex_array[v].in_queue
                 && x->w < g->vertex_array[v].dis) {
                g->vertex_array[v].parent = u;
                g->vertex_array[v].dis = x->w;
                priority_queue_change_index(pq, v);
            }
        }
    }
    priority_queue_destroy(pq);
}

int main()
{
    //数据根据书上的图 23-1
    char *str_vertex[9] = { "a", "b", "c", "d", "e", "f", "g", "h", "i" };
    graph g = graph_create(9, str_vertex);
```

```
	struct edge edges[] = {
		{0, 1, 4}, {0, 7, 8}, {1, 7, 11},
		{1, 2, 8}, {2, 8, 2}, {2, 5, 4}, {2, 3, 7},
		{3, 4, 9}, {3, 5, 14}, {4, 5, 10}, {5, 6, 2},
		{6, 7, 1}, {6, 8, 6}, {7, 8, 7}
	};
	for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
		graph_insert_edge(g, edges[i]);
	}
	printf("图信息:\n");
	graph_display(g);
	struct edge tree_edges[sizeof(edges) / sizeof(edges[0])];
	int edge_tree_num;
	printf("最小生成树的边集是:\n");
	graph_mst_prim(g, 0, tree_edges, &edge_tree_num);
	int weight_sum = 0;
	for (int i = 0; i < edge_tree_num; i++) {
		struct edge e = tree_edges[i];
		weight_sum += e.w;
		printf("%s %s %d\n", str_vertex[e.u], str_vertex[e.v], e.w);
	}
	printf("最小生成树的权值之和是:%d\n", weight_sum);
	graph_destroy(g);
	return 0;
}
```

22.2.2.2 Prim 算法,使用斐波那契堆实现

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>
typedef struct graph_type *graph;
struct edge {
	int u;
	int v;
	int w;
};
struct graph_node {
	int key;
```

```
	int w;
	struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key, int w)
{
	x->key = key;
	x->w = w;
	x->next = NULL;
}

struct vertex {
	int v;			//顶点
	int dis;
	int parent;
	char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
	v->v = -1;
	v->dis = INT_MAX;
	v->parent = -1;
	strcpy(v->str_vertex, "");
}

struct graph_type {
	struct graph_node **adj;
	struct vertex *vertex_array;
	int v_num;
	int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
	graph g = malloc(sizeof(struct graph_type));
	g->v_num = v_num;
	g->e_num = 0;
	g->adj = malloc(sizeof(struct graph_node *) * v_num);
	g->vertex_array = malloc(sizeof(struct vertex) * v_num);
	for (int i = 0; i < v_num; i++) {
		g->adj[i] = NULL;
		strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
	}
	return g;
}
```

```
void graph_destroy(graph g)
{
	for (int i = 0; i < g->v_num; i++) {
		for (struct graph_node * x = g->adj[i]; x != NULL;) {
			struct graph_node *del = x;
			x = x->next;
			free(del);
		}
	}
	free(g->adj);
	free(g->vertex_array);
	free(g);
}

void graph_insert_edge(graph g, struct edge e)
{
	struct graph_node *u = malloc(sizeof(struct graph_node));
	graph_node_ini(u, e.u, e.w);
	struct graph_node *v = malloc(sizeof(struct graph_node));
	graph_node_ini(v, e.v, e.w);
	//从表头插入,将 v 插入到表头 u
	v->next = g->adj[e.u];
	g->adj[e.u] = v;
	//从表头插入,将 u 插入到表头 v
	u->next = g->adj[e.v];
	g->adj[e.v] = u;
	++g->e_num;
}

void graph_display(graph g)
{
	printf("%d vertices,%d edges\n", g->v_num, g->e_num);
	for (int i = 0; i < g->v_num; i++) {
		printf("%s: ", g->vertex_array[i].str_vertex);
		for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
			printf("%s,%d ", g->vertex_array[x->key].str_vertex,
				x->w);
		}
		printf("\n");
	}
}

typedef struct heap *heap;
```

```
struct heap_node {
	void *key;
	int degree;
	bool mark;
	struct heap_node *child;
	struct heap_node *left;
	struct heap_node *right;
	struct heap_node *parent;
};
struct heap {
	int (*comp) (const void *, const void *);
	struct heap_node *min;
	int num;
};
void heap_node_ini(struct heap_node *x, void *key)
{
	x->key = key;
	x->degree = 0;
	x->mark = false;
	x->parent = NULL;
	x->child = NULL;
	x->left = x;
	x->right = x;
}

void swap(void *a, void *b, size_t elem_size)
{
	if (a == NULL || b == NULL || a == b)
		return;
	char temp[elem_size];	/*变长数组 */
	memcpy(temp, a, elem_size);
	memcpy(a, b, elem_size);
	memcpy(b, temp, elem_size);
}

heap heap_create(int (*comp) (const void *, const void *))
{
	heap h = malloc(sizeof(struct heap));
	h->comp = comp;
	h->num = 0;
	h->min = NULL;
	return h;
}
```

```
//删除结点，如果只有 x 一个结点的话，这个函数无效
void list_delete(struct heap_node **pos, struct heap_node *x)
{
	if (x->right == x)//只有一个结点
	{
		*pos = NULL;
		return;
	}
	x->left->right = x->right;
	x->right->left = x->left;
	if (*pos == x) {
		*pos = x->right;
	}
}

//插入结点 x 到 pos 的左边,如果 pos 为空，pos＝x
void list_insert(struct heap_node **pos, struct heap_node *x)
{
	if (*pos == NULL) {
		*pos = x;
		x->left = x;
		x->right = x;
	} else {
		x->left = (*pos)->left;
		(*pos)->left->right = x;
		x->right = (*pos);
		(*pos)->left = x;
	}
}

void add_root(heap h, struct heap_node *x)
{
	list_insert(&h->min, x);
	x->parent = NULL;
	x->mark = false;
	if (h->comp(x->key, h->min->key) < 0) {
		h->min = x;
	}
}

//下面的过程将结点 x 插入斐波那契堆中，假定结点 x 已被分配，且 key[x]也已
填有内容
void heap_insert(heap h, struct heap_node *x)
{
```

```c
	x->degree = 0;
	x->parent = NULL;
	x->child = NULL;
	x->left = x;
	x->right = x;
	add_root(h, x);
	++h->num;
}

//最小结点
struct heap_node *heap_minimum(heap h)
{
	return h->min;
}

void heap_destroy(heap h);
//将另一个斐波那契堆合并到当前堆，另一堆合并到当前最小结点的右边
void heap_union(heap ha, heap hb)
{
	if (hb == NULL || hb->min == NULL) {
		return;
	}
	if (ha->min == NULL) {
		ha->min = hb->min;
	} else {
		//最小结点的右边结点
		struct heap_node *ha_min_right = ha->min->right;
		ha->min->right = hb->min;
		//另一个堆最小结点的左结点，即最后一个结点
		struct heap_node *hb_min_left = hb->min->left;
		hb->min->left = ha->min;
		hb_min_left->right = ha_min_right;
		ha_min_right->left = hb_min_left;
	}
	if (ha->min == NULL
		 || (hb->min != NULL && ha->comp(hb->min->key, ha->min->key) < 0)) {
		ha->min = hb->min;
	}
	ha->num += hb->num;
	hb->min = NULL;
	heap_destroy(hb);
}

void link(heap h, struct heap_node *y, struct heap_node *x)
```

```
{
    list_delete(&h->min, y);
    list_insert(&x->child, y);
    y->parent = x;
    y->mark = false;
    ++x->degree;
}

//合并根表
void consolidate(heap h)
{
    if (h->min == NULL)
        return;
    int D = floor(log(h->num) / log(1.618));    //计算 D 值
    struct heap_node *A[D];
    for (int i = 0; i < D; i++) {
        A[i] = NULL;
    }
    struct heap_node *x = NULL;
    struct heap_node *y = NULL;
    int d;
    struct heap_node *w = h->min;
    struct heap_node *end = h->min->left;
    bool loop_flag = true;
    while (loop_flag) {
        x = w;
        if (w != end) {
            w = w->right;
        } else {
            loop_flag = false;    //w 到达最后一个结点，循环结束
        }
        d = x->degree;
        while (A[d] != NULL) {
            y = A[d];
            if (h->comp(x->key, y->key) > 0) {
                swap(&x, &y, sizeof(struct heap_node *));
            }
            link(h, y, x);
            A[d] = NULL;
            ++d;
        }
        A[d] = x;
    }
    h->min = NULL;
```

```
    for (int i = 0; i < D; ++i) {
        if (A[i] != NULL) {
            add_root(h, A[i]);
        }
    }
}

//抽取具有最小关键字的结点，并返回一个指向该结点的指针
struct heap_node *heap_extract_min(heap h)
{
    struct heap_node *z = h->min;
    if (z == NULL)
        return NULL;
    struct heap_node *x = NULL;
    while (z->degree > 0) {
        x = z->child;
        if (x->right == x) {
            z->child = NULL;
        } else {
            z->child = z->child->right;
        }
        list_delete(&z->child, x);
        add_root(h, x);
        --z->degree;
    }
    if (z == z->right) {
        list_delete(&h->min, z);
    } else {
        list_delete(&h->min, z);
        consolidate(h);
    }
    --h->num;
    return z;
}

void cut(heap h, struct heap_node *x, struct heap_node *y)
{
    list_delete(&y->child, x);
    add_root(h, x);
    --y->degree;
}

void cascading_cut(heap h, struct heap_node *y)
{
```

```
	struct heap_node *z = y->parent;
	if (z == NULL)
		return;
	if (y->mark == false) {
		y->mark = true;
	} else {
		cut(h, y, z);
		cascading_cut(h, z);
	}
}

//将斐波那契堆中的某一结点 x 的关键字减少为一个新值 k，如果 k 大于 x 的当前关键字值，直接返回
void heap_decrease_key(heap h, struct heap_node *x)
{
	struct heap_node *y = x->parent;
	if (y != NULL && h->comp(x->key, y->key) < 0) {
		cut(h, x, y);
		cascading_cut(h, y);
	}
	if (h->comp(x->key, h->min->key) < 0) {
		h->min = x;
	}
}

bool heap_is_empty(heap h)
{
	return h->min == NULL;
}

void heap_destroy(heap h)
{
	while (!heap_is_empty(h)) {
		struct heap_node *x = heap_extract_min(h);
		free(x->key);
		free(x);
	}
	free(h);
}

int cmp_vertex(const void *p1, const void *p2)
{
	const struct vertex *pa = p1;
	const struct vertex *pb = p2;
```

```
    if (pa->dis < pb->dis)
        return -1;
    if (pa->dis == pb->dis)
        return 0;
    return 1;
}

void graph_mst_prim(graph g, int r, struct edge tree_edges[],
             int *tree_edge_num)
{
    heap h = heap_create(cmp_vertex);
    struct heap_node *x = NULL;
    struct heap_node *node_array[g->v_num];
    struct vertex *p_vertex;
    for (int i = 0; i < g->v_num; i++) {
        x = malloc(sizeof(struct heap_node));
        heap_node_ini(x,&g->vertex_array[i]);
        p_vertex=x->key;
        p_vertex->dis = INT_MAX;
        p_vertex->parent = -1;
        p_vertex->v = i;
        node_array[i] = x;
        heap_insert(h, x);
    }
    p_vertex = node_array[r]->key;
    p_vertex->dis = 0;
    heap_decrease_key(h, node_array[r]);
    *tree_edge_num = 0;
    while (!heap_is_empty(h)) {
        x = heap_extract_min(h);
        p_vertex = x->key;
        int u = p_vertex->v;
        if (u != r) {
            struct edge e = { p_vertex->parent, u, p_vertex->dis };
            tree_edges[(*tree_edge_num)++] = e;
        }
        free(x);
        node_array[u] = NULL;
        for (struct graph_node * p = g->adj[u]; p != NULL; p = p->next) {
            int v = p->key;
            //在队列中
            if (node_array[v] != NULL) {
                p_vertex = node_array[v]->key;
                if (p->w < p_vertex->dis) {
```

```
                    p_vertex->parent = u;
                    p_vertex->dis = p->w;
                    heap_decrease_key(h, node_array[v]);
                }
            }
        }
    }
    heap_destroy(h);
}

int main()
{
    //数据根据书上的图 23-1
    char *str_vertex[9] = { "a", "b", "c", "d", "e", "f", "g", "h", "i" };
    graph g = graph_create(9, str_vertex);
    struct edge edges[] = {
        {0, 1, 4}, {0, 7, 8}, {1, 7, 11},
        {1, 2, 8}, {2, 8, 2}, {2, 5, 4}, {2, 3, 7},
        {3, 4, 9}, {3, 5, 14}, {4, 5, 10}, {5, 6, 2},
        {6, 7, 1}, {6, 8, 6}, {7, 8, 7}
    };
    for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
        graph_insert_edge(g, edges[i]);
    }
    printf("图信息:\n");
    graph_display(g);
    struct edge tree_edges[sizeof(edges) / sizeof(edges[0])];
    int edge_tree_num;
    printf("最小生成树的边集是:\n");
    graph_mst_prim(g, 0, tree_edges, &edge_tree_num);
    int weight_sum = 0;
    for (int i = 0; i < edge_tree_num; i++) {
        struct edge e = tree_edges[i];
        weight_sum += e.w;
        printf("%s %s %d\n", str_vertex[e.u], str_vertex[e.v], e.w);
    }
    printf("最小生成树的权值之和是:%d\n", weight_sum);
    graph_destroy(g);
    return 0;
}
```

22.2.2.3 Prim 算法,使用二项堆实现

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
typedef struct graph_type *graph;
struct edge {
	int u;
	int v;
	int w;
};
struct vertex {
	int v;			//顶点
	int dis;
	int parent;
	char str_vertex[256]; //顶点的字符串表示，显示用
	//堆结点发生交换时，这个数组要更新相应位置的指针
	struct heap_node **node_array;
};
struct graph_node {
	int key;
	int w;
	struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key, int w)
{
	x->key = key;
	x->w = w;
	x->next = NULL;
}

struct graph_type {
	struct graph_node **adj;
	struct vertex *vertex_array;
	int v_num;
	int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
	graph g = malloc(sizeof(struct graph_type));
```

```
    g->v_num = v_num;
    g->e_num = 0;
    g->adj = malloc(sizeof(struct graph_node *) * v_num);
    g->vertex_array = malloc(sizeof(struct vertex) * v_num);
    for (int i = 0; i < v_num; i++) {
        g->adj[i] = NULL;
        strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
    }
    return g;
}

void graph_destroy(graph g)
{
    for (int i = 0; i < g->v_num; i++) {
        for (struct graph_node * x = g->adj[i]; x != NULL;) {
            struct graph_node *del = x;
            x = x->next;
            free(del);
        }
    }
    free(g->adj);
    free(g->vertex_array);
    free(g);
}

void graph_insert_edge(graph g, struct edge e)
{
    struct graph_node *u = malloc(sizeof(struct graph_node));
    graph_node_ini(u, e.u, e.w);
    struct graph_node *v = malloc(sizeof(struct graph_node));
    graph_node_ini(v, e.v, e.w);
    //从表头插入,将 v 插入到表头 u
    v->next = g->adj[e.u];
    g->adj[e.u] = v;
    //从表头插入,将 u 插入到表头 v
    u->next = g->adj[e.v];
    g->adj[e.v] = u;
    ++g->e_num;
}

void graph_display(graph g)
{
    printf("%d vertices,%d edges\n", g->v_num, g->e_num);
    for (int i = 0; i < g->v_num; i++) {
```

```
        printf("%s: ", g->vertex_array[i].str_vertex);
        for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
            printf("%s,%d ", g->vertex_array[x->key].str_vertex,
                    x->w);
        }
        printf("\n");
    }
}

typedef struct binomial_heap *heap;
struct heap_node {
    void *key;
    int degree;
    struct heap_node *child;
    struct heap_node *sibling;
    struct heap_node *parent;
};
struct binomial_heap {
    int (*comp) (const void *, const void *);
    //这个函数是用于结点交换时通知调用
    void (*on_swap) (struct heap_node *, struct heap_node *);
    struct heap_node *head;
};
void swap(void *a, void *b, size_t elem_size)
{
    if (a == NULL || b == NULL || a == b)
        return;
    char temp[elem_size];    /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}

void heap_node_ini(struct heap_node *x, void *key)
{
    x->key = key;
    x->degree = 0;
    x->parent = NULL;
    x->child = NULL;
    x->sibling = NULL;
}

heap heap_create(int (*comp) (const void *, const void *),
         void (*on_swap) (struct heap_node *, struct heap_node *))
```

```c
{
	heap h = malloc(sizeof(struct binomial_heap));
	h->comp = comp;
	h->on_swap = on_swap;
	h->head = NULL;
	return h;
}

//返回一个指针，它指向包含 n 个结点的二项堆 H 中具有最小关键字的结点
struct heap_node *heap_minimum(heap h)
{
	struct heap_node *y = NULL;
	struct heap_node *x = h->head;
	void *min;
	bool first = true;
	while (x != NULL) {
		if (first || h->comp(x->key, min) < 0) {
			first = false;
			min = x->key;
			y = x;
		}
		x = x->sibling;
	}
	return y;
}

bool heap_is_empty(heap h)
{
	return h->head == NULL;
}

//将结点 y 为根和 z 为根的树连接过来，使 z 成为 y 的父结点
void link(struct heap_node *y, struct heap_node *z)
{
	y->parent = z;
	y->sibling = z->child;
	z->child = y;
	z->degree = z->degree + 1;
}

void heap_destroy(heap h);
//将 ha 和 hb 合并成一个按度数的单调递增次序排列的链表
struct heap_node *heap_merge(heap ha, heap hb)
{
```

```
struct heap_node *pa = ha->head;
struct heap_node *pb = hb->head;
struct heap_node *head = NULL;
struct heap_node *tail = NULL;
while (pa != NULL && pb != NULL) {
    if (pa->degree <= pb->degree) {
        if (head == NULL) {
            head = pa;
            tail = pa;
            pa = pa->sibling;
            tail->sibling = NULL;
        } else {
            tail->sibling = pa;
            pa = pa->sibling;
            tail = tail->sibling;
            tail->sibling = NULL;
        }
    } else {
        if (head == NULL) {
            head = pb;
            tail = pb;
            pb = pb->sibling;
            tail->sibling = NULL;
        } else {
            tail->sibling = pb;
            pb = pb->sibling;
            tail = tail->sibling;
            tail->sibling = NULL;
        }
    }
}
if (pa != NULL && pb == NULL) {
    if (head == NULL) {
        head = pa;
        tail = pa;
    } else {
        tail->sibling = pa;
    }
}
if (pa == NULL && pb != NULL) {
    if (head == NULL) {
        head = pb;
        tail = pb;
    } else {
```

```
            tail->sibling = pb;
        }
    }
    hb->head = NULL;
    heap_destroy(hb);
    return head;
}

//将 hb 合并到 ha 中
void heap_union(heap ha, heap hb)
{
    //将 ha 和 hb 的根表合并成一个按度数的单调递增次序排列的链表
    ha->head = heap_merge(ha, hb);
    if (ha->head == NULL) {
        return;
    }
    struct heap_node *prev = NULL;
    struct heap_node *x = ha->head;
    struct heap_node *next = x->sibling;
    while (next != NULL) {
        //情况 1:x->degree!=next->degree
        //情况 2:x->degree==next->degree==next->sibling->degree
        if ((x->degree != next->degree) ||
             (next->sibling != NULL
              && next->sibling->degree == x->degree)) {
            prev = x;
            x = next;
        } else if (ha->comp(x->key, next->key) <= 0) {
            //                                    情                                    况
3:x->degree==next->degree!=next->sibling->degree,x->key<=next->key
            x->sibling = next->sibling;
            link(next, x);
        } else {
            //                                    情                                    况
4:x->degree==next->degree!=next->sibling->degree,next->key<=x->key
            if (prev == NULL) {
                ha->head = next;
            } else {
                prev->sibling = next;
            }
            link(x, next);
            x = next;
        }
        next = x->sibling;
```

```
	}
}
```

//反转 x 的孩子，随便把 x 的孩子的父结点置为空

```
void reverse_children(struct heap_node *x)
{
	if (x == NULL || x->child == NULL)
		return;
	struct heap_node *prev = x->child;
	struct heap_node *current = prev->sibling;
	struct heap_node *next = NULL;
	while (current != NULL) {
		next = current->sibling;
		current->sibling = prev;
		current->parent = NULL;
		prev = current;
		current = next;
	}
	x->child->sibling = NULL;
	x->child->parent = NULL;
	x->child = prev;
}
```

//下面的过程将结点 x 插入二项堆中，假定结点 x 已被分配，且 key[x]也已填有内容

```
void heap_insert(heap h, struct heap_node *x)
{
	heap hb = heap_create(h->comp, h->on_swap);
	hb->head = x;
	heap_union(h, hb);
}

struct heap_node *heap_remove_minimum(heap h)
{
	struct heap_node *x = h->head;
	if (x == NULL)
		return NULL;
	struct heap_node *prev = NULL;
	struct heap_node *min_prev = NULL;
	void *min;
	bool first = true;
	while (x != NULL) {
		if (first || h->comp(x->key, min) < 0) {
			first = false;
```

```
            min = x->key;
            min_prev = prev;
        }
        prev = x;
        x = x->sibling;
    }
    //删除结点 x
    if (min_prev == NULL) {
        x = h->head;
        h->head = x->sibling;
    } else {
        x = min_prev->sibling;
        min_prev->sibling = x->sibling;
    }
    return x;
}

//抽取具有最小关键字的结点，并返回一个指向该结点的指针
struct heap_node *heap_extract_min(heap h)
{
    struct heap_node *x = heap_remove_minimum(h);
    if (x == NULL)
        return NULL;
    reverse_children(x);
    heap hb = heap_create(h->comp, h->on_swap);
    hb->head = x->child;
    heap_union(h, hb);
    return x;
}

//将二项堆中的某一结点 x 的关键字减少为一个新值 k，如果 k 大于 x 的当前关键字值，直接返回
void heap_decrease_key(heap h, struct heap_node *x)
{
    struct heap_node *y = x;
    struct heap_node *z = y->parent;
    while (z != NULL && h->comp(y->key, z->key) < 0) {
        swap(&y->key, &z->key, sizeof(void *));
        if (h->on_swap != NULL) {
            h->on_swap(y, z);
        }
        y = z;
        z = y->parent;
    }
```

```
}

void display_node(struct heap_node *x, void (*print_key) (const void *))
{
	print_key(x->key);
	printf(" ");
	if (x->child != NULL) {
		display_node(x->child, print_key);
	}
	if (x->sibling != NULL) {
		display_node(x->sibling, print_key);
	}
}

void heap_display(heap h, void (*print_key) (const void *))
{
	display_node(h->head, print_key);
	printf("\n");
}

void heap_destroy(heap h)
{
	while (!heap_is_empty(h)) {
		struct heap_node *x = heap_extract_min(h);
		free(x->key);
		free(x);
	}
	free(h);
}

void vertex_ini(struct vertex *v)
{
	v->v = -1;
	v->dis = INT_MAX;
	v->parent = -1;
	strcpy(v->str_vertex, "");
}

void on_swap(struct heap_node *left, struct heap_node *right)
{
	struct vertex *lv = left->key;
	struct vertex *rv = right->key;
	lv->node_array[lv->v] = left;
	lv->node_array[rv->v] = right;
```

```
}

int cmp_vertex(const void *p1, const void *p2)
{
	const struct vertex *pa = p1;
	const struct vertex *pb = p2;
	if (pa->dis < pb->dis)
		return -1;
	if (pa->dis == pb->dis)
		return 0;
	return 1;
}

void graph_mst_prim(graph g, int r, struct edge tree_edges[],
			int *tree_edge_num)
{
	heap h = heap_create(cmp_vertex, on_swap);
	struct heap_node *x = NULL;
	struct heap_node *node_array[g->v_num];
	struct vertex *p_vertex;
	for (int i = 0; i < g->v_num; i++) {
		x = malloc(sizeof(struct heap_node));
		heap_node_ini(x,&g->vertex_array[i]);
		p_vertex = x->key;
		p_vertex->dis = INT_MAX;
		p_vertex->parent = -1;
		p_vertex->v = i;
		p_vertex->node_array = node_array;
		node_array[i] = x;
		heap_insert(h, x);
	}
	p_vertex = node_array[r]->key;
	p_vertex->dis = 0;
	heap_decrease_key(h, node_array[r]);
	*tree_edge_num = 0;
	while (!heap_is_empty(h)) {
		x = heap_extract_min(h);
		p_vertex = x->key;
		int u = p_vertex->v;
		if (u != r) {
			struct edge e = { p_vertex->parent, u, p_vertex->dis };
			tree_edges[(*tree_edge_num)++] = e;
		}
		free(x);
```

```
        node_array[u] = NULL;
        for (struct graph_node * p = g->adj[u]; p != NULL; p = p->next) {
            int v = p->key;
            //在队列中
            if (node_array[v] != NULL) {
                p_vertex = node_array[v]->key;
                if (p->w < p_vertex->dis) {
                    p_vertex->parent = u;
                    p_vertex->dis = p->w;
                    heap_decrease_key(h, node_array[v]);
                }
            }
        }
    }
    heap_destroy(h);
}

int main()
{
    //数据根据书上的图 23-1
    char *str_vertex[9] = { "a", "b", "c", "d", "e", "f", "g", "h", "i" };
    graph g = graph_create(9, str_vertex);
    struct edge edges[] = {
        {0, 1, 4}, {0, 7, 8}, {1, 7, 11},
        {1, 2, 8}, {2, 8, 2}, {2, 5, 4}, {2, 3, 7},
        {3, 4, 9}, {3, 5, 14}, {4, 5, 10}, {5, 6, 2},
        {6, 7, 1}, {6, 8, 6}, {7, 8, 7}
    };
    for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
        graph_insert_edge(g, edges[i]);
    }
    printf("图信息:\n");
    graph_display(g);
    struct edge tree_edges[sizeof(edges) / sizeof(edges[0])];
    int edge_tree_num;
    printf("最小生成树的边集是:\n");
    graph_mst_prim(g, 0, tree_edges, &edge_tree_num);
    int weight_sum = 0;
    for (int i = 0; i < edge_tree_num; i++) {
        struct edge e = tree_edges[i];
        weight_sum += e.w;
        printf("%s %s %d\n", str_vertex[e.u], str_vertex[e.v], e.w);
    }
    printf("最小生成树的权值之和是:%d\n", weight_sum);
```

```
	graph_destroy(g);
	return 0;
}
```

第 24 章 单源最源路径

24.1 Bellman-Ford 算法

```
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
typedef struct graph_type *graph;
struct edge {
	int u;
	int v;
	int w;
};
struct graph_node {
	int key;
	int w;
	struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key, int w)
{
	x->key = key;
	x->w = w;
	x->next = NULL;
}

struct vertex {
	int dis;
	int parent;
	char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
	v->dis = INT_MAX;
	v->parent = -1;
	strcpy(v->str_vertex, "");
```

```
}

struct graph_type {
	struct graph_node **adj;
	struct vertex *vertex_array;
	int v_num;
	int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
	graph g = malloc(sizeof(struct graph_type));
	g->v_num = v_num;
	g->e_num = 0;
	g->adj = malloc(sizeof(struct graph_node *) * v_num);
	g->vertex_array = malloc(sizeof(struct vertex) * v_num);
	for (int i = 0; i < v_num; i++) {
		g->adj[i] = NULL;
		strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
	}
	return g;
}

void graph_destroy(graph g)
{
	for (int i = 0; i < g->v_num; i++) {
		for (struct graph_node * x = g->adj[i]; x != NULL;) {
			struct graph_node *del = x;
			x = x->next;
			free(del);
		}
	}
	free(g->adj);
	free(g->vertex_array);
	free(g);
}

void graph_insert_edge(graph g, struct edge e)
{
	struct graph_node *v = malloc(sizeof(struct graph_node));
	graph_node_ini(v, e.v, e.w);
	//从表头插入,将 v 插入到表头 u
	v->next = g->adj[e.u];
	g->adj[e.u] = v;
```

```
	++g->e_num;
}

void graph_display(graph g)
{
	printf("%d vertices,%d edges\n", g->v_num, g->e_num);
	for (int i = 0; i < g->v_num; i++) {
		printf("%s: ", g->vertex_array[i].str_vertex);
		for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
			printf("%s,%d ", g->vertex_array[x->key].str_vertex,
				x->w);
		}
		printf("\n");
	}
}

void initialize_single_source(graph g, int s)
{
	for (int i = 0; i < g->v_num; i++) {
		g->vertex_array[i].dis = INT_MAX;
		g->vertex_array[i].parent = -1;
	}
	g->vertex_array[s].dis = 0;
}

void relax(graph g, int u, int v, int w)
{
	int dis = g->vertex_array[u].dis ==
		INT_MAX ? INT_MAX : g->vertex_array[u].dis + w;
	if (g->vertex_array[v].dis > dis) {
		g->vertex_array[v].dis = dis;
		g->vertex_array[v].parent = u;
	}
}

bool graph_bellman_ford(graph g, int s)
{
	initialize_single_source(g, s);
	for (int i = 0; i < g->v_num - 1; i++) {
		for (int u = 0; u < g->v_num; u++) {
			for (struct graph_node * x = g->adj[u]; x != NULL;
				x = x->next) {
				relax(g, u, x->key, x->w);
			}
```

```
        }
    }
    for (int u = 0; u < g->v_num; u++) {
        for (struct graph_node * x = g->adj[u]; x != NULL; x = x->next) {
            if (g->vertex_array[x->key].dis >
                 g->vertex_array[u].dis + x->w) {
                return false;
            }
        }
    }
    return true;
}

void graph_display_vertex(graph g)
{
    printf("各个顶点的数据:\n");
    for (int i = 0; i < g->v_num; i++) {
        printf("%s:%d\n", g->vertex_array[i].str_vertex,
                g->vertex_array[i].dis);
    }
}

int main()
{
    //数据根据书上的图 24-4
    char *str_vertex[5] = { "s", "t", "x", "y", "z" };
    graph g = graph_create(5, str_vertex);
    struct edge edges[] = { {0, 1, 6}, {0, 3, 7}, {1, 2, 5},
    {1, 3, 8}, {1, 4, -4}, {2, 1, -2}, {3, 2, -3},
    {3, 4, 9}, {4, 0, 2}, {4, 2, 7}
    };
    for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
        graph_insert_edge(g, edges[i]);
    }
    graph_display(g);
    graph_bellman_ford(g, 0);
    graph_display_vertex(g);
}
```

24.2 有向无回路图中的单源最短路径

```
#include <stdio.h>
#include <limits.h>
```

```c
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
typedef struct graph_type *graph;
struct edge {
	int u;
	int v;
	int w;
};
struct graph_node {
	int key;
	int w;
	struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key, int w)
{
	x->key = key;
	x->w = w;
	x->next = NULL;
}

struct vertex {
	int dis;
	int parent;
	bool in_queue;		//是否在队列里面
	char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
	v->dis = INT_MAX;
	v->parent = -1;
	v->in_queue = false;
	strcpy(v->str_vertex, "");
}

struct graph_type {
	struct graph_node **adj;
	struct vertex *vertex_array;
	int v_num;
	int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
```

```
    graph g = malloc(sizeof(struct graph_type));
    g->v_num = v_num;
    g->e_num = 0;
    g->adj = malloc(sizeof(struct graph_node *) * v_num);
    g->vertex_array = malloc(sizeof(struct vertex) * v_num);
    for (int i = 0; i < v_num; i++) {
        g->adj[i] = NULL;
        strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
    }
    return g;
}

void graph_destroy(graph g)
{
    for (int i = 0; i < g->v_num; i++) {
        for (struct graph_node * x = g->adj[i]; x != NULL;) {
            struct graph_node *del = x;
            x = x->next;
            free(del);
        }
    }
    free(g->adj);
    free(g->vertex_array);
    free(g);
}

void graph_insert_edge(graph g, struct edge e)
{
    struct graph_node *v = malloc(sizeof(struct graph_node));
    graph_node_ini(v, e.v, e.w);
    //从表头插入,将 v 插入到表头 u
    v->next = g->adj[e.u];
    g->adj[e.u] = v;
    ++g->e_num;
}

void graph_display(graph g)
{
    printf("%d vertices,%d edges\n", g->v_num, g->e_num);
    for (int i = 0; i < g->v_num; i++) {
        printf("%s: ", g->vertex_array[i].str_vertex);
        for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
            printf("%s,%d ", g->vertex_array[x->key].str_vertex,
                    x->w);
```

```
        }
        printf("\n");
    }
}

void graph_display_vertex(graph g)
{
    printf("各个顶点的数据:\n");
    for (int i = 0; i < g->v_num; i++) {
        printf("%s:%d\n", g->vertex_array[i].str_vertex,
                g->vertex_array[i].dis);
    }
}

void swap(void *a, void *b, size_t elem_size)
{
    if (a == NULL || b == NULL || a == b)
        return;
    char temp[elem_size];    /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}

/*基于索引堆的优先队列*/
typedef struct priority_queue_index_type *priority_queue;
struct priority_queue_index_type {
    int heap_size;
    int *index_array;
    int *index_pos_array;/*这个数组记录了索引在堆中位置 */
    void *data_array;
    size_t elem_size;
    int (*comp) (const void *, const void *);
};
int parent(int i)
{
    return (i - 1) / 2;
}

int left_child(int i)
{
    return i * 2 + 1;
}
```

```
int right_child(int i)
{
    return i * 2 + 2;
}

void swap_index(priority_queue pq, int i, int j)
{
    swap(&pq->index_pos_array[i], &pq->index_pos_array[j], sizeof(int));
    pq->index_array[pq->index_pos_array[i]] = i;
    pq->index_array[pq->index_pos_array[j]] = j;
}

/*最小堆用的比较函数*/
bool compare(priority_queue pq, int left, int right)
{
    if (pq->data_array == NULL)
        return false;
    char *pc_array = pq->data_array;
    return pq->comp(&pc_array[left * pq->elem_size],
            &pc_array[right * pq->elem_size]) > 0;
}

void heapify(priority_queue pq, int i)
{
    int left = left_child(i);
    int right = right_child(i);
    int largest = i;
    if (left < pq->heap_size
         && compare(pq, pq->index_array[largest], pq->index_array[left])) {
        largest = left;
    }
    if (right < pq->heap_size
         && compare(pq, pq->index_array[largest], pq->index_array[right])) {
        largest = right;
    }
    if (largest != i) {
        swap_index(pq, pq->index_array[i], pq->index_array[largest]);
        heapify(pq, largest);
    }
}

void fix_up(priority_queue pq, int i)
{
    while (i > 0
```

```
            && compare(pq, pq->index_array[parent(i)], pq->index_array[i])) {
        swap_index(pq, pq->index_array[parent(i)], pq->index_array[i]);
        i = parent(i);
    }
}

priority_queue priority_queue_create(void *p_data_array, size_t elem_size,
                      int length, int (*comp) (const void *,
                                     const void *))
{
    priority_queue pq = malloc(sizeof(struct priority_queue_index_type));
    pq->index_array = malloc(sizeof(int) * length);
    pq->index_pos_array = malloc(sizeof(int) * length);
    pq->data_array = p_data_array;
    pq->elem_size = elem_size;
    pq->heap_size = 0;
    pq->comp = comp;
    return pq;
}

void priority_queue_destroy(priority_queue pq)
{
    free(pq->index_array);
    free(pq->index_pos_array);
    free(pq);
}

int priority_queue_top(priority_queue pq)
{
    return pq->index_array[0];
}

/*去掉并返回堆的第一个元素 */
int priority_queue_extract_top(priority_queue pq)
{
    swap_index(pq, pq->index_array[0], pq->index_array[pq->heap_size - 1]);
    --pq->heap_size;
    heapify(pq, 0);
    return pq->index_array[pq->heap_size];
}

/*把元素的索引插入队列 */
void priority_queue_insert(priority_queue pq, int index)
{
```

```
    ++pq->heap_size;
    int i = pq->heap_size - 1;
    pq->index_array[i] = index;
    pq->index_pos_array[index] = i;
    fix_up(pq, i);
}

bool priority_queue_is_empty(priority_queue pq)
{
    return pq->heap_size == 0;
}

/*下标为 index 的数据修改了，调用这个函数来修复索引堆*/
void priority_queue_change_index(priority_queue pq, int index)
{
    fix_up(pq, pq->index_pos_array[index]);
    heapify(pq, pq->index_pos_array[index]);
}

int cmp_vertex(const void *p1, const void *p2)
{
    const struct vertex *pa = p1;
    const struct vertex *pb = p2;
    if (pa->dis < pb->dis)
        return -1;
    if (pa->dis == pb->dis)
        return 0;
    return 1;
}

void initialize_single_source(graph g, int s)
{
    for (int i = 0; i < g->v_num; i++) {
        g->vertex_array[i].dis = INT_MAX;
        g->vertex_array[i].parent = -1;
    }
    g->vertex_array[s].dis = 0;
}

void relax(graph g, int u, int v, int w)
{
    int dis = g->vertex_array[u].dis ==
         INT_MAX ? INT_MAX : g->vertex_array[u].dis + w;
    if (g->vertex_array[v].dis > dis) {
```

```
            g->vertex_array[v].dis = dis;
            g->vertex_array[v].parent = u;
        }
}

void dijkstra(graph g, int s)
{
        initialize_single_source(g, s);
        priority_queue pq =
                priority_queue_create(g->vertex_array, sizeof(struct vertex),
                                g->v_num, cmp_vertex);
        for (int i = 0; i < g->v_num; i++) {
                priority_queue_insert(pq, i);
        }
        priority_queue_change_index(pq, s);
        while (!priority_queue_is_empty(pq)) {
                int u = priority_queue_extract_top(pq);
                for (struct graph_node * x = g->adj[u]; x != NULL; x = x->next) {
                        int v = x->key;
                        relax(g, u, v, x->w);
                        priority_queue_change_index(pq, v);
                }
        }
        priority_queue_destroy(pq);
}

int main()
{
        //数据根据书上的图 24-6
        char *str_vertex[5] = { "s", "t", "x", "y", "z" };
        graph g = graph_create(5, str_vertex);
        struct edge edges[] =
                { {0, 1, 10}, {0, 3, 5}, {1, 2, 1}, {1, 3, 2}, {2, 4, 4}, {3, 1, 3},
        {3, 4, 2}, {3, 2, 9}, {4, 0, 7}, {4, 2, 6}
        };
        for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
                graph_insert_edge(g, edges[i]);
        }
        graph_display(g);
        dijkstra(g, 0);
        graph_display_vertex(g);
        graph_destroy(g);
}
```

24.3 Dijkstra 算法

24.3.1 Dijkstra 算法,使用最小优先级队列实现

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
typedef struct graph_type *graph;
struct edge {
	int u;
	int v;
	int w;
};
struct graph_node {
	int key;
	int w;
	struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key, int w)
{
	x->key = key;
	x->w = w;
	x->next = NULL;
}

struct vertex {
	int dis;
	int parent;
	char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
	v->dis = INT_MAX;
	v->parent = -1;
	strcpy(v->str_vertex, "");
}

struct graph_type {
	struct graph_node **adj;
	struct vertex *vertex_array;
	int v_num;
```

```c
	int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
	graph g = malloc(sizeof(struct graph_type));
	g->v_num = v_num;
	g->e_num = 0;
	g->adj = malloc(sizeof(struct graph_node *) * v_num);
	g->vertex_array = malloc(sizeof(struct vertex) * v_num);
	for (int i = 0; i < v_num; i++) {
		g->adj[i] = NULL;
		strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
	}
	return g;
}

void graph_destroy(graph g)
{
	for (int i = 0; i < g->v_num; i++) {
		for (struct graph_node * x = g->adj[i]; x != NULL;) {
			struct graph_node *del = x;
			x = x->next;
			free(del);
		}
	}
	free(g->adj);
	free(g->vertex_array);
	free(g);
}

void graph_insert_edge(graph g, struct edge e)
{
	struct graph_node *v = malloc(sizeof(struct graph_node));
	graph_node_ini(v, e.v, e.w);
	//从表头插入,将 v 插入到表头 u
	v->next = g->adj[e.u];
	g->adj[e.u] = v;
	++g->e_num;
}

void graph_display(graph g)
{
	printf("%d vertices,%d edges\n", g->v_num, g->e_num);
```

```
	for (int i = 0; i < g->v_num; i++) {
		printf("%s: ", g->vertex_array[i].str_vertex);
		for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
			printf("%s,%d ", g->vertex_array[x->key].str_vertex,
					x->w);
		}
		printf("\n");
	}
}

void graph_display_vertex(graph g)
{
	printf("各个顶点的数据:\n");
	for (int i = 0; i < g->v_num; i++) {
		printf("%s:%d\n", g->vertex_array[i].str_vertex,
				g->vertex_array[i].dis);
	}
}

void swap(void *a, void *b, size_t elem_size)
{
	if (a == NULL || b == NULL || a == b)
		return;
	char temp[elem_size];	/*变长数组 */
	memcpy(temp, a, elem_size);
	memcpy(a, b, elem_size);
	memcpy(b, temp, elem_size);
}

/*基于索引堆的优先队列*/
typedef struct priority_queue_index_type *priority_queue;
struct priority_queue_index_type {
	int heap_size;
	int *index_array;
	int *index_pos_array;/*这个数组记录了索引在堆中位置 */
	void *data_array;
	size_t elem_size;
	int (*comp) (const void *, const void *);
};
int parent(int i)
{
	return (i - 1) / 2;
}
```

```
int left_child(int i)
{
    return i * 2 + 1;
}

int right_child(int i)
{
    return i * 2 + 2;
}

void swap_index(priority_queue pq, int i, int j)
{
    swap(&pq->index_pos_array[i], &pq->index_pos_array[j], sizeof(int));
    pq->index_array[pq->index_pos_array[i]] = i;
    pq->index_array[pq->index_pos_array[j]] = j;
}

/*最小堆用的比较函数*/
bool compare(priority_queue pq, int left, int right)
{
    if (pq->data_array == NULL)
        return false;
    char *pc_array = pq->data_array;
    return pq->comp(&pc_array[left * pq->elem_size],
            &pc_array[right * pq->elem_size]) > 0;
}

void heapify(priority_queue pq, int i)
{
    int left = left_child(i);
    int right = right_child(i);
    int largest = i;
    if (left < pq->heap_size
         && compare(pq, pq->index_array[largest], pq->index_array[left])) {
        largest = left;
    }
    if (right < pq->heap_size
         && compare(pq, pq->index_array[largest], pq->index_array[right])) {
        largest = right;
    }
    if (largest != i) {
        swap_index(pq, pq->index_array[i], pq->index_array[largest]);
        heapify(pq, largest);
    }
```

```
}

void fix_up(priority_queue pq, int i)
{
    while (i > 0
            && compare(pq, pq->index_array[parent(i)], pq->index_array[i])) {
        swap_index(pq, pq->index_array[parent(i)], pq->index_array[i]);
        i = parent(i);
    }
}

priority_queue priority_queue_create(void *p_data_array, size_t elem_size,
                int length, int (*comp) (const void *,
                        const void *))
{
    priority_queue pq = malloc(sizeof(struct priority_queue_index_type));
    pq->index_array = malloc(sizeof(int) * length);
    pq->index_pos_array = malloc(sizeof(int) * length);
    pq->data_array = p_data_array;
    pq->elem_size = elem_size;
    pq->heap_size = 0;
    pq->comp = comp;
    return pq;
}

void priority_queue_destroy(priority_queue pq)
{
    free(pq->index_array);
    free(pq->index_pos_array);
    free(pq);
}

int priority_queue_top(priority_queue pq)
{
    return pq->index_array[0];
}

/*去掉并返回堆的第一个元素 */
int priority_queue_extract_top(priority_queue pq)
{
    swap_index(pq, pq->index_array[0], pq->index_array[pq->heap_size - 1]);
    --pq->heap_size;
    heapify(pq, 0);
    return pq->index_array[pq->heap_size];
```

```
}

/*把元素的索引插入队列 */
void priority_queue_insert(priority_queue pq, int index)
{
	++pq->heap_size;
	int i = pq->heap_size - 1;
	pq->index_array[i] = index;
	pq->index_pos_array[index] = i;
	fix_up(pq, i);
}

bool priority_queue_is_empty(priority_queue pq)
{
	return pq->heap_size == 0;
}

/*下标为 index 的数据修改了，调用这个函数来修复索引堆*/
void priority_queue_change_index(priority_queue pq, int index)
{
	fix_up(pq, pq->index_pos_array[index]);
	heapify(pq, pq->index_pos_array[index]);
}

int cmp_vertex(const void *p1, const void *p2)
{
	const struct vertex *pa = p1;
	const struct vertex *pb = p2;
	if (pa->dis < pb->dis)
		return -1;
	if (pa->dis == pb->dis)
		return 0;
	return 1;
}

void initialize_single_source(graph g, int s)
{
	for (int i = 0; i < g->v_num; i++) {
		g->vertex_array[i].dis = INT_MAX;
		g->vertex_array[i].parent = -1;
	}
	g->vertex_array[s].dis = 0;
}
```

```
void relax(graph g, int u, int v, int w)
{
	int dis = g->vertex_array[u].dis ==
		INT_MAX ? INT_MAX : g->vertex_array[u].dis + w;
	if (g->vertex_array[v].dis > dis) {
		g->vertex_array[v].dis = dis;
		g->vertex_array[v].parent = u;
	}
}

void dijkstra(graph g, int s)
{
	initialize_single_source(g, s);
	priority_queue pq =
		priority_queue_create(g->vertex_array, sizeof(struct vertex),
				g->v_num, cmp_vertex);
	for (int i = 0; i < g->v_num; i++) {
		priority_queue_insert(pq, i);
	}
	priority_queue_change_index(pq, s);
	while (!priority_queue_is_empty(pq)) {
		int u = priority_queue_extract_top(pq);
		for (struct graph_node * x = g->adj[u]; x != NULL; x = x->next) {
			int v = x->key;
			relax(g, u, v, x->w);
			priority_queue_change_index(pq, v);
		}
	}
	priority_queue_destroy(pq);
}

int main()
{
	//数据根据书上的图 24-6
	char *str_vertex[5] = { "s", "t", "x", "y", "z" };
	graph g = graph_create(5, str_vertex);
	struct edge edges[] =
		{ {0, 1, 10}, {0, 3, 5}, {1, 2, 1}, {1, 3, 2}, {2, 4, 4}, {3, 1, 3},
	{3, 4, 2}, {3, 2, 9}, {4, 0, 7}, {4, 2, 6}
	};
	for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
		graph_insert_edge(g, edges[i]);
	}
	graph_display(g);
```

```
	dijkstra(g, 0);
	graph_display_vertex(g);
	graph_destroy(g);
	return 0;
}
```

24.3.2 Dijkstra 算法,使用斐波那契堆实现

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <math.h>
typedef struct graph_type *graph;
struct edge {
	int u;
	int v;
	int w;
};
struct graph_node {
	int key;
	int w;
	struct graph_node *next;
};
void graph_node_ini(struct graph_node *x, int key, int w)
{
	x->key = key;
	x->w = w;
	x->next = NULL;
}

struct vertex {
	int v;			//顶点
	int dis;
	int parent;
	char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
	v->v = -1;
	v->dis = INT_MAX;
	v->parent = -1;
```

```
    strcpy(v->str_vertex, "");
}

struct graph_type {
    struct graph_node **adj;
    struct vertex *vertex_array;
    int v_num;
    int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
    graph g = malloc(sizeof(struct graph_type));
    g->v_num = v_num;
    g->e_num = 0;
    g->adj = malloc(sizeof(struct graph_node *) * v_num);
    g->vertex_array = malloc(sizeof(struct vertex) * v_num);
    for (int i = 0; i < v_num; i++) {
        g->adj[i] = NULL;
        strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
    }
    return g;
}

void graph_destroy(graph g)
{
    for (int i = 0; i < g->v_num; i++) {
        for (struct graph_node * x = g->adj[i]; x != NULL;) {
            struct graph_node *del = x;
            x = x->next;
            free(del);
        }
    }
    free(g->adj);
    free(g->vertex_array);
    free(g);
}

void graph_insert_edge(graph g, struct edge e)
{
    struct graph_node *v = malloc(sizeof(struct graph_node));
    graph_node_ini(v, e.v, e.w);
    //从表头插入,将 v 插入到表头 u
    v->next = g->adj[e.u];
```

```
	g->adj[e.u] = v;
	++g->e_num;
}

void graph_display(graph g)
{
	printf("%d vertices,%d edges\n", g->v_num, g->e_num);
	for (int i = 0; i < g->v_num; i++) {
		printf("%s: ", g->vertex_array[i].str_vertex);
		for (struct graph_node * x = g->adj[i]; x != NULL; x = x->next) {
			printf("%s,%d ", g->vertex_array[x->key].str_vertex,
					x->w);
		}
		printf("\n");
	}
}

void graph_display_vertex(graph g)
{
	printf("各个顶点的数据:\n");
	for (int i = 0; i < g->v_num; i++) {
		printf("%s:%d\n", g->vertex_array[i].str_vertex,
				g->vertex_array[i].dis);
	}
}

typedef struct heap *heap;
struct heap_node {
	void *key;
	int degree;
	bool mark;
	struct heap_node *child;
	struct heap_node *left;
	struct heap_node *right;
	struct heap_node *parent;
};
struct heap {
	int (*comp) (const void *, const void *);
	struct heap_node *min;
	int num;
};
void heap_node_ini(struct heap_node *x, void *key)
{
	x->key = key;
```

```
    x->degree = 0;
    x->mark = false;
    x->parent = NULL;
    x->child = NULL;
    x->left = x;
    x->right = x;
}

void swap(void *a, void *b, size_t elem_size)
{
    if (a == NULL || b == NULL || a == b)
        return;
    char temp[elem_size];    /*变长数组 */
    memcpy(temp, a, elem_size);
    memcpy(a, b, elem_size);
    memcpy(b, temp, elem_size);
}

heap heap_create(int (*comp) (const void *, const void *))
{
    heap h = malloc(sizeof(struct heap));
    h->comp = comp;
    h->num = 0;
    h->min = NULL;
    return h;
}

//删除结点，如果只有 x 一个结点的话，这个函数无效
void list_delete(struct heap_node **pos, struct heap_node *x)
{
    if (x->right == x)//只有一个结点
    {
        *pos = NULL;
        return;
    }
    x->left->right = x->right;
    x->right->left = x->left;
    if (*pos == x) {
        *pos = x->right;
    }
}

//插入结点 x 到 pos 的左边,如果 pos 为空，pos＝x
void list_insert(struct heap_node **pos, struct heap_node *x)
```

```
{
	if (*pos == NULL) {
		*pos = x;
		x->left = x;
		x->right = x;
	} else {
		x->left = (*pos)->left;
		(*pos)->left->right = x;
		x->right = (*pos);
		(*pos)->left = x;
	}
}

void add_root(heap h, struct heap_node *x)
{
	list_insert(&h->min, x);
	x->parent = NULL;
	x->mark = false;
	if (h->comp(x->key, h->min->key) < 0) {
		h->min = x;
	}
}

//下面的过程将结点 x 插入斐波那契堆中，假定结点 x 已被分配，且 key[x]也已填有内容
void heap_insert(heap h, struct heap_node *x)
{
	x->degree = 0;
	x->parent = NULL;
	x->child = NULL;
	x->left = x;
	x->right = x;
	add_root(h, x);
	++h->num;
}

//最小结点
struct heap_node *heap_minimum(heap h)
{
	return h->min;
}

void heap_destroy(heap h);
//将另一个斐波那契堆合并到当前堆，另一堆合并到当前最小结点的右边
```

```
void heap_union(heap ha, heap hb)
{
    if (hb == NULL || hb->min == NULL) {
        return;
    }
    if (ha->min == NULL) {
        ha->min = hb->min;
    } else {
        //最小结点的右边结点
        struct heap_node *ha_min_right = ha->min->right;
        ha->min->right = hb->min;
        //另一个堆最小结点的左结点，即最后一个结点
        struct heap_node *hb_min_left = hb->min->left;
        hb->min->left = ha->min;
        hb_min_left->right = ha_min_right;
        ha_min_right->left = hb_min_left;
    }
    if (ha->min == NULL
         || (hb->min != NULL && ha->comp(hb->min->key, ha->min->key) < 0)) {
        ha->min = hb->min;
    }
    ha->num += hb->num;
    hb->min = NULL;
    heap_destroy(hb);
}

void link(heap h, struct heap_node *y, struct heap_node *x)
{
    list_delete(&h->min, y);
    list_insert(&x->child, y);
    y->parent = x;
    y->mark = false;
    ++x->degree;
}

//合并根表
void consolidate(heap h)
{
    if (h->min == NULL)
        return;
    int D = floor(log(h->num) / log(1.618));    //计算 D 值
    struct heap_node *A[D];
    for (int i = 0; i < D; i++) {
        A[i] = NULL;
```

```
	}
	struct heap_node *x = NULL;
	struct heap_node *y = NULL;
	int d;
	struct heap_node *w = h->min;
	struct heap_node *end = h->min->left;
	bool loop_flag = true;
	while (loop_flag) {
		x = w;
		if (w != end) {
			w = w->right;
		} else {
			loop_flag = false;	//w 到达最后一个结点，循环结束
		}
		d = x->degree;
		while (A[d] != NULL) {
			y = A[d];
			if (h->comp(x->key, y->key) > 0) {
				swap(&x, &y, sizeof(struct heap_node *));
			}
			link(h, y, x);
			A[d] = NULL;
			++d;
		}
		A[d] = x;
	}
	h->min = NULL;
	for (int i = 0; i < D; ++i) {
		if (A[i] != NULL) {
			add_root(h, A[i]);
		}
	}
}

//抽取具有最小关键字的结点，并返回一个指向该结点的指针
struct heap_node *heap_extract_min(heap h)
{
	struct heap_node *z = h->min;
	if (z == NULL)
		return NULL;
	struct heap_node *x = NULL;
	while (z->degree > 0) {
		x = z->child;
		if (x->right == x) {
```

```
            z->child = NULL;
        } else {
            z->child = z->child->right;
        }
        list_delete(&z->child, x);
        add_root(h, x);
        --z->degree;
    }
    if (z == z->right) {
        list_delete(&h->min, z);
    } else {
        list_delete(&h->min, z);
        consolidate(h);
    }
    --h->num;
    return z;
}

void cut(heap h, struct heap_node *x, struct heap_node *y)
{
    list_delete(&y->child, x);
    add_root(h, x);
    --y->degree;
}

void cascading_cut(heap h, struct heap_node *y)
{
    struct heap_node *z = y->parent;
    if (z == NULL)
        return;
    if (y->mark == false) {
        y->mark = true;
    } else {
        cut(h, y, z);
        cascading_cut(h, z);
    }
}

//将斐波那契堆中的某一结点 x 的关键字减少为一个新值 k，如果 k 大于 x 的当前关键字值，直接返回
void heap_decrease_key(heap h, struct heap_node *x)
{
    struct heap_node *y = x->parent;
    if (y != NULL && h->comp(x->key, y->key) < 0) {
```

```
        cut(h, x, y);
        cascading_cut(h, y);
    }
    if (h->comp(x->key, h->min->key) < 0) {
        h->min = x;
    }
}

bool heap_is_empty(heap h)
{
    return h->min == NULL;
}

void heap_destroy(heap h)
{
    while (!heap_is_empty(h)) {
        struct heap_node *x = heap_extract_min(h);
        free(x->key);
        free(x);
    }
    free(h);
}

int cmp_vertex(const void *p1, const void *p2)
{
    const struct vertex *pa = p1;
    const struct vertex *pb = p2;
    if (pa->dis < pb->dis)
        return -1;
    if (pa->dis == pb->dis)
        return 0;
    return 1;
}

void initialize_single_source(graph g, int s)
{
    for (int i = 0; i < g->v_num; i++) {
        g->vertex_array[i].dis = INT_MAX;
        g->vertex_array[i].parent = -1;
    }
    g->vertex_array[s].dis = 0;
}

void relax(graph g, int u, int v, int w)
```

```
{
	int dis = g->vertex_array[u].dis ==
		INT_MAX ? INT_MAX : g->vertex_array[u].dis + w;
	if (g->vertex_array[v].dis > dis) {
		g->vertex_array[v].dis = dis;
		g->vertex_array[v].parent = u;
	}
}

void dijkstra(graph g, int s)
{
	heap h = heap_create(cmp_vertex);
	struct heap_node *x = NULL;
	struct heap_node *node_array[g->v_num];
	struct vertex *p_vertex;
	initialize_single_source(g, s);
	for (int i = 0; i < g->v_num; i++) {
		x = malloc(sizeof(struct heap_node));
		heap_node_ini(x,&g->vertex_array[i]);
		p_vertex = x->key;
		p_vertex->v = i;
		node_array[i] = x;
		heap_insert(h, x);
	}
	heap_decrease_key(h, node_array[s]);
	while (!heap_is_empty(h)) {
		x = heap_extract_min(h);
		p_vertex = x->key;
		int u = p_vertex->v;
		free(x);
		node_array[u] = NULL;
		for (struct graph_node * x = g->adj[u]; x != NULL; x = x->next) {
			int v = x->key;
			if (node_array[v] != NULL) {
				relax(g, u, v, x->w);
				heap_decrease_key(h, node_array[v]);
			}
		}
	}
	heap_destroy(h);
}

int main()
{
```

```
    //数据根据书上的图 24-6
    char *str_vertex[5] = { "s", "t", "x", "y", "z" };
    graph g = graph_create(5, str_vertex);
    struct edge edges[] =
        { {0, 1, 10}, {0, 3, 5}, {1, 2, 1}, {1, 3, 2}, {2, 4, 4}, {3, 1, 3},
    {3, 4, 2}, {3, 2, 9}, {4, 0, 7}, {4, 2, 6}
    };
    for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
        graph_insert_edge(g, edges[i]);
    }
    graph_display(g);
    dijkstra(g, 0);
    graph_display_vertex(g);
    graph_destroy(g);
}
```

第 25 章 每对顶点间的最短路径

25.1 最短路径与矩阵乘法

```
#include <stdio.h>
#include <limits.h>
#include <string.h>
#include <stdlib.h>
typedef struct graph_type *graph;
struct edge {
    int u;
    int v;
    int w;
};
struct vertex {
    char str_vertex[256]; //顶点的字符串表示，显示用
};
void vertex_ini(struct vertex *v)
{
    strcpy(v->str_vertex, "");
}

struct graph_type {
    int **adj;
    struct vertex *vertex_array;
```

```
    int v_num;
    int e_num;
};
//顶点是编号为 0~v_num-1 的数,str_vertex 是顶点的字符串表示，显示用
graph graph_create(int v_num, char *str_vertex[])
{
    graph g = malloc(sizeof(struct graph_type));
    g->v_num = v_num;
    g->e_num = 0;
    g->adj = malloc(sizeof(int *) * v_num);
    for (int i = 0; i < v_num; i++) {
        g->adj[i] = malloc(sizeof(int) * v_num);
        for (int j = 0; j < v_num; j++) {
            g->adj[i][j] = 0;
        }
    }
    g->vertex_array = malloc(sizeof(struct vertex) * v_num);
    for (int i = 0; i < v_num; i++) {
        strcpy(g->vertex_array[i].str_vertex, str_vertex[i]);
    }
    return g;
}

void graph_destroy(graph g)
{
    for (int i = 0; i < g->v_num; i++) {
        free(g->adj[i]);
    }
    free(g->adj);
    free(g->vertex_array);
    free(g);
}

void graph_insert_edge(graph g, struct edge edge)
{
    g->adj[edge.u][edge.v] = edge.w;
    ++g->e_num;
}

void graph_display(graph g)
{
    printf("%d vertices,%d edges\n", g->v_num, g->e_num);
    for (int i = 0; i < g->v_num; i++) {
        printf("%s: ", g->vertex_array[i].str_vertex);
```

```
        for (int j = 0; j < g->v_num; j++) {
            if (g->adj[i][j] != 0) {
                printf("%s,%d ", g->vertex_array[j].str_vertex,
                        g->adj[i][j]);
            }
        }
        printf("\n");
    }
}

typedef struct matrix_type *matrix;
struct matrix_type {
    int row;
    int col;
    int **data;
};
matrix matrix_create(int row, int col)
{
    if (row == 0)
        return NULL;
    matrix m = malloc(sizeof(struct matrix_type));
    m->row = row;
    m->col = col;
    m->data = malloc(sizeof(int *) * row);
    for (int i = 0; i < row; i++) {
        m->data[i] = malloc(sizeof(int) * col);
        for (int j = 0; j < col; j++) {
            m->data[i][j] = 0;
        }
    }
    return m;
}

void matrix_destroy(matrix m)
{
    for (int i = 0; i < m->row; i++)
        free(m->data[i]);
    free(m->data);
    free(m);
}

void matrix_display(matrix m)
{
    for (int i = 0; i < m->row; ++i) {
```

```
		for (int j = 0; j < m->col; ++j) {
			printf("%2d ", m->data[i][j]);
		}
		printf("\n");
	}
}

void matrix_multiply(matrix A, matrix B, matrix C)
{
	if (A->col != B->row) {
		return;
	}
	for (int i = 0; i < A->row; ++i) {
		for (int j = 0; j < B->col; ++j) {
			C->data[i][j] = 0;
			for (int k = 0; k < A->col; ++k) {
				C->data[i][j] += A->data[i][k] * B->data[k][j];
			}
		}
	}
}

void matrix_copy(matrix mdst, matrix msrc)
{
	if (mdst->row != msrc->row || mdst->col != msrc->col) {
		matrix_destroy(mdst);
		mdst->row = msrc->row;
		mdst->col = msrc->col;
		mdst->data = malloc(sizeof(int *) * mdst->row);
		for (int i = 0; i < mdst->row; i++) {
			mdst->data[i] = malloc(sizeof(int) * mdst->col);
		}
	}
	for (int i = 0; i < mdst->row; i++) {
		for (int j = 0; j < mdst->col; j++) {
			mdst->data[i][j] = msrc->data[i][j];
		}
	}
}

void extend_shortest_paths(matrix in,
			matrix weight,
			matrix out, matrix parent_in, matrix parent_out)
{
```

```
    int n = in->col;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            out->data[i][j] = in->data[i][j];
            parent_out->data[i][j] = parent_in->data[i][j];
            for (int k = 0; k < n; k++) {
                int dis = (in->data[i][k] == INT_MAX
                        || weight->data[k][j] ==
                        INT_MAX) ? INT_MAX : in->data[i][k]
                      + weight->data[k][j];
                if (dis < out->data[i][j]) {
                    out->data[i][j] = dis;
                    parent_out->data[i][j] =
                        parent_in->data[k][j];
                }
            }
        }
    }
}

void slow_all_pairs_shortest_paths(graph g, matrix out, matrix parent_out)
{
    int n = g->v_num;
    matrix weight = matrix_create(n, n);
    matrix parent_in = matrix_create(n, n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) {
                weight->data[i][j] = 0;
                parent_in->data[i][j] = i;
            } else {
                weight->data[i][j] =
                    g->adj[i][j] != 0 ? g->adj[i][j] : INT_MAX;
                parent_in->data[i][j] =
                    g->adj[i][j] != 0 ? i : -1;
            }
        }
    }
    matrix in = matrix_create(weight->row, weight->col);
    matrix_copy(in, weight);
    for (int m = 2; m <= n - 1; m++) {
        extend_shortest_paths(in, weight, out, parent_in, parent_out);
        matrix_copy(in, out);
        matrix_copy(parent_in, parent_out);
```

```
	}
	matrix_destroy(weight);
	matrix_destroy(parent_in);
	matrix_destroy(in);
}

void faster_all_pairs_shortest_paths(graph g, matrix out, matrix parent_out)
{
	int n = g->v_num;
	matrix weight = matrix_create(n, n);
	matrix parent_in = matrix_create(n, n);
	for (int i = 0; i < n; i++) {
		for (int j = 0; j < n; j++) {
			if (i == j) {
				weight->data[i][j] = 0;
				parent_in->data[i][j] = i;
			} else {
				weight->data[i][j] =
					g->adj[i][j] != 0 ? g->adj[i][j] : INT_MAX;
				parent_in->data[i][j] =
					g->adj[i][j] != 0 ? i : -1;
			}
		}
	}
	matrix in = matrix_create(weight->row, weight->col);
	matrix_copy(in, weight);
	for (int m = 1; m < n - 1; m *= 2) {
		extend_shortest_paths(in, in, out, parent_in, parent_out);
		matrix_copy(in, out);
		matrix_copy(parent_in, parent_out);
	}
	matrix_destroy(weight);
	matrix_destroy(parent_in);
	matrix_destroy(in);
}

void print_all_pairs_shortest_path(graph g, matrix parent, int i, int j)
{
	if (i == j) {
		printf("%s ", g->vertex_array[i].str_vertex);
	} else {
		if (parent->data[i][j] == -1) {
			printf("no path from %s to %s exist\n",
					g->vertex_array[i].str_vertex,
```

```
                        g->vertex_array[j].str_vertex);
            } else {
                print_all_pairs_shortest_path(g, parent, i,
                                        parent->data[i][j]);
                printf("%s ", g->vertex_array[j].str_vertex);
            }
        }
}

int main()
{
    //数据根据书上的图 22-1
    char *str_vertex[5] = { "1", "2", "3", "4", "5" };
    graph g = graph_create(5, str_vertex);
    struct edge edges[] =
        { {0, 1, 3}, {0, 2, 8}, {0, 4, -4}, {1, 4, 7}, {1, 3, 1}, {2, 1, 4},
    {3, 2, -5}, {3, 0, 2}, {4, 3, 6}
    };
    for (unsigned i = 0; i < sizeof(edges) / sizeof(edges[0]); i++) {
        graph_insert_edge(g, edges[i]);
    }
    graph_display(g);
    matrix out = matrix_create(g->v_num, g->v_num);
    matrix parent = matrix_create(g->v_num, g->v_num);
    slow_all_pairs_shortest_paths(g, out, parent);
    printf("慢速算法的最短路径矩阵:\n");
    matrix_display(out);
    printf("慢速算法的前驱矩阵:\n");
    matrix_display(parent);
    int i = 4;
    int j = 1;
    printf("path from %s to %s\n", str_vertex[i], str_vertex[j]);
    print_all_pairs_shortest_path(g, parent, i, j);
    printf("\n");
    faster_all_pairs_shortest_paths(g, out, parent);
    printf("快速算法的最短路径矩阵:\n");
    matrix_display(out);
    printf("快速算法的前驱矩阵:\n");
    matrix_display(parent);
    printf("path from %s to %s\n", str_vertex[i], str_vertex[j]);
    print_all_pairs_shortest_path(g, parent, i, j);
    printf("\n");
    matrix_destroy(out);
    matrix_destroy(parent);
```

```
    graph_destroy(g);
    return 0;
}
```