Queues using linked lists in C

Sam Serbouti serbouti@kth.se

October 3, 2024

Introduction

The goal of this assignment is to implement a queue structure using linked lists in C and analyze its performance (particularly the time complexity of enqueueing and dequeueing). Additionally, I will explore an optimization of enqueuing by modifying the structure of the queue.

1 Implementing queues using a linked list structure

```c
void enque(queue* q, int v) {
    node *nd = (node*)..
      ..malloc(sizeof(node));
    nd->value = v;
    nd->next = NULL;
    node *prv = NULL;
    node *nxt = q->first;
    while (nxt != NULL) {
        prv = nxt;
        nxt = nxt->next;
    }
    if (prv != NULL) {
        prv->next = nd;
    } else { //empty queue
        q->first = nd;
    }
}
```

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

int empty(queue *q) {
    return q->first == NULL;
}

int dequeue(queue *q) {
    int res = 0;
    if (q->first != NULL) {
        node *temp = q->first;
        res = temp->value;
        q->first = temp->next;
        free(temp);
    }
    return res;//0 if emty queue
}
```

Queues operate on a FIFO (First-In, First-Out) principle, similar to an everyday life queue. In a queue, each element (`node`) has two parts: a value that holds the data (`int value`) and a next pointer that points to the next node in the queue (`node *next`). The queue itself is (for now) only identified by its front side (`node *first`).

Here's what happenss for `enqueue()`:

1. allocate memory for a new node and set its value, it should point to NULL[1] because it is the last element of the queue.
2. traverse the queue using two pointers: previous node and next node
3. nxt starts at the front of the queue and moves forward until it reaches the end (NULL)
4. insert the new node:
   - if prv is not NULL, it means the queue has elements, so we attach the new node to the last one
   - if prv is still NULL, it means the queue was empty, so we set the new node as the first element

So adding to the queue involves going through the list until you find the end and attaching the new node there: we can expect a $\mathcal{O}(n)$ time complexity.

When you remove an element from the queue `dequeue()`:

1. initialize result to 0 if the queue is empty
2. if the queue isn't empty : store its first node into a temporary node
3. set the results to its value
4. move the head pointer to the next node
5. free the memoory of the old head node
6. return the result

Removing an element always happens at the head of the queue: this takes a time independent of the size of the queue ie $\mathcal{O}(1)$

One downside is that the value 0 cannot be put as a value in a node of the queue because it is used to check if its empty (NULL corresponds to 0). We could instead return an error code.

2 Enqueue and dequeue runtime benchmarks

Before each benchmark, I perform a warm-up operation with a queue containing 90000 nodes. For each queue of size n that goes from 100 to 100000 nodes (on a log10 scale), I build a queue with n nodes with random values

[1] NULL, the null pointer, is a macro that represents a fake or empty memory address

not exceeding 100. I then measure the time it takes to enqueue a random node, and later the time it takes to dequeue.

```
long benchmark(int n) {
    struct timespec t_start, t_stop;
    queue *q = create_queue();
    for(int i =0; i<n;i++){
        enqueue(q,rand()%100);
    }
    clock_gettime(...);
    //enqueue(q,10) or dequeue(q)
    clock_gettime(...);
    long wall = ..
    nano_seconds(&t_start, &t_stop);

    free(q);
    return wall;
}
```

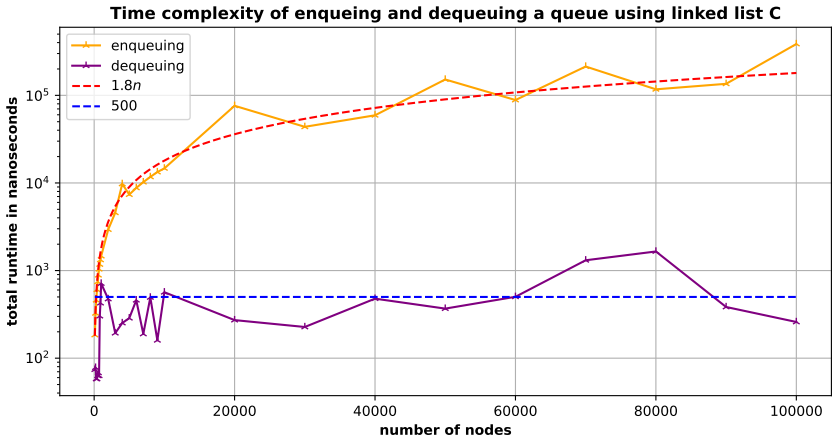

Figure 1: The runtime for enqueue() and dequeue() follow $\mathcal{O}(n)$ and $\mathcal{O}(1)$ respectively which matches the predictions

3 Adding a tail pointer to the queue

Enqueueing, so far, is $\mathcal{O}(n)$. This can be improved to a constant time by having a pointer to the last node (adding a node *tail pointer to the queue structure).

```
typedef struct queue {
    node *head;  // front of the queue
    node *tail;   // end of the queue
} queue;
```

```c
void enque(queue* q, int v) {
    node *new_node = (node*)malloc(sizeof(node));
    new_node->value = v;
    new_node->next = NULL;

    if (q->head == NULL) {
        q->head = new_node;
        q->tail = new_node;
    } else {
        q->tail->next = new_node;
        q->tail = new_node;
    }
}
```

4 Enqueue and dequeue second benchmark

This time, because of the drastic improvement on runtime, I perform 20 measurements for each value of n (ie number of nodes) and extract the average value. I notice that the runtime seems independent of n and thus $t(n) \hookrightarrow \mathcal{O}(1)$

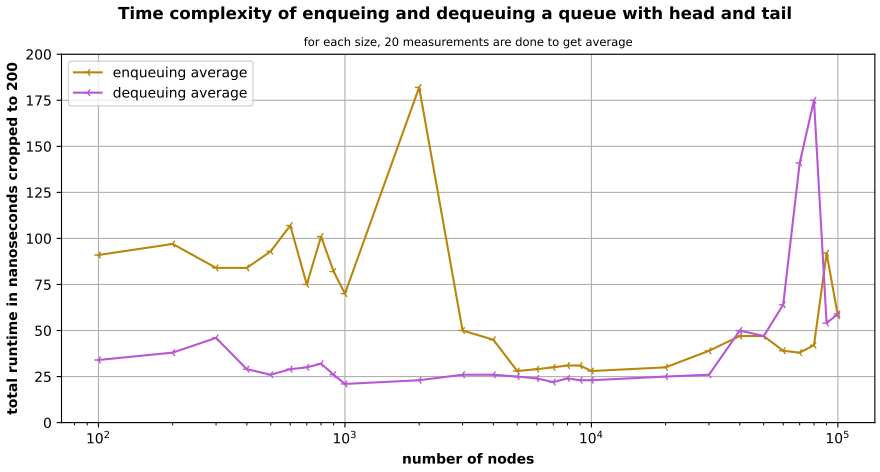

Figure 2: Their runtime is now independent of n

Conclusion

The initial implementation of the queue, though functional, has a time complexity of $\mathcal{O}(n)$ for enqueuing because it needs to go through the list to find the last node. This drawback can significantly affect performance as the number of elements in the queue grows. But I optimized this operation to run in constant time $\mathcal{O}(1)$ by adding symmetry to the data structure.