Queues

Mahdi Nazari

October 2022

Introduction

In this assignment the queue data structure will be presented. This data structure will be implemented by linked list and array and it will be explained how does every of these implementations work. Binary tree from previous assignment will be reimplemented and this time by a queue instead of stack. Queue implemented by array has many corner cases which should be considered and this will be discussed. To be able to insert elements to the queue without any trouble, the queue was constructed on a dynamic array which can increase and decrease in size.

Queue linked list

Queue is a data structure which has one property and that is the queues head. The only principle of the queue is so called FIFO(first in, first out). The simplest way to implement a queue is by a linked list. To add an element to the queue we traverse in the list and insert the element at the end of the list and make the last element to point to this new element. To remove an element we make that the head of the queue points to the next element in the queue and reference of the previous head element should point to null. By this implementation the add method of the queue has a time complexity of O(n) type since the program has to traverse in the list until it reaches the end of the list. Time complexity of removing an element is constant and an O(1) type.
To make the time complexity of the add method in the queue as a constant(O(1)) we add another pointer in the queues properties which points to the end of the list. This help us to avoid the traversing in the list to add an element at the end of the list. Since the new pointer has the reference to the last element we have to make that the last element points to the new element and update the pointer so that it points to the new element which is now the end of the list. The optimized add method looks as follows:

```
public void add(BinaryTree.Node item) {
```

```
        Node newNode = new Node(item, null);
        this.end.tail = newNode;
        this.end = newNode;
    }
```

Binary tree queue

Now we can reimplement the binary tree with a queue instead of a stack. But to do this we need to introduce a new traversing algorithm which is called Breadth first traversing. In a breadth first algorithm the program start at the root of the tree and then go down to deeper level after it has explored the current level. If we use queue in the binary tree iterator the constructor of tree iterator create a queue by the trees root. The iterator will still have hasNext() and next() methods as before. The hasNext() method will check if the queue is not empty. The next() method will check if the first element in the queue that should be removed from the queue has left or right branches in the tree, if it does the next() will add those elements in the queue. This procedure goes on until the tree iterator has traversed in the tree. The next() method in the tree iterator looks as follows:

```
    public Integer next() {
        if(hasNext() == false){
            return null;
        }
        else{
            BinaryTree.Node tempNode = queue.head.item;
            if(tempNode.left != null){
                queue.add(tempNode.left);
            }
            if(tempNode.right != null){
                queue.add(tempNode.right);
            }
            tempNode = queue.remove();
            return tempNode.value;
        }
    }
```

Queue array

In this part, the queue data structure will be constructed by an array. To explain the principle briefly, to use an array as a queue we add elements

at the end of the array and remove from the beginning of the array. But since there are many corner cases we need to take a checklist of the corner cases and try to tackle those one by one. First of all how we can add and remove an element in and from the queue based on array? To perform these operations we need two pointers that will be called "front" a pointer that points to the start of the queue and "end" a pointer that points to the end of the queue. Now to add an element in the queue we just need to increment "end" by one which gives us an index and then put the new element at that index. To remove an element we increment the pointer "front" by one which means that queue has a new "front" and the old front does not exist anymore in the queue. The "front" and "end" pointers are pointing to the index before the start of the array. When we first time insert an element in the queue the "front" and "end" will be incremented to zero and then the first element will be inserted at the index zero. The constructor of the queue:

```
public QueueArray(){
     front = -1;
     end = -1;
 }
```

Now we need to consider situation there we need to add an element to the queue but the "end" pointer points to last index in the array then we should check if there is place before the pointer "front" then we can overwrite on those indexes. To wrap around the edges of the array we can use the module operation. If the queue is initialized by an array of length n we can take the module of the pointers "front" and "end" every time they are incremented. This gives us the possibility to wrap around the edges of the array. Since module operation perform as a cyclic group and will give us the index from 0 to n-1. We need to keep track of when the "end" and "front" pointers are next to each other, then it means that the queue is full and we have to solve this problem which will be discussed in the nexst section. Insert method in the queue looks as follows:

```
public void insert(int value){
      if((end +1) \% n == front){ // the queue is full
          increase();
      }
      if(front == -1 && end == -1){ // the queue is empty
          front = end = 0;
      }
      else {
          end = (end + 1) \% n ;
      }
```

```
        arr[end] = new Node(value, null);
        size++;
    }
```

The delete method has some corner cases as well. If the queue is empty (front == -1 && end == -1) then we just return. If (front == end) it means that just one element is left in the queue then we can set (front = end = -1) which means that the "front" and "end" points to the index before the beginning of the array. Else it will just increment the "end" as end = (end +1)%n

Dynamic queue

To add more elements to the queue which is full a dynamic queue is needed. A dynamic queue will work as a dynamic stack since both stack and queue are constructed of arrays. We can add a line code to the insert method that will call the increase method when the queue is full. Then increase method will create an array of length (2*n) where (n) is the initialized size of the array. Then it will check two possible scenarios. 1) if(end> front) then the method should copy all elements from index "front" to index "end" to the new larger array and update "front", "end" and n. 2) if(front> end) then the method should copy all elements from index "front" to the n-1 to the new larger array and then copy all elements from index 0 to the index "end" to the larger array and update "front", "end" and n. Below is the increase method in a dynamic queue:

```
Node[] newArr = new  Node[N*2];
           int j=0;
       if(end > front){
           for(int i=0; i <= (end - front); i++){
               newArr[j++] = arr[i + front];
           }
       }
       else if(front > end) {
           for(int i =front; i<N; i++){
               newArr[j++] = arr[i];
           }
           for(int t=0; t<=end; t++){
               newArr[j++] = arr[t];
           }
       }
           front = 0;  end = n-1;   arr = newArr;  n = n*2;
```

Now we can also add another function to the queue that will decrease the size of the array when it drops below a limit. To do this we need another variable that will keep track of number of elements . The decrease method constructed so when number of elements in the queue is less than one fourth of the initialized size of the array and the size of array is greater than 5 (this will prevent shrinking of the queue when the size is 5 which would not generate any benefits) then the size of array should shrink to half and then it should copy all elements from the larger array to the new smaller array. To do this decrease method should check two possible scenarios. 1) if (end >= front) then the method should copy all elements from index "front" to index "end" to the new smaller array and update "front", "end" and n. 2) if(front > end) then the method should copy all elements from index "front" to index n-1 (last index) to the new smaller array and then copy all elements from index 0 to the index "end" to the smaller array and update "front", "end" and n. The decrease method is as follows:

```
if(size < (n/4) && n>5){
    Node[] newArr = new  Node[n/2];
    int j = 0;
    if(end >= front){
        for(int i=0; i<=(end-front); i++){
            newArr[j++] = arr[i+front];
        }
    }
    else if(front > end){
        for(int t=0; t<n; t++){
            newArr[j++] = arr[t];
        }
        for(int i=0; i<= end; i++){
            newArr[j++] = arr[i];
        }
    }
    front = 0;   end = --size;    arr = newArr;   n = n/2;
```