Homework 3

Wei Ye[*]
CIS5006 - Data Structure

Due on Oct 16, 2022

1 Stack

- 5.

  It's overflow. As the top element has already reached to the max index of this array, and our array is static array. We can't expand the array dynamically, so it's overflow if we push another element.

  Letter: X, Stack.top: 4; overflow: T; underflow = F items: A, B, C, D, E

- 7.
  Letter = z; stack.top: 3; overflow = F; underflow = F; items: A,B,X,Y

- 10.
  a

- 15.

```
ReplaceItem(StackType &stack, ItemType oldItem, ItemType newItem)
    //use newStack to receive the newItem
    stack <ItemType> newStack;
    while(!stack.isEmpty()){
        if(stack.top()==oldItem){ newStack.push(newItem);}
        else {newStack.push(stack.top());}
        stack.pop();
    }
    while(!newStack.isEmpty()){
        stack.push(newStack.top());//put it back
        newStack.pop();
```

[*]2nd year PhD student in Economics Department at Fordham University. Email: wye22@fordham.edu

```
        }
    }
```

Time complexity is still $O(n)$. The drawback here is we increase the space complexity by creating a new stack.

- 16.

```
    //set up the candy container as a stack.
    stack <string> candy;// colors in string
    stack <string> newCandy;
    for(int i = 0; i < candy.size();++i){
        while(candy.top()!= "yellow"){
            newCandy.push(candy.top());
            candy.pop();
        }
        candy.pop();
    }
    for (int i = 0; i < newCandy.size();++i){
        //put candies in originial order except
        //we consumed yellow candies.
        candy.push(newCandy.top());
        newCandy.pop();
    }
```

- 18.

```
    Bool Identical(StackType stack1, StackType stack2){
        if (stack1.length()!= stack2.length()){
            return false;
        }
        while(!stack1.isEmpty() &&!stack2.isEmpty()){
            if (stack1.top()==stack2.top()){
                    stack1.pop();
                    stack2.pop();
            } else{
                return false;
            }
        }
        return true;
    }
```

2 Queue

- 22.

  a. First, enqueue a couple of elements in the queue. 0, 1, 5. The pop out 0, it becomes 1,5. Then enqueue 16, 0, 4 respectively. It becomes 1,5,16,0,4. Dequeue 1. The queue becomes 5,16,0,4. And then it prints items 1, 2, and 3. while loop is for when the enumerate the queue, when the queue is not empty, always dequeue 1. When dequeue 1, it print 1 to show how many item1 have been dequeued.

     cout would be 1 0 4 5 16 0 3

  b. It first forms item2 from the summation of item1 and 1. item2 becomes 5. First, the queue pushes (enqueue) some items. it becomes 5,6,4. then dequeue item 2, becoming 6,4. Assign item1 with new value to 6. then enqueue, the queue is 6,4,4,0.while loop to pop out all the item3. then print item3. In the end, print the new values of item1, 2, and 3.

     cout would be 6 4 6 0 6 5 0

- 26.
  L, 0, 4, F, F, V, W, X, Y, L.

- 29.
  W, 2, 0, F, F, V X Y Z (note: there is no char at index 1)

- 33.
  The logic is to use new container(queue) tp receive the elements, then put them back. The client can only get the temp, others are unavailable. Here are the codes.

-
```
    ItemType Front(QueType &queue){
        ItemType temp;
        QueType newQue;
        //receive the element that's been dequeued from original queue.
        queue.Dequeue(temp);
        newQue.Enqueue(temp);
        ItemType item;
        while(!queue.isEmpty()){
            queue.Dequeue(item);//pop out the rest
            newQue.Enqueue(item);
        }
        //add items back to original queue.
        while(!newQue.isEmpty()){
            newQue(item);//pop out
            queue.Enqueue(item);
        }
        return temp;//return the front here.

    }
```

- 34.

```
    void ReplaceItem(QueType queue, int oldItem, int newItem){
        ItemType temp;
        QueType newQue;
        while(!queue.isEmpty()){
            queue.Dequeue(temp);
            if (temp == oldItem){
                newQue.Enqueue(newItem);
            }else{
                newQue.Enqueue(temp);
            }
        }
        //put the elements of newQue back to queue.
        while(!newQue.isEmpty()){
            newQue.Dequeue(temp);
            queue.Enqueue(temp);
        }
    }
```

- 40.

  1. If as a constructor, it's called by default when we implement the class containing the constructor. However, if MakeEmpty() as an operation, it's only called when the user does the operation.
  2. Constructor can initialize private data member in the class. MakeEmpty() can reset private members but data stored in the class can't be changed.
  3. Constructor doesn't return any type of data, but MakeEmpty() can, aka void as return type.