CSC263: Assignment 4

Jingqi Zhang, Yulin WANG, Tianjing Ruan

Due Date: *07-22-2019*

Question	Written By	Checked By
1	Jingqi Zhang	Tianjing Ruan
2	Yulin WANG	Tianjing Ruan

Question 1

a)

I use Linked List as my underlying data structure. For each Deque we construct, besides a Linked Node N to keep track and store all elements (each nodes which has 3 fields: value, front, and next), We add another 4 fields in our Deque: ptHead (the pointer pointed to the head node we stored), ptBack (the pointer pointed to the last node we stored), MaxList(a doubly linked list to keep track the pointer of node we store in N as decrease value order), ptMax(the pointer pointed to a node in MaxList which value pointed to the max value node we have stored in N).

For DequeWithMax, we construct 4 fields we need.

For pushFront() function, if the deque is empty, let N, ptHead, ptBack point to the newly pushed node x, and add newly pushed node x's pointer in MaxList then let ptMax pointed to this node in MaxList. If the deque is not empty, let the front of the head node points to the newly pushed node x and let x.next point to the head node. Then move the pointer ptHead to x so that x becomes the new head node. Next, we update the MaxList by comparing value x with the head node's value of MaxList, poping all the node pointer which pointed to the smaller value node that we stored in N, until there exist a node greater than or equal to x. If the MaxList 's ptMax node has been poped out or the MaxList is empty, change ptMax to this new node x's pointer then append new node x's pointer as the new head node of MaxList (Notice that we maintain MaxList from head node to the ptMax node in increasing order)

For popFront() function, if the deque is empty, the function returns NIL. Otherwise, we pop the head node x out, then let x.next.front becomes NIL and the pointer ptHead points to x.next so that x.next becomes the new head node. If the node that has been popped is the max value node, we update the MaxList by dropping the ptMax node (which value is the pointer pointed to the max value node we stored in N). Then we move the pointer ptMax to the new max value node, which is either the front node or the next node of this dropped ptMax node in MaxList (obviously we point to the greater one).Then we return the poped node.

In addition, pushBack(), popBack() function updates the pointer ptBack and MaxList in a similar way. To be specific,

For pushBack() function, if the deque is empty, let N, ptHead, ptBack point to the newly pushed node x, and add newly pushed node x's pointer in MaxList then let ptMax pointed to this node in MaxList. If the deque is not empty, let the next of the back node points to the newly pushed node x and let x.front point to the back node. Then move the pointer ptBack to x so that x becomes

the new back node. Next, we update the MaxList by comparing value x with the back node's value of MaxList, poping all the node pointer which pointed to the smaller value node that we stored in N, until there exist a node greater than or equal to x. If the MaxList 's ptMax node has been poped out or the MaxList is empty, change ptMax to this new node x's pointer then append new node x's pointer as the new back node of MaxList (Notice that we maintain MaxList from back node to the ptMax node in increasing order)

For popBack() function, if the deque is empty, the function returns NIL. Otherwise, we pop the back node x out, then let x.front.next becomes NIL and the pointer ptBack points to x.front so that x.front becomes the new back node. If the node that has been popped is the max value node, we update the MaxList by dropping the ptMax node (which value is the pointer pointed to the max value node we stored in N). Then we move the pointer ptMax to the new max value node, which is either the front node or the next node of this dropped ptMax node in MaxList (obviously we point to the greater one). Then we return the poped node.

All in all, we keep the MaxList's node pointed to the node in N whose values are as a double decreasing order from ptMax node to both the head node of MaxList and the back node of the MaxList.So for maximum(), we always keep track ptMax pointed to the node in MaxList whose value is the pointer pointed to the max value node in N. That is to say, We return the node of ptMax's value pointed to.

Now I show you how each function work using some pseudo code.

-DequeWithMax
N ← NIL, ptHead ← NIL,ptBack ← NIL, MaxList ← NIL, ptMax ← NIL

-pushFront(x)
```
NewNode := LinkedNode.Init(x)
NewNodePt := LinkedNode.Init(&NewNode)
IF N = NIL
    N ←NewNode
    ptHead ← &NewNode
    ptBack ← &NewNode
    MaxList ←NewNodePt
    ptMax ← MaxList
ELSE
    *(ptHead).front ← &NewNode
    NewNode.next ← ptHead
    N←NewNode
    ptHead← &NewNode
    searchNode ← MaxList
    WHILE x ≥ ((*searchNode).value).value
```

```
        IF *ptMax = searchNode
            ptMax ← &NewNodePt
        MaxList ← *(MaxList.next)
        MaxList.front ← NIL
        searchNode ← MaxList
    IF MaxList = NIL
        ptMax ← &NewNodePt
    MaxList.front ← &NewNodePt
    NewNodePt.next ← &MaxList
    MaxList ← NewNodePt
```

-popFront()

```
IF N = NIL
    RETURN NIL
ELSE
    popNode ← *ptHead
    N ← N.next
    N.front ← NIL
    ptHead← &N
    IF &popNode = (*ptMax).value
        NewMaxPt = &MaxValueNode(*(*ptMax.value).front,*(*ptMax.value).next)
        (*ptMax).front.next ← (*ptMax).next
        (*ptMax).next.front ← (*ptMax).front
        ptMax ← &NewMaxPt
RETURN popNode
```

-pushBack(x)

```
NewNode := LinkedNode.Init(x)
NewNodePt := LinkedNode.Init(&NewNode)
IF N = NIL
    N←NewNode
    ptHead← &NewNode
    ptBack ← &NewNode
    MaxList ←NewNodePt
    ptMax← &NewNode
ELSE
    NewNode.front ← ptBack
    *(ptBack).next ← &NewNode
    ptBack← &NewNode
    searchNode ← MaxList.back
    WHILE x ≥ ((*searchNode).value).value
        IF *ptMax = searchNode
            ptMax ← &NewNodePt
        MaxList.back ← *(MaxList.back.front)
        MaxList.back.next ← NIL
        searchNode ← MaxList.back
```

```
    IF MaxList = NIL
        ptMax ← &NewNodePt
    MaxList.next ← &NewNodePt
    NewNodePt.front ← &MaxList
    MaxList.back ← NewNodePt

-popBack()
IF N = NIL
    RETURN NIL
ELSE
    popNode ← *ptBack
    ptBack← &ptBack.front
    (*ptBack).next ← NIL
    IF &popNode = (*ptMax).value
        NewMaxPt = &MaxValueNode(*(*ptMax.value).front,*(*ptMax.value).next)
        (*ptMax).front.next ← (*ptMax).next
        (*ptMax).next.front ← (*ptMax).front
        ptMax ← &NewMaxPt
RETURN popNode

-maximum()
RETURN *(*ptMax.value)
```

Question 2

a)

Use disjoint set as the data structure for this algorithm.
Firstly, for each numbered cell in the maze, we make a set. Then the number of sets num_set = rc, where r represents the value of rows and c for columns. While num_set > 1, we randomly select a wall called (a,b) from the list of wall objects called Walls. If Find-set(a) $\neq$ Find-set(b), then we do Union(a,b) and num_set -= 1, then remove this wall from the list Walls. Otherwise, if Find-set(a) = Find-set(b), then pop this wall from Walls and add it into a new List called Results. Finally when num_set = 1, add all the walls left in list Walls into list Results and then return the list Results, containing all still existing walls in the maze.

b)

Firstly, making set for cells takes O(rc), since there are rc sets and each make-set operation takes O(1). And for the while loop, it has at most O(rc) iterations, and it takes at most O($\alpha(n)$) for each iteration since operations Find-set and Union each cost O($\alpha(n)$) and other operations cost constant time. Thus the whole while loop takes at most O($\alpha(n) \cdot rc$) time. And since then it takes constant time to form and return the list Results, so this algorithm takes at most O($\alpha(n) \cdot rc$) + O(rc) = O(rc) time in total. Therefore, the worst case runtime is O(rc).