Data Structures

Queue Implementation in C

Dimitrios Michail

Dept. of Informatics and Telematics
Harokopio University of Athens

Queue

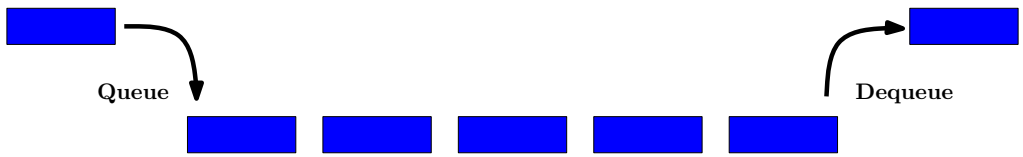
as an abstract data type

A queue is an abstract data type which include the following operations:

- Insert a new element, PUSH(S,x).
- Delete the first element which was added in the queue, POP(S).
- Check if the queue is empty, EMPTY(S).
- Return the size of the queue, SIZE(S).
- ..

Queue

A queue follows the **first-in, first-out** (FIFO) principle.



Queue

Some requirements:

- Separate interface from implementation
- Users are not allowed to change the internal representation (information-hiding)
- Users can create multiple queues at the same time

Queue

interface in C (queue.h)

```
#ifndef _QUEUE_H
#define _QUEUE_H

typedef int item_type;
typedef struct _queue* queue;

queue queue_create();
void queue_destroy(queue q);

void queue_push(queue q, item_type elem);
item_type queue_pop(queue q);
item_type queue_first(queue q);

int  queue_is_empty(queue q);
int  queue_size(queue q);
void queue_clear(queue q);

#endif
```

Queue Implementation

Queue Representation (queue.c)

```
#include <stdio.h>
#include <stdlib.h>
#include "queue.h"

struct node {
    item_type data;
    struct node* next;
};

struct _queue {
    struct node* head;
    struct node* tail;
    int size;
};
```

Queue Implementation

Queue Creation (queue.c)

```
queue queue_create() {
    queue q = (queue) malloc(sizeof(struct _queue));
    if (q == NULL) {
        fprintf(stderr, "Insufficient memory to \
 initialize queue.\n");
        abort();
    }

    q->head = NULL;
    q->tail = NULL;
    q->size = 0;

    return q;
}
```

Queue Implementation

Queue Destruction (queue.c)

```
void queue_destroy(queue q) {
    if (q == NULL) {
        fprintf(stderr, "Cannot destroy queue\n");
        abort();
    }

    queue_clear(q);
    free(q);
}
```

Queue Implementation

Push (queue.c)

```
void queue_push(queue q, item_type elem) {
    struct node* n;
    n = (struct node*) malloc(sizeof(struct node));
    if (n == NULL) {
        fprintf(stderr, "Insufficient memory to \
create node.\n");
        abort();
    }
    n->data = elem;
    n->next = NULL;

    if (q->head == NULL) {
        q->head = q->tail = n;
    } else {
        q->tail->next = n;
        q->tail = n;
    }
    q->size += 1;
}
```

Queue Implementation

Pop (queue.c)

```
item_type queue_pop(queue q) {
    if (queue_is_empty(q)) {
        fprintf(stderr, "Can't pop element from queue: \
            queue is empty.\n");
        abort();
    }
    struct node* head = q->head;
    if (q->head == q->tail) {
        q->head = NULL;
        q->tail = NULL;
    } else {
        q->head = q->head->next;
    }
    q->size -= 1;

    item_type data = head->data;
    free(head);
    return data;
}
```

Queue Implementation

First (queue.c)

```
item_type queue_first(queue q) {
    if (queue_is_empty(q)) {
        fprintf(stderr, "Can't return element from queue: \
            queue is empty.\n");
        abort();
    }

    return q->head->data;
}
```

Queue Implementation

Empty and Size (queue.c)

```
int queue_is_empty(queue q) {
    if (q==NULL) {
        fprintf(stderr, "Cannot work with NULL queue.\n");
        abort();
    }

    return q->head == NULL;
}

int queue_size(queue q) {
    if (q==NULL) {
        fprintf(stderr, "Cannot work with NULL queue.\n");
        abort();
    }

    return q->size;
}
```

Queue Implementation

Clear (queue.c)

```
void queue_clear(queue q) {
    if (q==NULL) {
        fprintf(stderr, "Cannot work with NULL queue.\n");
        abort();
    }

    while(q->head != NULL) {
        struct node* tmp = q->head;
        q->head = q->head->next;
        free(tmp);
    }
    q->tail = NULL;
    q->size = 0;
}
```

Testing the Queue

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "queue.h"

int main() {
    queue q;
    int i;

    q = queue_create();

    assert(queue_is_empty(q));

    for(i = 0; i < 10000; i++) {
        queue_push(q, i);
        assert(queue_first(q) == 0);
    }
    assert(queue_size(q) == 10000);
    assert(!queue_is_empty(q));

    for(i = 0; i < 10000; i++) {
        assert(queue_pop(q) == i);
    }
    assert(queue_is_empty(q));
    assert(queue_size(q) == 0);

    queue_destroy(q);
}
```