# 数据结构第三周实验报告

2018 软件工程创新班 邰思童 2018204493 / 2019/11/22

## 实验题目

栈 / 队列

## 实验环境

Windows 10

gcc (tdm64-2) 4.8.1

std=c99

## 栈

### 实验内容

实现一种后进先出（LIFO）的数据结构（栈），支持从栈顶插入，弹出栈顶元素，判断栈是否为空，获取栈的大小等操作。支持栈大小的动态修改，当栈已满的情况出现时，栈容量自动扩展为原先的两倍。

### 算法流程

栈顶指针总是指向栈顶元素之上的一个位置，弹出操作只需要下移栈顶指针即可，插入操作先插入值再修改栈顶指针。

### 程序代码

#ifndef STACK\_H  
#define STACK\_H  
  
typedef int element\_type;  
  
typedef struct Stack {  
 element\_type \*stack; //一个不定长的数组，表示栈本身  
  
 int top; //栈顶的位置，指向栈顶元素的下一个位置  
 int capacity; //栈的容量  
} Stack;  
  
Stack\* newStack(int size); //构造一个新的栈，初始大小为 size  
void push(Stack \*s, element\_type x); //向栈顶压入一个元素  
void pop(Stack \*s); //弹出栈顶元素  
element\_type top(Stack \*s); //获取栈顶元素的值  
int empty(Stack \*s); //判断栈是否为空，1表示空，0表示非空  
int size(Stack \*s); //获取当前栈的大小（栈内的元素个数）  
  
#endif

//stack.c  
#include <stdlib.h>  
#include <string.h>  
#include <stdio.h>  
  
#include "stack.h"  
  
typedef int element\_type;  
  
Stack\* newStack(int size) {  
 Stack \*ret = (Stack\*)malloc(sizeof(Stack));  
 ret->capacity = size;  
 ret->top = 0;  
 ret->stack = (element\_type\*)malloc(size \* sizeof(element\_type));  
  
 return ret;  
}  
  
int size(Stack \*s) {  
 return s->top;  
}  
  
int empty(Stack \*s) {  
 return s->top == 0;  
}  
  
void push(Stack \*s, element\_type x) {  
 if (s->top == s->capacity - 1) { //若栈已满，则将栈的空间扩大至原来的两倍  
 element\_type \*tmp = (element\_type\*)malloc(s->capacity \* 2 \* sizeof(element\_type));  
 memcpy(tmp, s->stack, sizeof(element\_type) \* s->capacity);  
  
 s->stack = tmp;  
 s->capacity \*= 2;  
 }  
  
 s->stack[s->top++] = x;  
}  
  
void pop(Stack \*s) {  
 if (empty(s)) printf("The Stack is empty\n");  
 else {  
 s->top--;  
 }  
}  
  
 top(Stack \*s) {  
 if (empty(s)) printf("The Stack is empty\n");  
 else return s->stack[s->top - 1];  
}

//main.c  
#include <stdio.h>  
  
#include "stack.h"  
  
#define MAXN 100  
  
int main() {  
 Stack \*stack = newStack(10);  
 for (int i = 0; i < 10; i++) {  
 push(stack, i);  
 }  
  
 while (!empty(stack)) {  
 printf("%d ", top(stack));  
 pop(stack);  
 }  
 return 0;  
}

### 运行结果

λ main  
9 8 7 6 5 4 3 2 1 0

### 结果分析

从0到9向栈内插入十个数，再将他们依次弹出。

## 队列

### 实验内容

实现一种先入先出（FIFO）的数据结构（队列），支持从队尾插入，从队头弹出，判断队列是否为空，获取队头元素等操作。

具体实现使用了单链表，避免了令人迷惑的循环队列中怪异的取模运算。

### 算法流程

使用队列实现了一个简单的无向图的广度优先遍历（BFS）算法。

从任意点开始作为起点，将其放入队列中，每一次从队列中弹出一个点并将所有与这个点邻接的点放入队列，直到所有点都被访问过，其出队的顺序就是广度优先遍历（BFS）的顺序。

### 程序代码

//Queue.h  
#ifndef QUEUE\_H  
#define QUEUE\_H  
  
typedef struct qNode {  
 int data;  
 struct qNode \*next;  
} qNode;  
  
typedef struct Queue {  
 qNode \*head, \*tail;  
} Queue;  
  
Queue \*newQueue();  
qNode \*newNode(int data);  
int empty(Queue \*q);  
int front(Queue \*q);  
void push(Queue \*q, int data);  
void pop(Queue \*q);  
  
#endif

//queue.c  
#include <stdio.h>  
#include <stdlib.h>  
  
#include "queue.h"  
  
Queue\* newQueue() {  
 Queue \*newQ = (Queue\*)malloc(sizeof(Queue));  
  
 if (newQ == NULL) exit(0);  
  
 newQ->head = newQ->tail = NULL;  
}  
  
qNode \*newNode(int data) {  
 qNode \*newN = (qNode\*)malloc(sizeof(qNode));  
  
 if (newN == NULL) exit(0);  
  
 newN->data = data;  
 newN->next = NULL;  
}  
  
int empty(Queue \*q) {  
 return (!q->head && !q->tail);  
}  
  
int front(Queue \*q) {  
 if (empty(q)) printf("Error\n");  
 else {  
 return q->head->data;  
 }  
}  
  
void push(Queue \*q, int data) {  
 qNode \*newN = newNode(data);  
 if (!q->tail) q->head = q->tail = newN;  
 else {  
 q->tail->next = newN;  
 q->tail = newN;  
 }  
}  
  
void pop(Queue \*q) {  
 if (!q->head) printf("Error! Queue is empty\n");  
 else {  
 qNode \*old = q->head;  
   
 if (q->head == q->tail) q->head = q->tail = old->next;  
 else q->head = old->next;  
   
 free(old);  
 old = NULL;  
 }  
}

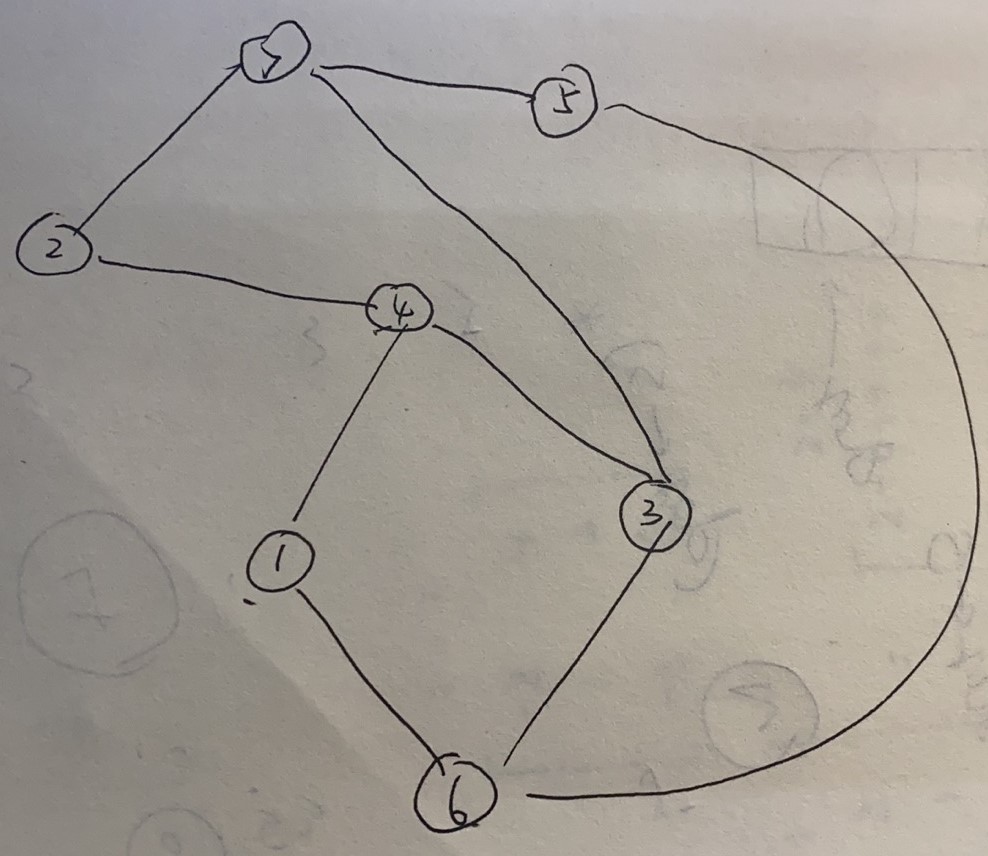
//main.c  
#include <stdio.h>  
#include <stdlib.h>  
  
#include "queue.h"  
  
#define MAXN 100 + 10  
  
struct Node;  
struct Edge;  
  
typedef struct Node {  
 int id;  
 int vis;  
 struct Edge \*edges;  
} Node;  
  
Node nodes[MAXN];  
  
typedef struct Edge {  
 Node \*fr, \*to;  
 struct Edge \*next;  
} Edge;  
  
Edge \*new\_edge(Node \*u, Node \*v) {  
 Edge \*ret = (Edge\*)malloc(sizeof(Edge));  
 ret->fr = u;  
 ret->to = v;  
 ret->next = u->edges;  
  
 return ret;  
}  
  
void add\_edge(int u, int v) {  
 nodes[u].edges = new\_edge(&nodes[u], &nodes[v]);  
 nodes[v].edges = new\_edge(&nodes[v], &nodes[u]);  
}  
  
void bfs(int x) {  
 Queue \*q = newQueue();  
 push(q, x);  
  
 while (!empty(q)) {  
 int v = front(q);  
 pop(q);  
  
 if (nodes[v].vis) continue;  
 nodes[v].vis = 1;  
 printf("%d ", v);  
  
 for (Edge \*e = nodes[v].edges; e; e = e->next) {  
 push(q, e->to->id);  
 }  
 }  
}  
  
int main() {  
 int n, m;  
 scanf("%d%d", &n, &m);  
  
 for (int i = 1; i <= n; i++) {  
 nodes[i].id = i;  
 }  
  
 for (int i = 0; i < m; i++) {  
 int u, v;  
 scanf("%d%d", &u, &v);  
 add\_edge(u, v);  
 }  
  
 bfs(1);  
 return 0;  
}  
  
// 7 11  
// 2 4  
// 1 4  
// 7 2  
// 3 4  
// 5 7  
// 7 3  
// 6 1  
// 6 3  
// 2 4  
// 5 6  
// 7 2

### 运行结果

λ main  
7 11  
2 4  
1 4  
7 2  
3 4  
5 7  
7 3  
6 1  
6 3  
2 4  
5 6  
7 2  
1 6 4 5 3 2 7

### 结果分析

样例中的图大概这样



从一号点开始进行广度优先遍历（BFS），1号点先进队，出队时4和6两个点入队，6出队时3和5两个点入队，4出队时，由于3和1都在之前被访问过，只把2入队，3出队把7入队，至此所有点都被访问过了，输出队列中的元素即可。

## 注意事项