# Queues using Linked List

Muhammad Shahwaiz Abbasi

VT 2026

## Introduction

This report presents the sixth assignment for the Data Structures and Algorithms (ID1021) course.

We will build a basic queue using a linked list. We will add and remove elements from the queue. Adding is called *enqueue* and removing is called *dequeue*. We will also analyze the performance of a queue. At last, we will create an improved version of the queue that has both a head (pointing to the first element) and a tail (pointing to the last element) and discuss its advantages.
*For benchmark the time was measured in nanoseconds using a function from `Assignment 1`.*

The objective of this assignment is to understand Queues, what they are, how they are made, how they work and how to make them even better.

## What is a Queue?

A queue is data structure just like stack but opposite. The first element added in queue is the first one to be removed. Think of it like a line of people, first one in the line gets to go first. Because of that queues are often called FIFO (First In, First Out). New elements are enqueued by simply adding them to the end of the list and dequeue removes them from the front.
The following is the blueprint for Queue that only holds one property the `head`:

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

typedef struct queue {
    node *first; } queue;
```

The structure is built using nodes to create a linked list where each node contains a value and a reference to the next element. The queue structure contains only one piece of information and that is the first pointer which represents the head of the list.

Our next task in this assignment is to implement the `empty`, `enqueue` and `dequeue` operations to make our queue functional.

```c
bool empty(queue *q) { // is the queue empty?
    return q->first == NULL;
}
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 {
        q->first = nd;
    }
}
int dequeue(queue *q) {
    int res = 0;
    if (q->first != NULL) {
        node *nd = q->first;
        res = nd->value;
        q->first = nd->next;
        free(nd);
    }
    return res;
}
```

In the enqueue operation we have to go through the whole list to add an element to the last. This is not time efficient as the list sizes grows, the runtime increases significantly. This makes the time complexity for enqueue operation $\mathcal{O}(n)$. The dequeue operation on the other hand, is not dependent on the size of the list because it only focuses on the first element in the list. So the time complexity for dequeue operation is $\mathcal{O}(1)$.

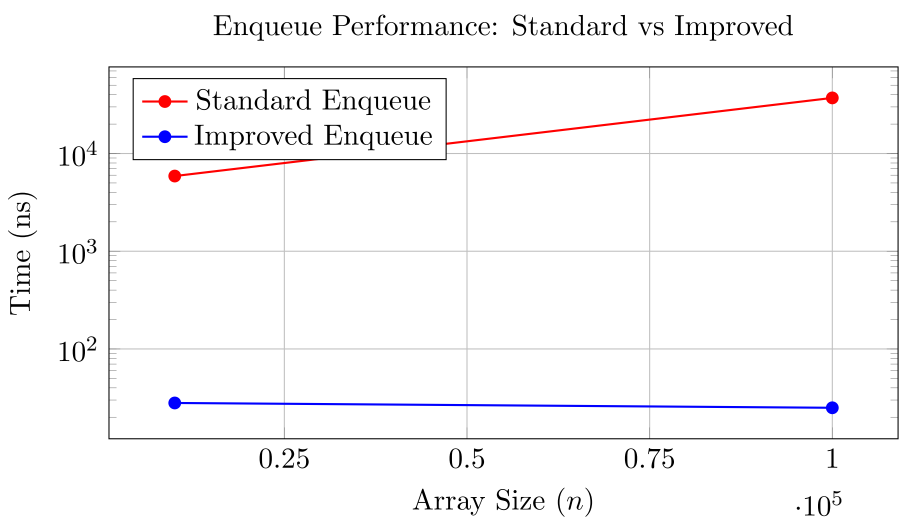
## An improvement

```
typedef struct queue {
    node *first;
     node *last;
    } queue;
void enque(queue* q, int v) {
    node *nd = (node*)malloc(sizeof(node));
    nd->value = v;
    nd->next = NULL;
    if (q->first != NULL) {
        q->last->next = nd;
    } else {
        q->first = nd;
    }
    q->last = nd;
}
```

The improved queue structure now contains two information, the first pointer and the last pointer. The last pointer make it easy for queue to add integers to the end instantly, unlike the old version, you have to go through the whole list to add the new element. So now the time complexity for enqueue operation becomes $\mathcal{O}(1)$.

## Comparison between Standard and Improved Implementation

The graphs in `Figure 1` confirm our expectations. The time complexity of `enqueue` from standard to improved have changed from $\mathcal{O}(n)$ to $\mathcal{O}(1)$.



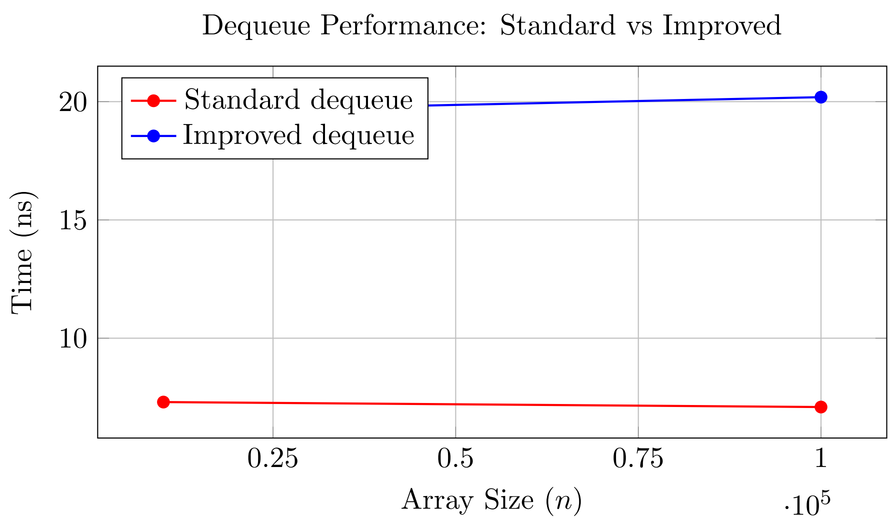


Figure 1: Standard vs Improved Queue Operations Performance