Lists, Stacks and Queues in C

CHAN Hou Pong, Ken

CSCI2100A Data Structures Tutorial 4

Outline

- Structure
- Linked List
  - Overview
  - Implementation
- Stack
  - Overview
  - Implementation
- Queue
  - Overview
  - Implementation

Structure

- A collection of values (members)
  - Like a class in java or c++, but without methods and access controls (private, protected, public.)

```
struct time
{
    int hh;
    int mm;
    int ss;
};
…
struct time t1;
t1.hh = 20;
t1.mm=12;
t1.ss=30;
printf(“The time is %d:%d:%d”, t1.hh, t1.mm, t1.ss);
//Output: The time is 20:12:30
```

Pointer to structure

- Define a structure and declare a variable

```
struct time
{
    int hh;
    int mm;
    int ss;
};
struct time t1;
t1.hh=20;
```

- Declare a pointer to struct time

```
struct time* t1_ptr;
```

- Store the address of a struct time variable in the pointer variable

```
t1_ptr = &t1;
```

- Access the member of the structure

```
t1_ptr->hh=21;  //same as (*t1_ptr).hh=21;
```

Some words about **typedef**

- Allow us to define alias for a data type:
  - *typedef int* My_integer_type;
  - My_interger_type x =3; //same as int x = 3;
- **typedef** can be used for **structures**:

```
struct time
{
    int hh;
    int mm;
    int ss;
};
struct time t1;
t1.hh=20;
```

```
typedef struct time
{
    int hh;
    int mm;
    int ss;
}Time_type;
Time_type t1;
t1.hh = 12;
```

Dynamic Memory Allocations

- We can allocate memory at run time using malloc
  - malloc can be used to allocate a piece of memory of the specified size, and returns a pointer to it.
- Example:

```
Time_type *t1;
t1 = (Time_type*)malloc(sizeof(Time_type));
```

  - Allocate enough memory for storing a Time_type variable (which is a structure).
  - Return a pointer to it.
  - Cast it to a pointer to Time_type, and assign it to t1.

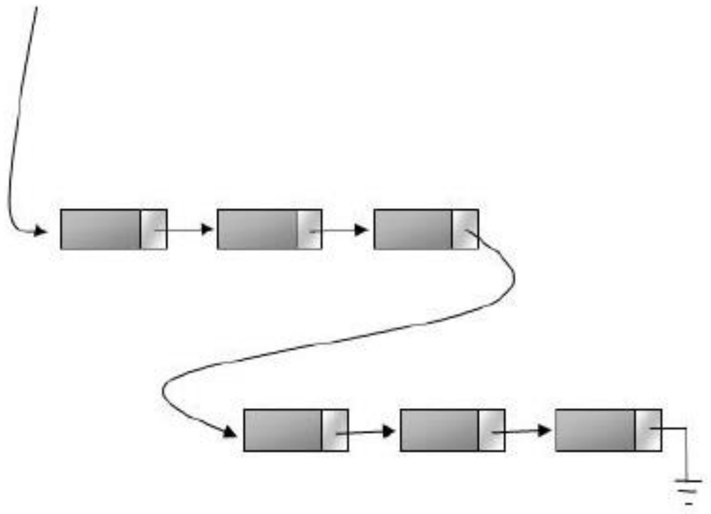
Dynamic Memory Allocations

- Use free to de-allocate the memory when it is no longer needed.
- This is important because there is *no garbage collection* in C. So you will run out of memory if you keep allocating without de-allocating. (“Memory Leak”)

```
Time_type *t1;
t1 = (Time_type*)malloc(sizeof(Time_type));
…
t1->hh = 12;
…
free(t1); //de-allocate when we no longer need it.
```

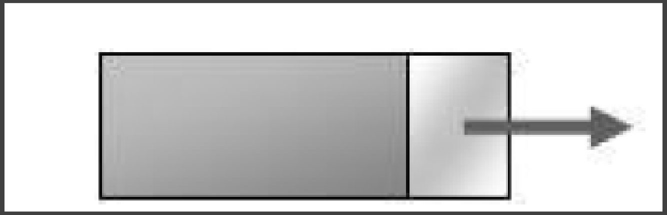
Linked List Overview

- A list of structures (nodes)
- Each structure contains
  - Element (to store data)
  - Pointer to next structure
- Insert()
- Delete()
- Print()



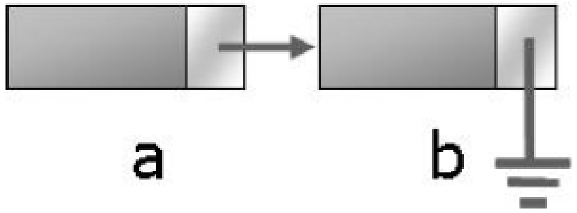
Linked List Implementation

```
struct node_s
{
    int data;
    struct node_s *next;
};
typedef struct node_s node;
//To create a node variable
node anode; //static, allocate in compile time
//or dynamic allocation
node *anode = (node*)malloc(sizeof(node));
```



Linked List Implementation

- Link two nodes together



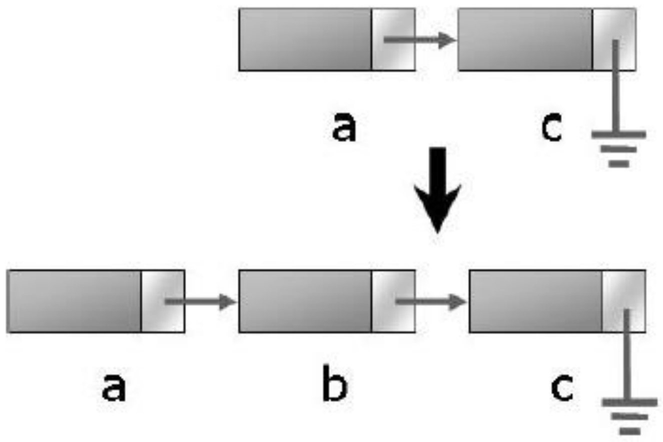
```
node a,b;
a.next = &b;
b.next = NULL;

//use pointer
node* a, *b;
a = (node*)malloc(sizeof(node));
b = (node*)malloc(sizeof(node));
b->next = NULL;
a->next = b;
```

Linked List Implementation

- Insert a node to a list

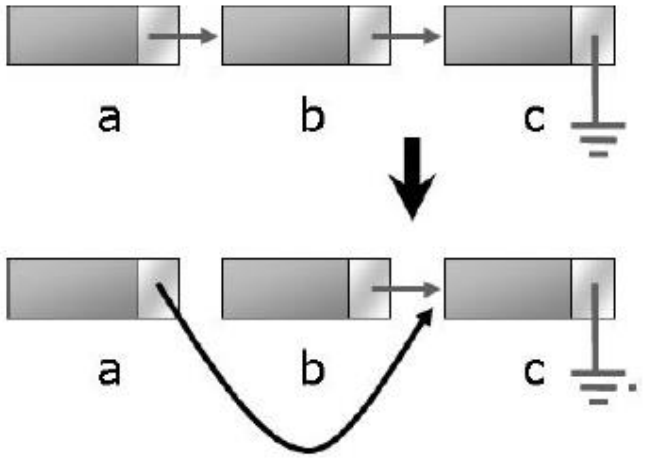
```
node a,b,c;
c.next = NULL;
//originally, only a and c
a.next = &c;
//insert b between a and c
b.next = &c;
a.next = &b;
```



Linked List Implementation

- Delete a node from a list

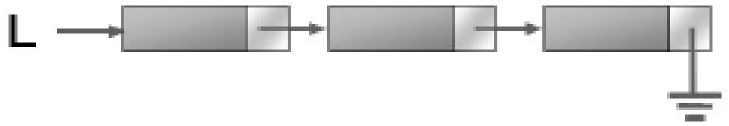
```
node a,b,c;
c.next = NULL;
//orginial
a.next = &b;
b.next = &c;
//remove b from the list
a.next = &c;
```



```
struct node_s {
  int data;
  struct node_s *next;
};
typedef struct node_s node;

// Create a list first
node *L = (node *)malloc(sizeof(node));
node *p;
L->data = 0;
p = L;
for (x=1 ; x<=num ; x++){
  p->next = (node *)malloc(sizeof(node));
  p = p->next;
  p->data = x;
}
p->next = NULL;

//And then print it
p = L;
while (p != NULL) {
  printf("%d ", p->data);
  p = p->next;
}
putchar('\n');
```

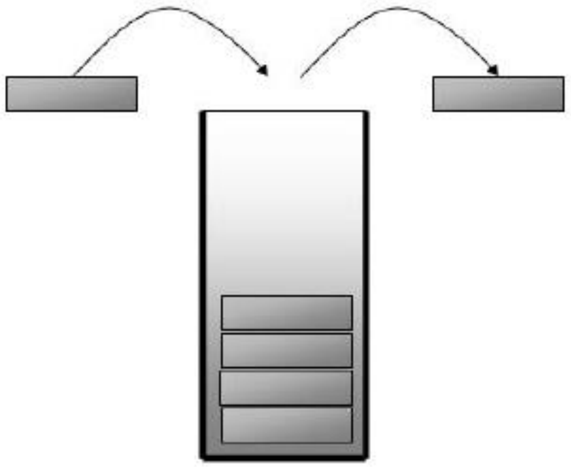


Stack

books on your table?

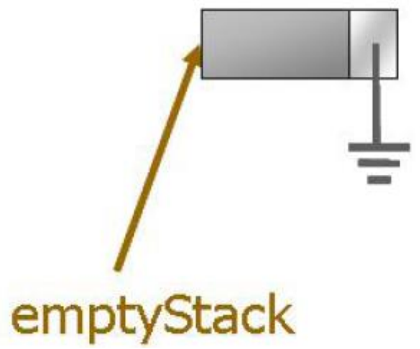
Stack Overview

- Last In First Out (LIFO)
- Push()
- Pop()
- Top()
- is_empty()
- Can be implemented by **linked list** or **array**



Stack Implementation using linked list

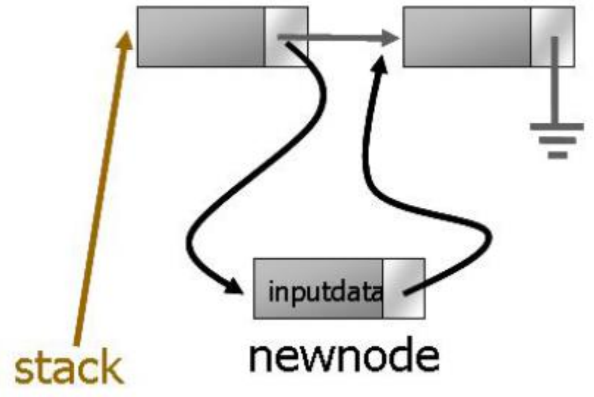
- Create an empty stack



```
node *create_stack()
{
    node* emptyStack;
    emptyStack = (node*)malloc(sizeof(node));
    emptyStack->next = NULL;
    return emptyStack;
}
```

Stack Implementation using linked list

- Push an entry into the stack

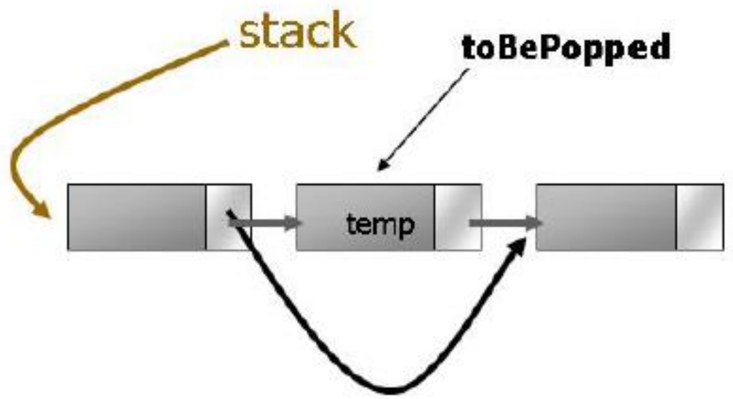


```
void Push(int inputdata, node *stack)
{
    node* newnode = (node*)malloc(sizeof(node));
    newnode->data = inputdata;
    newnode->next = stack->next; //should be first
    stack->next = newnode;
    //how about change the above 2 lines?
}
```

Stack Implementation

- Pop an entry from the stack

```
int Pop(node* stack)
{
    int temp;
    node* toBePopped;
    if(stack->next!=NULL)
    {
        temp = stack->next->data;
        toBePopped = stack->next;
        stack->next = stack->next->next;
        free(toBepopped);
        return temp;
    }
    else
        return 0; //error code, you can define according to the demand
}
```



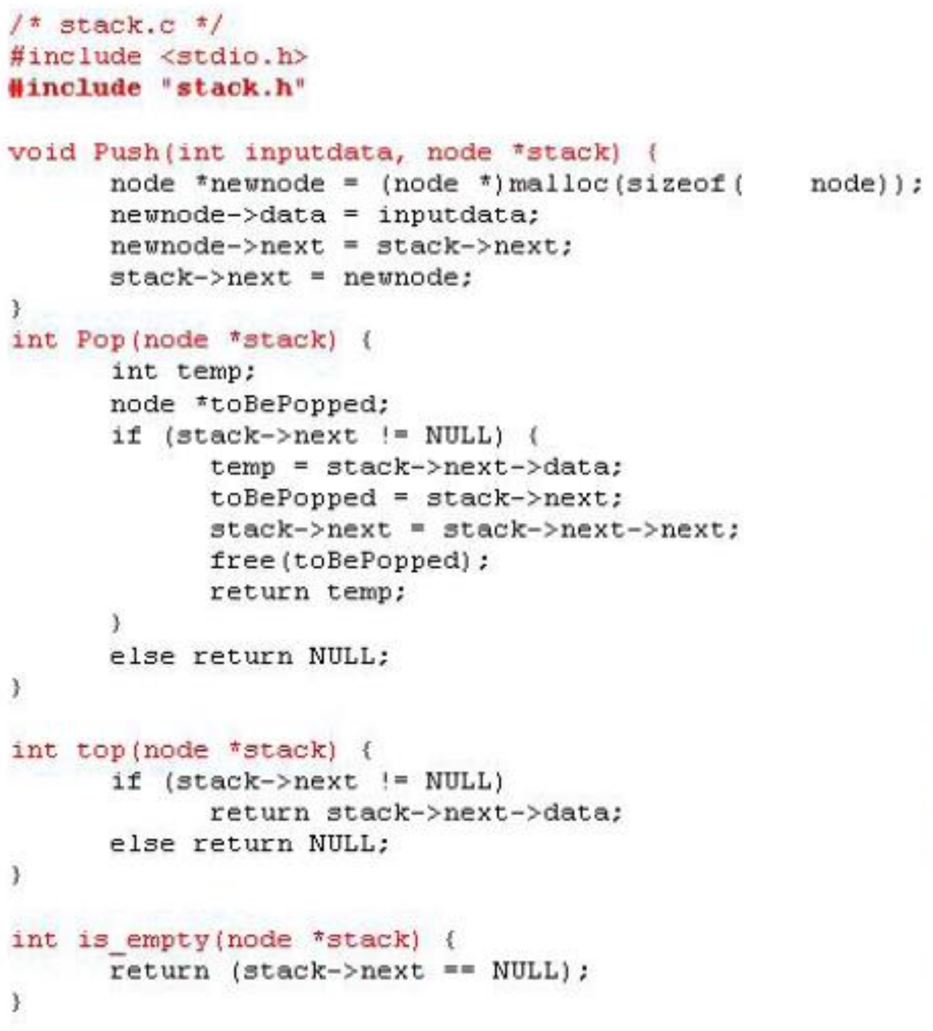
Stack Implementation

- Return the top element in the stack

```
int top(node* stack)
{
    if(stack->next!=NULL)
        return stack->next->data;
    else
        return 0;
}

//Determine if the stack is empty
int is_empty(node *stack)
{
    return (stack->next==NULL);
}
```

If you are implementing a large project, usually we can separate the declaration and implementation to .h and .c files.

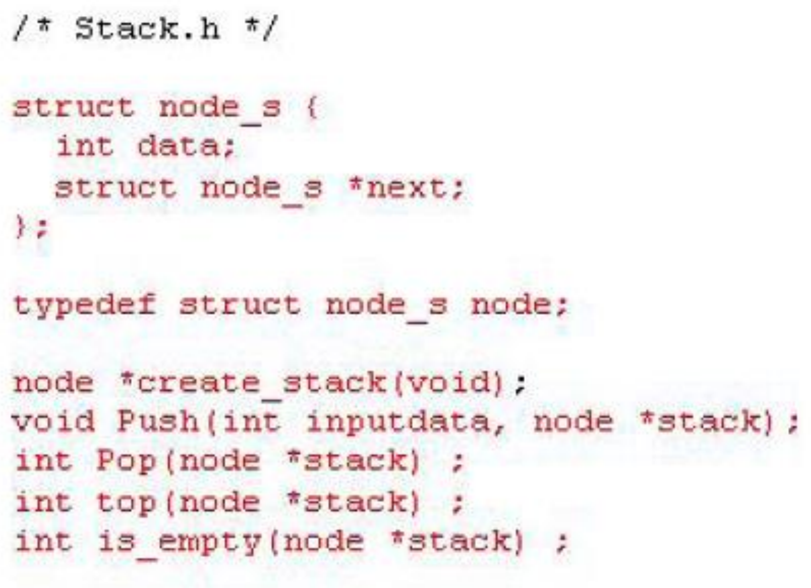


```c
/* stack.c */
#include <stdio.h>
#include "stack.h"

void Push(int inputdata, node *stack) {
        node *newnode = (node *)malloc(sizeof(    node));
        newnode->data = inputdata;
        newnode->next = stack->next;
        stack->next = newnode;
}
int Pop(node *stack) {
        int temp;
        node *toBePopped;
        if (stack->next != NULL) {
                temp = stack->next->data;
                toBePopped = stack->next;
                stack->next = stack->next->next;
                free(toBePopped);
                return temp;
        }
        else return NULL;
}

int top(node *stack) {
        if (stack->next != NULL)
                return stack->next->data;
        else return NULL;
}

int is_empty(node *stack) {
        return (stack->next == NULL);
}
```



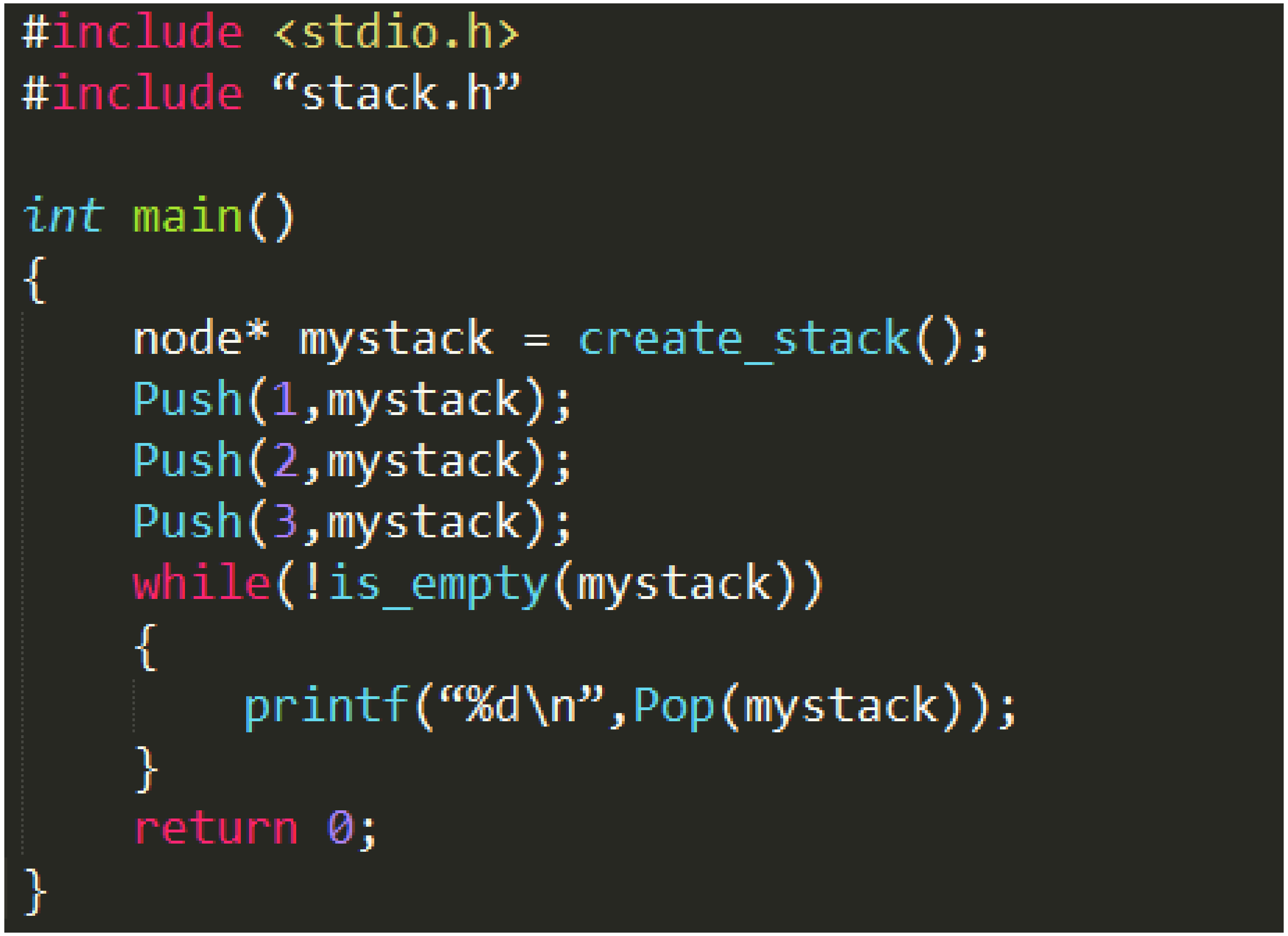
```c
/* Stack.h */

struct node_s {
   int data;
   struct node_s *next;
};

typedef struct node_s node;

node *create_stack(void);
void Push(int inputdata, node *stack);
int Pop(node *stack) ;
int top(node *stack) ;
int is_empty(node *stack) ;
```

Stack Implementation

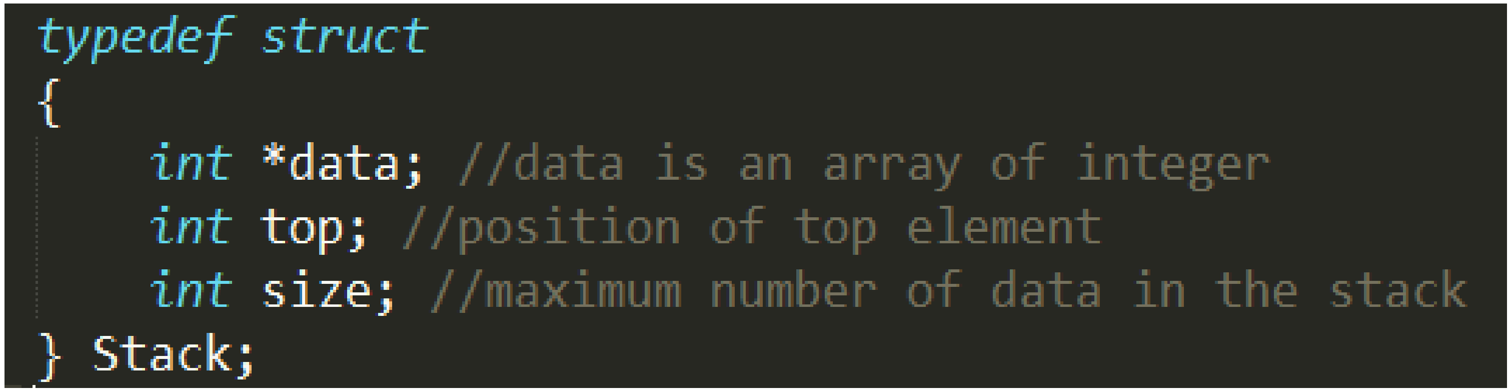


```c
#include <stdio.h>
#include “stack.h”

int main()
{
    node* mystack = create_stack();
    Push(1,mystack);
    Push(2,mystack);
    Push(3,mystack);
    while(!is_empty(mystack))
    {
        printf(“%d\n”,Pop(mystack));
    }
    return 0;
}
```

Stack Implementation using Array

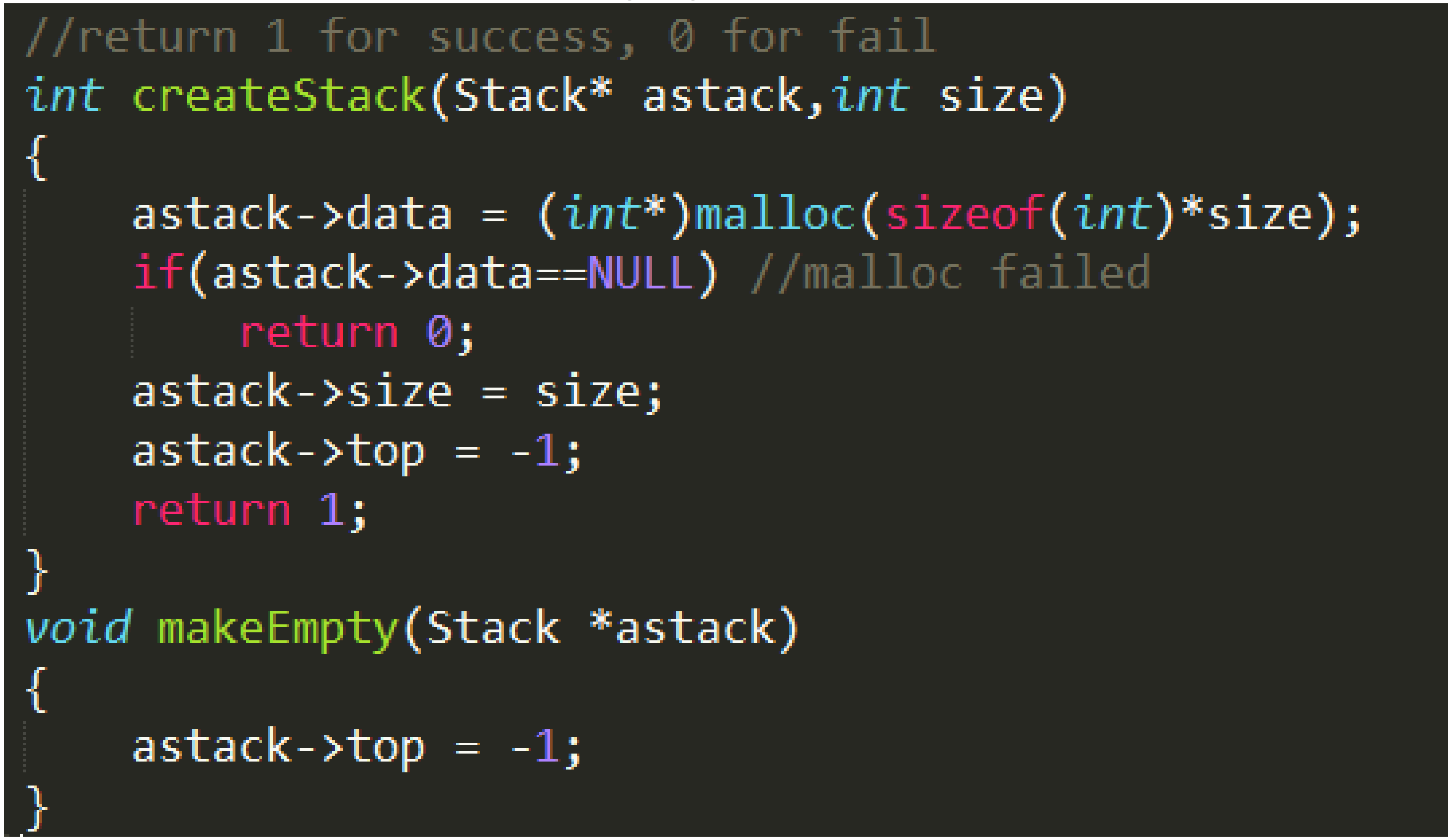
- Implement a stack using array



```
typedef struct
{
    int *data; //data is an array of integer
    int top; //position of top element
    int size; //maximum number of data in the stack
} Stack;
```

Stack Implementation using Array

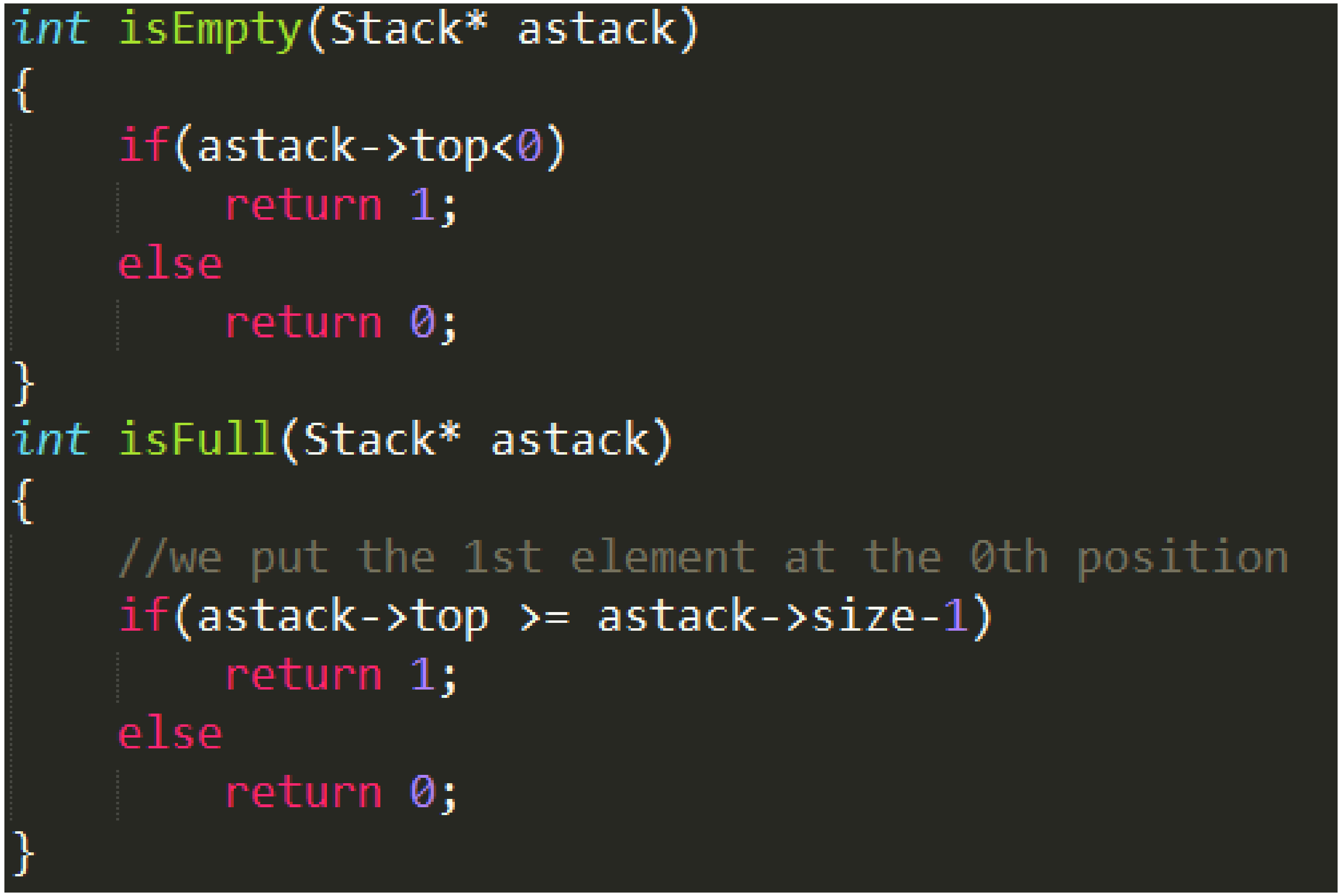
- createStack, makeEmpty



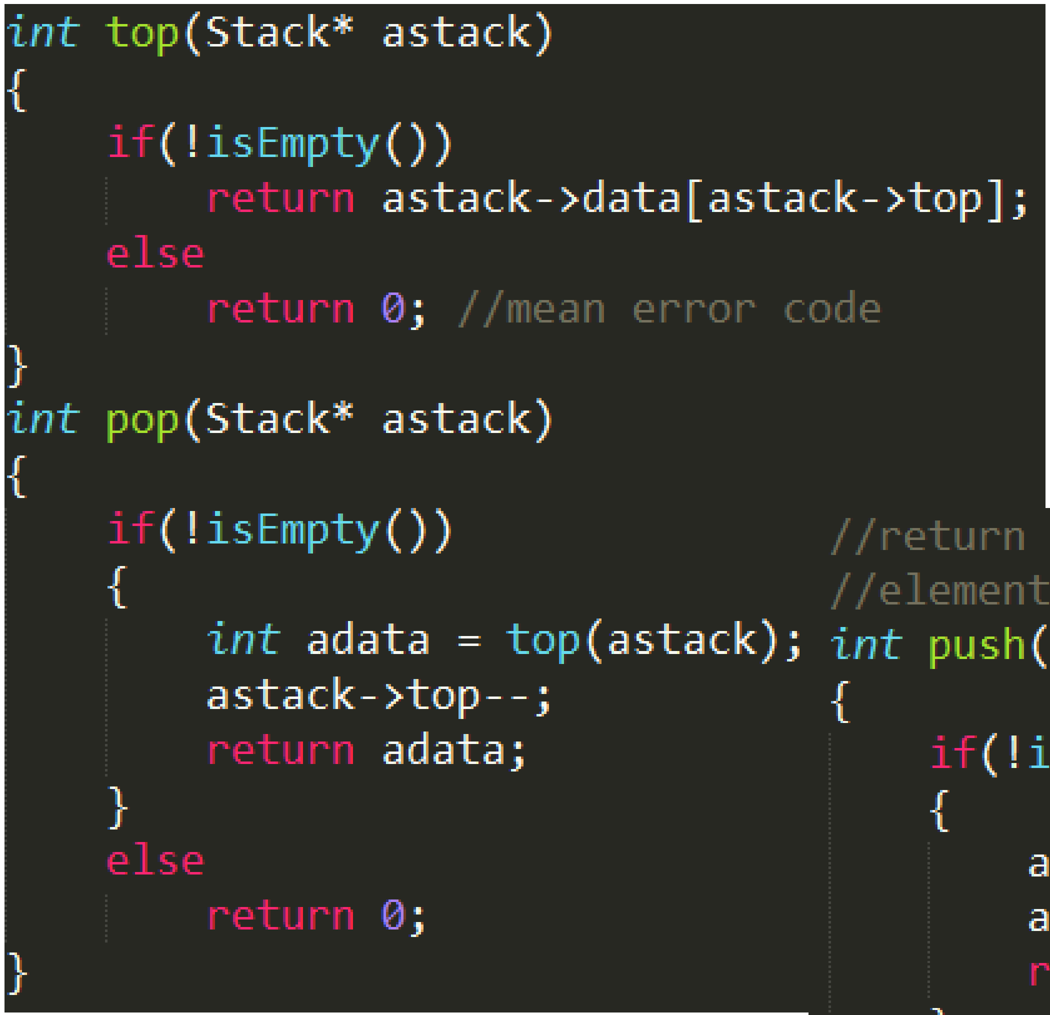
```c
//return 1 for success, 0 for fail
int createStack(Stack* astack,int size)
{
    astack->data = (int*)malloc(sizeof(int)*size);
    if(astack->data==NULL) //malloc failed
        return 0;
    astack->size = size;
    astack->top = -1;
    return 1;
}
void makeEmpty(Stack *astack)
{
    astack->top = -1;
}
```

Stack Implementation using Array

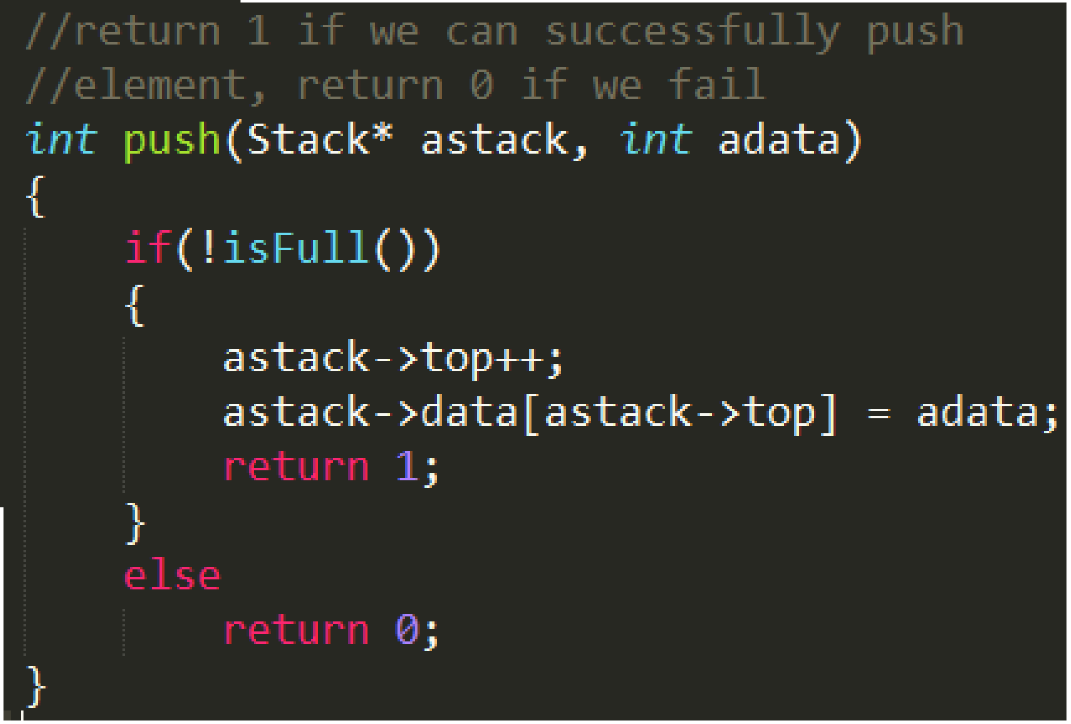
- isEmpty, isFull



```
int isEmpty(Stack* astack)
{
    if(astack->top<0)
        return 1;
    else
        return 0;
}
int isFull(Stack* astack)
{
    //we put the 1st element at the 0th position
    if(astack->top >= astack->size-1)
        return 1;
    else
        return 0;
}
```



```
int top(Stack* astack)
{
    if(!isEmpty())
        return astack->data[astack->top];
    else
        return 0; //mean error code
}
int pop(Stack* astack)
{
    if(!isEmpty())
    {
        int adata = top(astack);
        astack->top--;
        return adata;
    }
    else
        return 0;
}
```



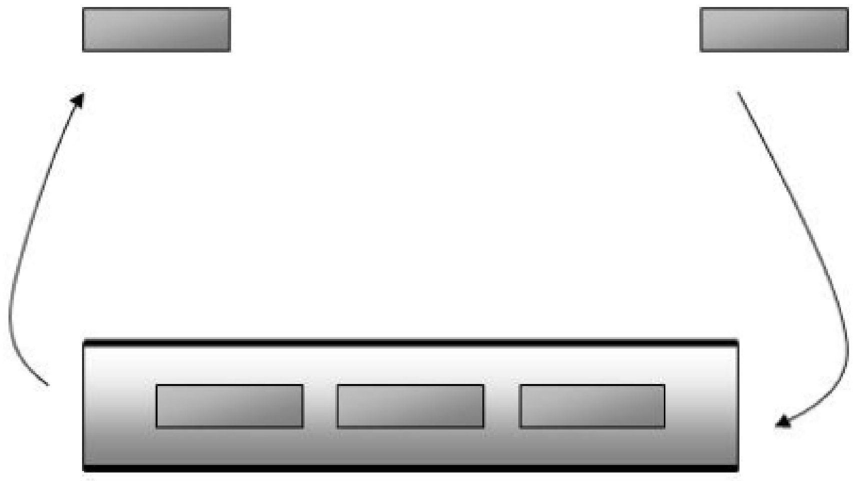
```
//return 1 if we can successfully push
//element, return 0 if we fail
int push(Stack* astack, int adata)
{
    if(!isFull())
    {
        astack->top++;
        astack->data[astack->top] = adata;
        return 1;
    }
    else
        return 0;
}
```

Queue

ticket office of the Ocean park?

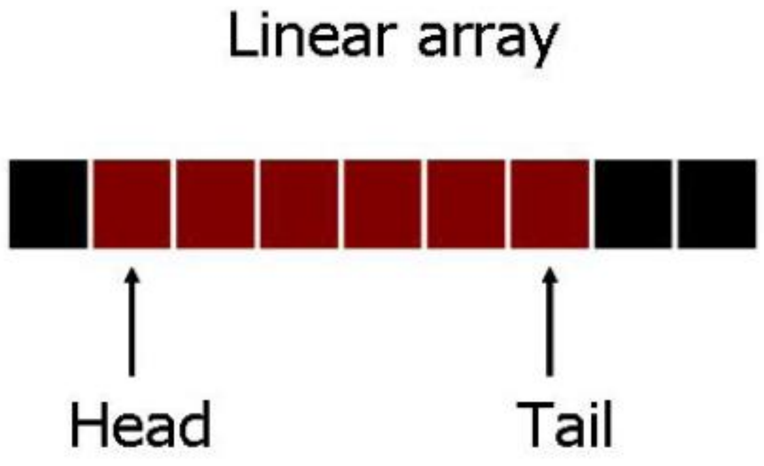
Queue Overview

- First In First Out (FIFO)
- Enqueue
- Dequeue



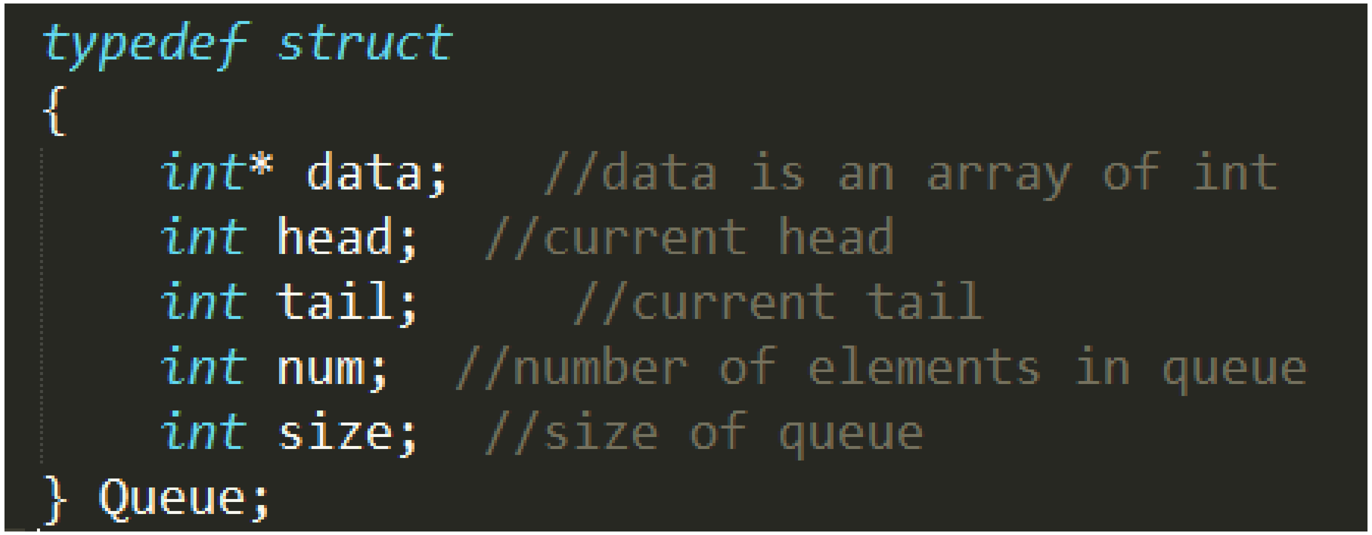
Queue Implementation

- A queue may be implemented using linked-list or array
- Implement a queue using array

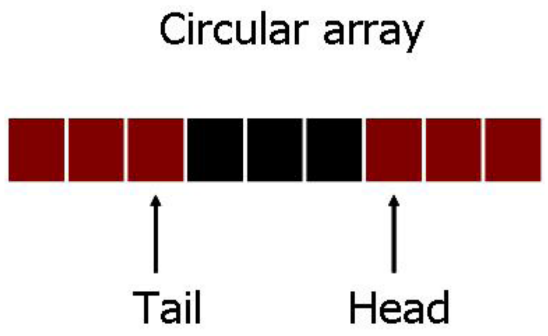


Queue Implementation

- Implementing a queue using **circular array**

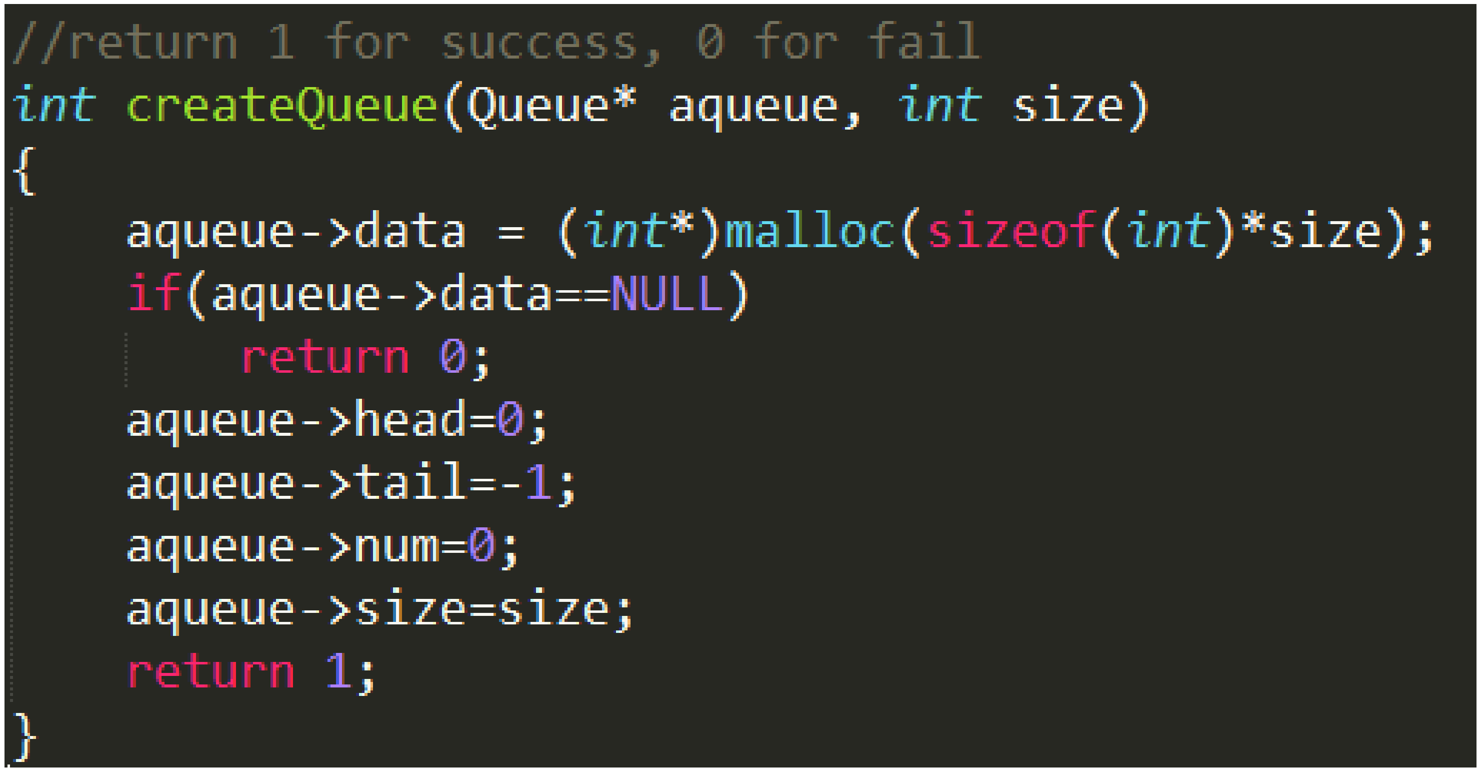


```
typedef struct
{
    int* data;    //data is an array of int
    int head;  //current head
    int tail;     //current tail
    int num;  //number of elements in queue
    int size;  //size of queue
} Queue;
```



Queue Implementation

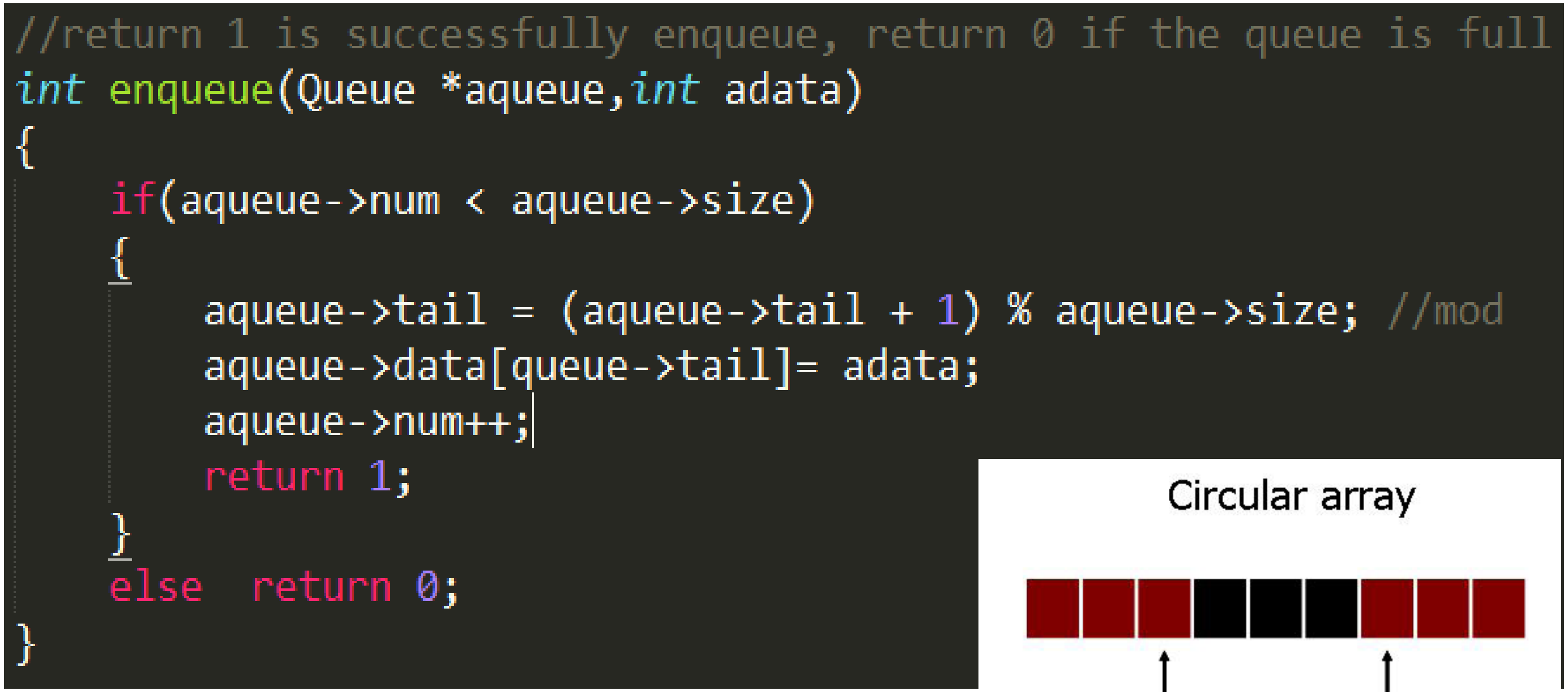
- **createQueue**



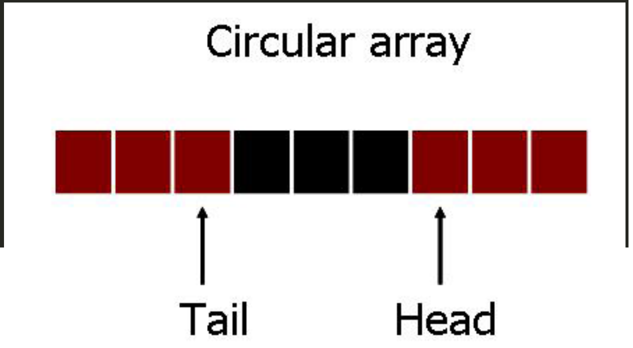
```
//return 1 for success, 0 for fail
int createQueue(Queue* aqueue, int size)
{
    aqueue->data = (int*)malloc(sizeof(int)*size);
    if(aqueue->data==NULL)
        return 0;
    aqueue->head=0;
    aqueue->tail=-1;
    aqueue->num=0;
    aqueue->size=size;
    return 1;
}
```

Queue Implementation

- **enqueue**

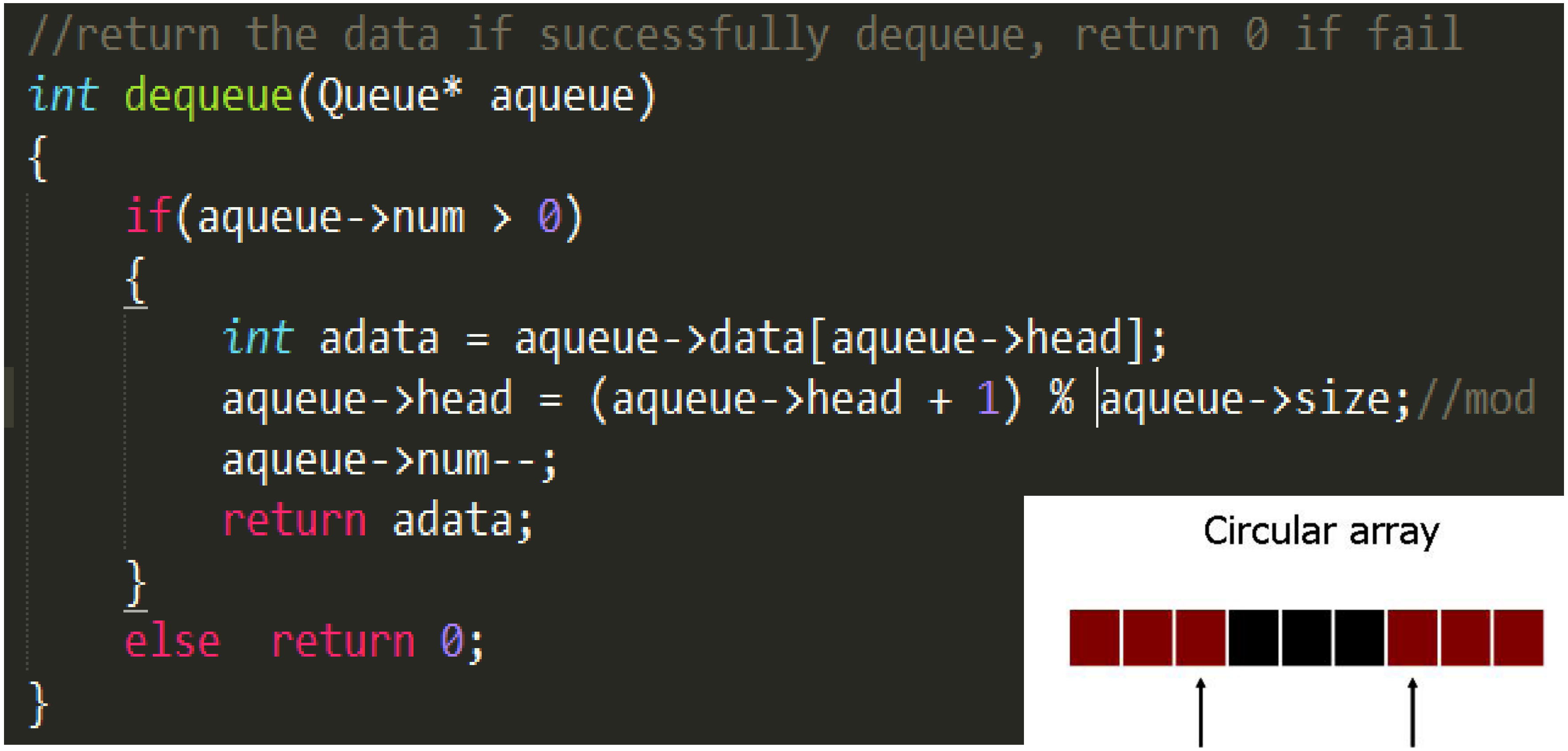


```
//return 1 is successfully enqueue, return 0 if the queue is full
int enqueue(Queue *aqueue,int adata)
{
    if(aqueue->num < aqueue->size)
    {
        aqueue->tail = (aqueue->tail + 1) % aqueue->size; //mod
        aqueue->data[queue->tail]= adata;
        aqueue->num++;
        return 1;
    }
    else  return 0;
}
```

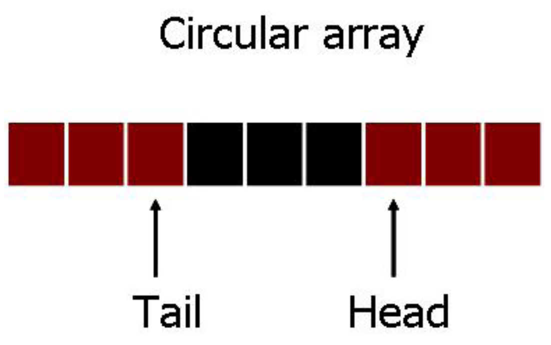


Queue Implementation

- **dequeue**

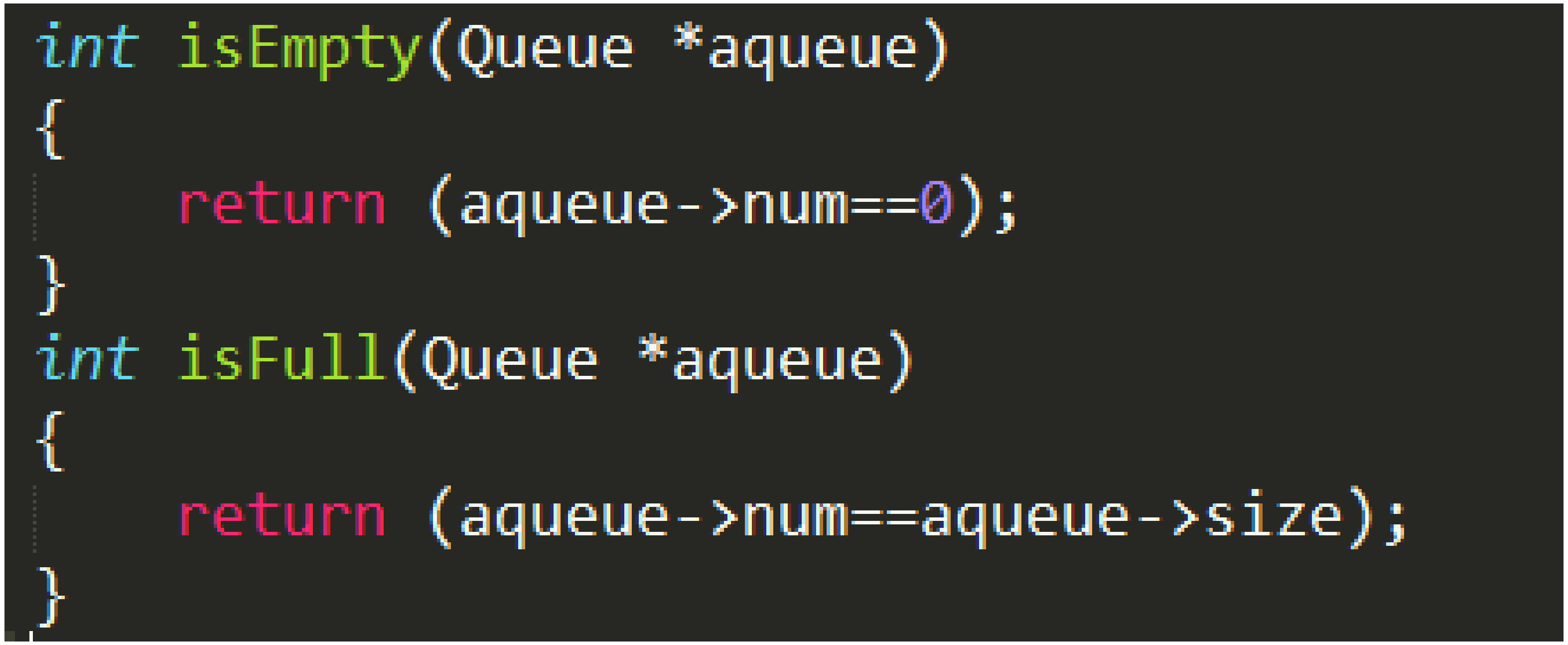


```
//return the data if successfully dequeue, return 0 if fail
int dequeue(Queue* aqueue)
{
    if(aqueue->num > 0)
    {
        int adata = aqueue->data[aqueue->head];
        aqueue->head = (aqueue->head + 1) % aqueue->size;//mod
        aqueue->num--;
        return adata;
    }
    else  return 0;
}
```



Queue Implementation

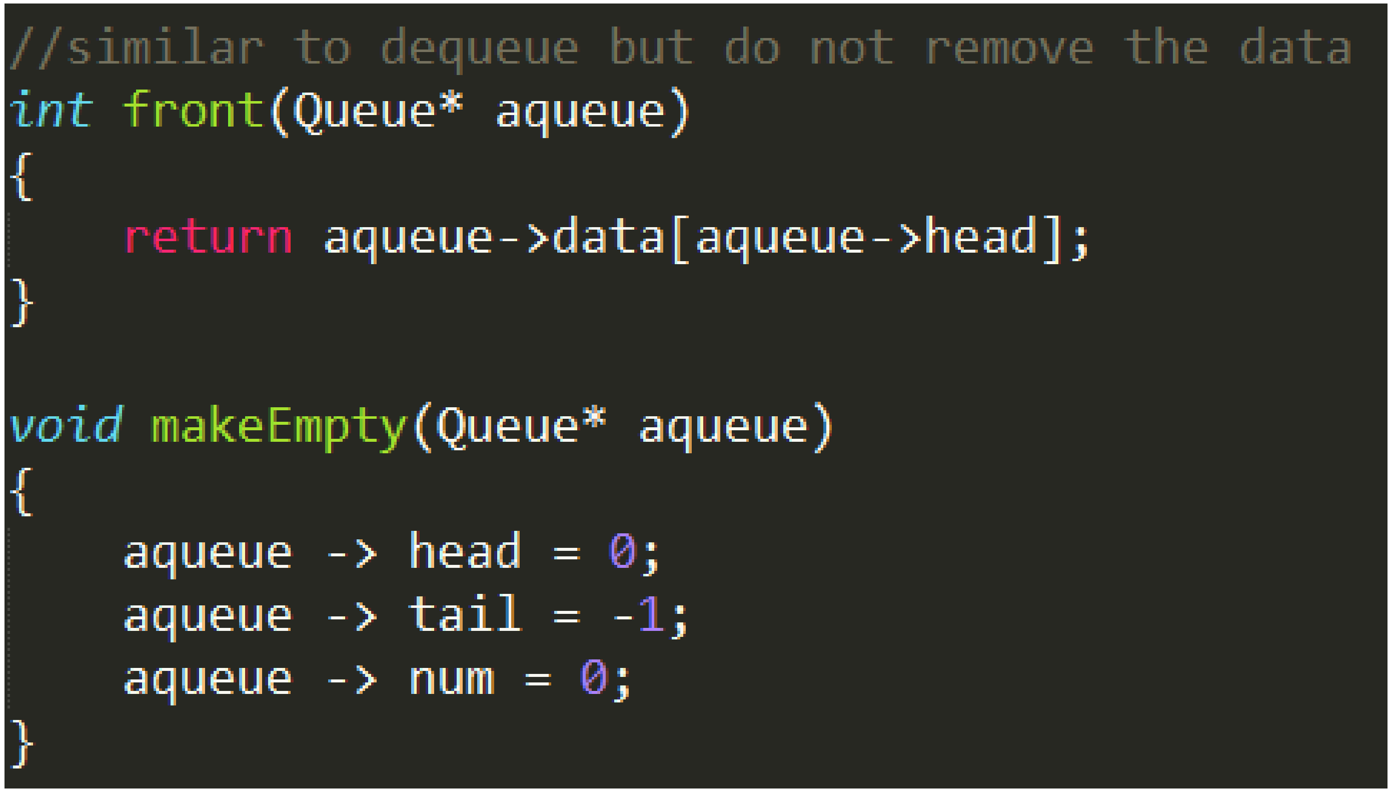
- **isEmpty, isFull**



```
int isEmpty(Queue *aqueue)
{
    return (aqueue->num==0);
}
int isFull(Queue *aqueue)
{
    return (aqueue->num==aqueue->size);
}
```

Queue Implementation

- **front, makeEmpty**



```
//similar to dequeue but do not remove the data
int front(Queue* aqueue)
{
    return aqueue->data[aqueue->head];
}

void makeEmpty(Queue* aqueue)
{
    aqueue -> head = 0;
    aqueue -> tail = -1;
    aqueue -> num = 0;
}
```