```c
/*
 * An implementation of Kosaraju's algorithm for the discovery of strongly
connected components.
 * Seehttp://www.giaithuatlaptrinh.com/?p=1680 for more details.
 *
 *  Created on: Dec 16, 2016
 *      Author: hunglv
 */

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

#define UNVISITED 0
#define VISITED 1
#define TRUE 1
#define FALSE 0
#define INFTY 1000000



// the vertex list data structure, it is a linked list
typedef struct vlist{
        int v;          // vertex v is adjacent to the index of the list
        struct vlist *next;
}vlist;

// the list representation of graph
typedef struct Graph{
        vlist **adjList;
        int n;          // the number of veritces
        int m;          // the number of edges
} Graph;
Graph *build_graph();
void  add_arc(Graph *G, int u, int v);
void print_list_graph(Graph *G);
void print_vertex_list(vlist *vertex_list);


typedef struct Stack {
        int *storage;
        int top;
} Stack;
Stack *init_stack(int capacity);
void push(Stack *S, int elem);  // push an element to the stack
int pop(Stack *S);              // take (and remove) an element from the queue
int is_stack_empty(Stack *S);
```

```
void Kosaraju(Graph *G);
void DFS(Graph *G, int i, Stack *S);
Graph *reverse(Graph *G);
void print_connected_component(Graph *H, Stack *S);
void dfs_and_print(int v, Graph *H);

int main(){
	Graph *G = build_graph();
	Kosaraju(G);
	return 0;
}

int* mark;		// an array to mark visited vertices
void Kosaraju(Graph *G){
	Stack *S = init_stack(G->n);
	mark = (int *)malloc(G->n*sizeof(int));
	memset(mark, UNVISITED, G->n*sizeof(int));
	int i = 0;
	for(i = 0; i < G->n; i++){
		if(mark[i] == UNVISITED){
			DFS(G, i, S);
		}
	}

	Graph *H = reverse(G);
	print_connected_component(H, S);
}

void DFS(Graph *G, int i, Stack *S){
	mark[i] = VISITED;
	vlist *arcs = G->adjList[i]; // the adjacent list of vertex i
	while(arcs != NULL){
		if(mark[arcs->v] == UNVISITED){
			DFS(G, arcs->v, S);
		}
		arcs = arcs->next;
	}
	push(S,i);	// push i to stack S after visited
}

int *Dm;	// an array to mark deleted vertices from reversed graph H and
Stack S
void print_connected_component(Graph *H, Stack *S){
	Dm = (int *)malloc(H->n*sizeof(int));
	memset(Dm, FALSE, H->n*sizeof(int));
	int c = 1;
	while(!is_stack_empty(S)){
		int v = pop(S);
		if(!Dm[v]){
```

```
			memset(mark, UNVISITED, H->n*sizeof(int));	// mark
all vertices of H UNVISITED
			printf("Strongly Connected Component #%d: ", c);
			dfs_and_print(v, H);
			printf("\n");
			c++;
		}
	}
}

// use dfs to list a set of vertices dfs_and_print from a vertex v in H
void dfs_and_print(int v, Graph *H){
		printf("%d,",v);
		mark[v] = VISITED;
		Dm[v] = TRUE;
		vlist *arcs = H->adjList[v]; // the adjacent list of vertex v
		while(arcs != NULL){
			int u = arcs->v;
			if(mark[u] == UNVISITED && Dm[u] == FALSE){
				dfs_and_print(u, H);
			}
			arcs = arcs->next;
		}
}
// build the reversed graph
Graph *reverse(Graph *G){
		Graph *H = (Graph *)malloc(sizeof(Graph));
		H->n = G->n;
		H->m = G->m;
		H->adjList = (vlist **)malloc(H->n*sizeof(vlist *));
		int i = 0;
		for(i = 0; i < H->n; i++){
			H->adjList[i] = NULL;
		}
		// reverse arcs of G
		vlist *arcs;
		for(i = 0; i < G->n; i++){
			arcs = G->adjList[i];
			while(arcs != NULL){
				add_arc(H, arcs->v, i);
				arcs = arcs->next;
			}
		}
		return H;
}

// build an instance of a graph
Graph *build_graph(){
	Graph *G = (Graph *)malloc(sizeof(Graph));
```

```c
	G->n = 9;
	G->m = 16;
	G->adjList = (vlist **)malloc(G->n*sizeof(vlist *));
	int i = 0;
	for(i = 0; i < G->n; i++){
		G->adjList[i] = NULL;
	}
	add_arc(G,0,1);
	add_arc(G,0,2);
	add_arc(G,0,4);
	add_arc(G,1,3);
	add_arc(G,2,4);
	add_arc(G,2,5);
	add_arc(G,3,0);
	add_arc(G,3,2);
	add_arc(G,3,7);
	add_arc(G,3,8);
	add_arc(G,4,5);
	add_arc(G,5,6);
	add_arc(G,5,8);
	add_arc(G,6,4);
	add_arc(G,8,2);
	add_arc(G,8,7);
	return G;
}

// add an u->v arc to the graph G
void  add_arc(Graph *G, int u, int v){
		// edd v to the head of the  vertex list of u
		vlist *arc = malloc(sizeof(vlist));
		arc->v = v;
		arc->next = G->adjList[u];
		G->adjList[u] = arc;
	}

void print_vertex_list(vlist *vertex_list){
		while(vertex_list != NULL){
			printf("%d ", vertex_list->v);
			vertex_list = vertex_list->next;
		}
		printf("\n");
}

void print_list_graph(Graph *G){
	printf("num vertices: %d\n", G->n);
	printf("num edges: %d\n", G->m);
	int i = 0;
	for(i = 0; i < G->n; i++){
		printf("adj list of %d : ",i);
```

```
                print_vertex_list(G->adjList[i]);
        }

}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
////////////////
////////////////        THE STACK INTERFACES
////////////////
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////
Stack *init_stack(int capacity){
        Stack *S = (Stack *)malloc(sizeof(Stack));
        S->top = -1;
        S->storage = (int*)malloc(capacity*sizeof(int));
        return S;
}

void push(Stack *S, int elem){
        S->top++;
        S->storage[S->top] = elem;
}
int pop(Stack *S){
        if(is_stack_empty(S)){
                printf("nothing to pop\n");
                exit(0);
        }
        int elem = S->storage[S->top];
        S->top--;
        return elem;
}

int is_stack_empty(Stack *S){
        return S->top < 0 ? TRUE: FALSE;
}
```