Queues

in C

Algorithms and data structures ID1021

Johan Montelius

Fall 2024

Introduction

We're now ready to take a look at a very common data structure that I'm sure your familiar with from your every day life. We're going to implement a queue i.e. a row of items waiting in line. The line is of course ordered so if you add an item to the queue you will have to wait for all other items in the queue to be removed before your new item is in the front of the queue. Queues are also called FIFO, *first-in-first-out*, that describes the functionality. This can be compared to a stack that is also referred to as *last-in-first-out*.

We will implement a linked list. The linked list implementation is as you will see quite easy to implement and work with. It is quite efficient allthough it does allocate and deallocate data structures in each operation.

Using a linked list

A queue has a only one property, the `head`, pointing to the first element in a list of items. New elements are simply *enqueued* to the end of this list and items are *dequeued* from the beginning. Your first queue could look like this:

```
# in queuelist.h

typedef struct node {
  int value;
  struct node *next;
} node;


typedef struct queue {
```

```
  node *first;
} queue;
```

... and here is some skeleton code to get you started on the procedures:

```
# in queue.c

queue *create_queue() {
  queue *q = (queue*)malloc(sizeof(queue));
  q->first = ...;
  return q;
}

int empty(queue *q) {
  return ...;
}

void enque(queue* q, int v) {
  node *nd = ...
  nd->value = ...
  nd->next = ..

  node *prv = NULL;
  node *nxt = q->first;

  while (nxt != NULL) {
    :
  }
  if (prv != NULL) {
    :
  } else {

  }
}

int dequeue(queue *q) {
  int res = 0;
  if (q->first != ...) {
     :
  }
  return res;
}
```

Don't forget to free the node when we dequeue it from the list. If you forget, you will have a memory leak.

We can return a zero if we try to remove an item from an empty queue. This is a drawback but it's ok for now; it does mean that we can not save a zero in the queue since we then would not know if a returned zero is a valid answer or not.

What is the drawback of this implementation? What is the cost of removing the next element? What is the cost of adding a new element? Can we do better?

An improvement

Change the structure of the queue so it holds a pointer to the first element, the `head`, of the queue but also a pointer to the last element, the `tail`. When adding a new item you can then access the last node directly and attach the new node immediately instead of traversing the list to find the last node. You have to be careful when removing the last element and when adding the first element but you should be able to do the implementation in a few lines of code.

What is now the cost of removing an element from the queue? Run some benchmarks and show that you have improved the implementation.