Queues

Jonathan Nilsson Cullgert

Fall 2023

Introduction

In this assignment we will implement the queue as a data structure. As one might expect from the name a queue behaves in the same way it does in real life. If you add something to the queue you will find it in the back and if an element is being removed it will be the first element added to the queue. This is very similar to the stack which is a last in first out data structure. The queue is a first in first out data structure. In this assignment we will implement the queue with a linked list structure as well as with an array and a dynamic array. The queue will also be used in the iterator for the binary tree from the previous assignment to implement a breadth first traversal instead of the depth first traversal with the stack.

Using a linked list

The first implementation of the queue is by using a linked list. Using a linked list to create a queue is perhaps not as effective as implementing it with an array but it is much simpler which might have its uses. The queue will have nodes like in a linked list which will point to another node as well as contain some form of data. The first implementation of the queue only had a pointer to the head of the queue, the item which will be removed first if that method is called. Is is handy to also have a pointer to the last element in the queue since that will allow us to add an element to the queue without having to traverse the entire to queue to find the last spot.

```
public Queue() {
    head = null;
    last = null;
}
```

The **add()** method will create a new node with some inputted data and a null pointer and then add it to the end of the queue. The reason that the added node will have a null pointer is because it is the last element so it does not need to point to anything.

```
public void add(Integer item) {
     Node node = new Node(item, null);
     if (head == null){
          head = node;
          last = head;
          return;
     }
     else
          last.next = node;
          last = node;
}
```

This will add the node to the end of the queue as well as keep both the head pointer and last pointer, so that they actually point to the correct node. This method and the entirety of the class could be modified so that it is a generic method which could take any data type, which could be helpful in the breadth first traversal.

The last method to be implemented is the remove method, which will return the data of the first node and move the head pointer to the next element in the queue. It will also check if the first node is null, in that case it will throw an exception saying that the queue is empty.

```
public Integer remove() {
     if (head == null)
          throw new IllegalArgumentException("queue is empty");
     Integer node;
     node = head.item;
     head = head.next;
     return node;
}
```

Implementing the queue this way will change the time complexity for the **add()** method to $O(1)$ instead of being $O(n)$ which makes this a very effective way of implementing the queue. It is also important to make sure that the nodes actually become unlinked in the correct way so that the garbage collector can remove data that is not in use.

Breadth first traversal

In this part of the assignment another approach to searching through a binary tree is implemented. Instead of searching through the depth to the

leafs of the tree, each level of the tree will be searched through. This method is called the breadth first traversal. When using depth first traversal the nodes are pushed onto the stack are then later used to find the parent nodes. The breadth first traversal uses the same principle but puts the nodes into the queue. When a node is removed from the queue, that is the level we are currently on.

To implement this, the iterator from the previous assignment is used but modified to accommodate the queue instead of the stack. The important part to note when implementing this iterator is to make sure that the nodes in the queue takes the nodes from the binary tree as the data type. Since I did not implement this queue as a generic method I had to change the datatype in the nodes to binarytree nodes instead of taking in Integers.

```
public Integer next(){
    temp = queue.remove();
    if(temp.left != null)
        queue.add(temp.left);
    if(temp.right != null)
        queue.add(temp.right);
    return temp.value;
}
```

This iterator works as following, it adds the root to the queue, stores the root in a temporary node, adds the left and right nodes of the temporary node to the queue as long as they do not equal null. It then return the value that the temporary node holds, though it could return the node itself if that is what was required. This implementation of the **next()** method of the iterator gives the correct order of the nodes. The **hasnext()** method is implemented very easily, it checks if the queue is empty or not.

This way of traversing the binary tree is useful if searching for something that is close to the root, especially if the binary tree is not evenly distributed.

An array implementation

The next part of the assignment is to implement the queue in an array instead of with a linked list. This way of implementing the queue seems very easy at first, the queue will have a maximum size and it will be very simply to keep track of the first and last element in the queue, just have pointers that can be incremented or decreased by one when something is added or removed in the queue. Some problems will arise when queue reaches the length of the array, the queue will be full. When removing elements from the queue spots in the front of the array will be empty, wasting memory.

One way of fixing this problem will be by having the queue wrap around, for example if the array has a size of five and two elements have been removed

it could look like this: $[null, null, 3, 4, 5]$. This queue will not be able to add more items and it is also wasting memory in the current implementation. If we let the queue wrap around so that we add new elements to the queue to the front of the array if the size of the array has been reached, this problem will somewhat be solved.

By wrapping around the array we will be able to fully utilize the allocated memory of the array and there will be no risk of reaching the end of the array, since it wraps around. There is still one problem though, if enough elements are added the index pointing to where elements should be added will be pointing at the same place where the first element in the queue is. This will make it look like the queue is empty even though it is full.

This final problem is solved by letting the queue and array be dynamic, similarly to the stack in the first assignment. If we reach the point where the two pointers are pointing to each other when the queue is full we simply let the array be bigger. When copying the items over to the bigger array we copy all items from the first item in the queue to the end of the previous arrays, these elements are added at index zero and up in the bigger array. We then copy all items from the start of the array to the end of the queue in the element next to where the previous copying ended in the bigger array. This way the queue will start at index 0 in the bigger array, at least until it new elements are added and removed.

```
public void enqueue(Integer itm){
    queue[last] = itm;
    last = (last + 1) % size;
    if (first == last){
        Integer[] copy = new Integer[size*2];
        int c = 0;
        for(int i = first; i < size; i++){
            copy[c] = queue[i];
            c++;
        }
        for(int i = 0; i<last; i++){
            copy[c] = queue[i];
            c++;
        }
        size = size*2;
        first = 0;
        last = c;
        queue = copy;
    }
}
```

So in this code snippet last is the pointer to the last element in the queue, first is to the first element in the queue and c will be the new position where the elements will be copied to in the new bigger array. By doing this the queue will not become full since it can always become bigger should it be necessary for it to do so. In summary this method uses modulo to keep track of when it should wrap around the array and two pointers, first and last, to know where the elements in the queue is and when the array it is located in should become bigger if it is needed.

It should also be noted that a way to shrink the array if the queue becomes small enough could also be implemented, this would help the memory usage of the queue. In this assignment the creation of such a function was not necessary.

Benchmark

The benchmarks I took were how long it would take to add and remove n amount of elements from the queue in an array and as a linked list. The time complexity for adding/removing elements in the queue is constant $O(1)$, at least in the way they were implemented in this assignment. The reason that the time changes in the array queue is because it is dynamic, it has to double its size, in the array queue the cache can affect the time it takes. In the list queue the time it takes to create the nodes will affect the time. There will also be a lot of different background tasks and similar things that affect the time, but the operations themselves should be constant (unless the adding of an element makes it so that the array has to be increased in size).

n	Arrayqueue with starting size 1000	Listqueue
1000	120	160
2000	90	120
4000	150	250
8000	280	300
16000	500	560

Table 1: A table which compares how long it takes to add and remove n amount of elements from an arrayqueue and a listqueue in (us)