Queue

Fredrik Berzins

Fall 2023

Introduction

A queue is a structure implementation that is built on first in first out so the inverse of a stack witch is first in last out. A queue implementation can be made from multiple data types/structures, in this report four different benchmarks will be done for different data types and implementations. This will show benefits and downside from using different data types for the queue implementation.

Queue

A queue is a data structure implementation that is useful when a first in first out is needed, this commonly used for ordered information like a instruction for parallel systems. It could also be used to order a binary tree when using a iterator dependent on distance to root, compared to using a stack witch will order it dependent on left most first. So dependent on use case and information stored in a binary tree it might be preferred to use a queue over a stack or vise versa.

Implementation

Queue as Array implementation

To use an array as a queue it needs keep two indexes as variables to the first item in the queue and the first empty slot in the queue. When adding items to the array it just adds them to the index of the first empty module array length, this adds the item to the first empty slot ad wraps round to the first index of the array if it retches the end. This is done to reuse slots that might be unused since removing values from the queue, since when removing values the first index just increments forward.

```
public void add (T val) {
    if (first%arr.length == firstEmpty%arr.length != this.empty()) {
        expArr();
    }
    arr[firstEmpty%arr.length] = val;
    firstEmpty++;
}
public T remove() {
    if (this.empty()) {
        return null;
    }
    T val = (T)arr[first%arr.length];
    first++;
    return val;
}
```

Queue as Linked List implementation

To use an linked list as a queue it just needs to unlink the first element when removing it and iterate to the end when adding. When implementing a queue like this it will have a big O of O(1) to remove but O(n) when adding since it has to iterate thru the list, This can be changed if it has a pointer to the last/end element in the linked list this will make both add and remove to have a big O of O(1). The difference between add and remove for the linked list with or without end pointer can be seen below.

```
public void add(T item) {
    if (first == null) {
        first = new Node(item);
    } else {
        Node n = first;
        while(n.next != null) {
   n = n.next;
        }
        n.next = new Node(item);
    }
}
public T remove() {
    Node n = first;
    if (n == null) {
        return null;
    }
    first = n.next;
    return n.item;
}
// With last/end pointer
public void add(T item) {
    if (first == null) {
        first = new Node(item);
        last = first;
    }
    else {
        last.next = new Node(item);
        last = last.next;
    }
}
public T remove() {
    Node n = first;
    if (first == null) {
        return null;
    } else if (first.next == last) {
        last = null;
    } else {
        first = first.next;
    }
    return n.item;
}
```

The largest difference are in the add where you go thru the list when their isn't a pointer to the last item, while the one with the pointer uses it and changes it to the new added node. The difference for remove isn't as major since both methods remove at the first pointer, the only difference is that the end pointer will be set to null if the queue gets emptied.

Benchmarks

The testing was done with different amount of add and remove actions between 100 and 3200 values(doubling each time, 100,200,400...). every 1'000 time tests was redone 1'000 time, this gives a low time showing avg from the group with the 1'000 quickest runs.

Result

The result shows a linear relation ship for a n amount of actions witch means it's a constant time or gives a big O of O(1) for add and remove action except for when the add of linked list without a end pointer witch has a big O of O(n). The results also show how the expansion of the arrays effect on total time, the effect is about 25 to 30 percent.

n	**Dynamic Array**	**Fixed size Array**
100	877	598
200	1782	1232
400	3614	2608
800	7321	5393
1600	15427	11432
3200	30303	22302

Table 1: time to add and remove n items from array based queue in ns

n	**Linked List**	**Linked List with end pointer**
100	5851	403
200	22888	841
400	89476	1768
800	354177	3725
1600	1450966	7595
3200	5737870	15744

Table 2: time to add and remove n items from linked list based queue in ns

Conclusion

In conclusion a linked list with pointer to the end is the fastest and has a easy implementation since you don't need to loop thru like without the pointer or expand like the array. Arrays have the down side of of being slower but