cs 5800 - hw7

Yijing Xiao

November 2021

1 Simple Data Structure

1.1 Stack and Queue

Both are dynamic sets in which the element removed from the set by DELETE operation is prespecified. In a stack, the element deleted from the set is the one most recently inserted: the stack implements a last-in, first-out. or LIFO, policy. In a queue, the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in, first-out, or FIFO, policy.

1.2 Linked list

A linked list is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object.

1.3 Representing rooted trees

Binary tree and rooted trees with unbounded branching.

2 Binary Search Tree

Basic operation on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in $\Theta(lgn)$ worst-case time. If the tree is a linear chain of n nodes, however, the same operations take $\Theta(n)$ worst time.

2.1 What is a binary tree

Theorem 12.1: If x is the root of an n-node subtree, then the call INORDER-TREE-WALK(x) takes $\Theta(n)$ time.

2.2 Querying a binary search tree

Theorem 12.2: we can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR so that each one runs in $\emptyset(h)$ time on a binary search tree of height h.

2.3 Insertion and deletion

Theorem 12.3: we can implement the dynamic-set operations INSERT and DELETE so that each one runs in $O(h)$ time on a binary search tree of height h.

2.4 Randomly built binary search trees

Theorem 12.4: The expected height of a randomly built binary search tree on n distinct keys is $O(lgn)$.

3 Exercise 10.1-1

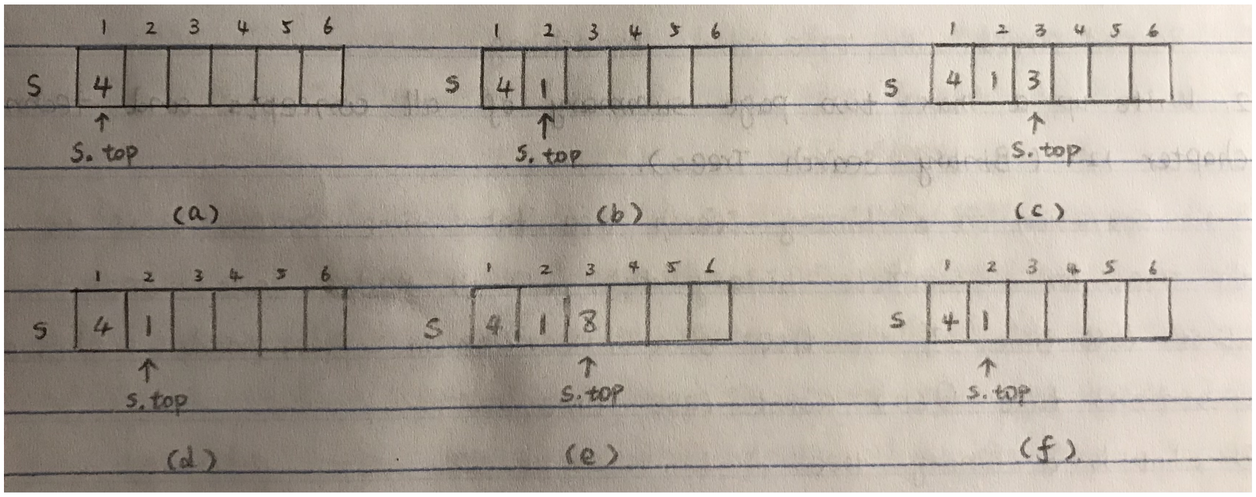

Figure 1: image

4 Exercise 10.1-4

When $Q.head = Q.tail = 1$, the queue is empty, and if we attempt to dequeue an element then the queue underflows. When $Q.head = Q.tail + 1$, the queue is full, and if we attempt to enqueue an element then the queue overflows.

Algorithm 1 Enqueue and dequeue

```
function ENQUEUE($Q, x$)
    if $Q.head == Q.tail + 1$ or ($Q.head == 1 and Q.tail == Q.length$) then
        $error"overflow"$
    end if
    $Q[Q.tail] = x$
    if $Q.tail == Q.length$ then
        $Q.tail = 1$
    else
        $Q.tail = Q.tail + 1$
    end if
end function

function DEQUEUE($Q$)
    if $Q.head == Q.tail == 1$ then
        $error"underflow"$
    end if
    $x = Q[Q.head]$
    if $Q.head == Q.length$ then
        $Q.head = 1$
    else
        $Q.head = Q.head + 1$
    end if
    return $x$
end function
```

5 Exercise 10.1-6

Algorithm 2 Implement a queue using two stacks

```
function ENQUEUE-STACK($stack1, stack2, value$)
    $stack1.push(value)$
end function

function DEQUEUE-STACK($stack1, stack2$)
    if $stack2$ is not empty then
        return $stack2.pop()$
    else
        while $stack1$ is not empty do
            $stack2.push(stack1.pop())$
        end while
        if $stack2$ is empty then
            return $-1$
        else
            return $stack2.pop()$
        end if
    end if
end function
```

The operation ENQUEUE-STACK is same as pushing an element on to stack 1 and it takes $O(1)$. For DEQUEUE-STACK. we pop an element from stack 2 if stack 2 is not empty; Otherwise, for each element in stack 1 we pop it off, then push it on to stack 2 and finally pop the top item from stack 2. This operation is $O(n)$ in the worst-case.

6 Exercise 10.1-7

Algorithm 3 Implement a stack using two queues

```
function PUSH-QUEUE($queue1, queue2, value$)
    if $queue2$ is empty then
        ENQUEUE($queue1, value$)
    else
        ENQUEUE($queue2, value$)
    end if
end function
```

```
function Pop-Queue($queue1, queue2$)
    if $queue1$ is empty and $queue2$ is empty then
        return $-1$
    else
        $DelData \leftarrow 0$
        if $queue2$ is not empty then
            $len \leftarrow queue2.length$
            while $len > 1$ do
                $DelData \leftarrow$ Dequeue($queue2$)
                Enqueue($queue1, DelData$)
            end while
            $DelData \leftarrow$ Dequeue($queue2$)
            return $DelData$
        end if
        if $queue1$ is not empty then
            $len \leftarrow queue1.length$
            while $len > 1$ do
                $DelData \leftarrow$ Dequeue($queue1$)
                Enqueue($queue2, DelData$)
            end while
            $DelData \leftarrow$ Dequeue($queue1$)
            return $DelData$
        end if
    end if
end function
```

The implementing a stack using two queues, where pop takes $O(n)$ time and push takes $O(1)$ time. The operation PUSH-QUEUE is same as enqueueing each element as we push it. For POP-QUEUE, we dequeue each element from one of the queues and enqueue another queue in the order, but stopping just before the last element. Then, the single element left in the original queue.

7 Exercise 10.2-2

Algorithm 4 Implement a stack using a single linked list

```
function Push-List($L, x$)
    $x.next = L.head$
    if $L.head \neq NIL$ then
        $L.head.prev = x$
    end if
    $L.head = x$
    $x.prev = NIL$
end function
```

```
function Pop-List($L, L.head$)
    $x \leftarrow L.head$
    if $x.prev \neq NIL$ then
        $x.prev.next = x.next$
    else
        $L.head \leftarrow x.next$
    end if
    if $x.next \neq NIL$ then
        $x.next.prev = x.prev$
    end if
end function
```

The PUSH-LIST operation is same as linked list insertion, and POP-LIST operation is same as linked list deletion.

8 Exercise 10.2-6

This can be implemented by doubly linked list. If both sets are a doubly linked list, we just link the last element of the first list to the first element in the second list. Wherever we have a reference to NIL in list code, we replace it by a reference to the sentinel L.nil lies between the head and tail. .

```
function List-Union($L1, L2$)
    $L2.nil.next.prev = L1.nil.prev$
    $L1.nil.prev.next = L2.nil.next$
    $L2.nil.prev.next = L1.nil$
    $L1.nil.prev = L2.nil.prev$
end function
```

9 Exercise 10.4-2

Algorithm 5 Prints out the key of each node in the binary tree

```
function Inorder-Tree-Walk($x$)
    if $x \neq NIL$ then
        Inorder-Tree-Walk($x.left$)
        print $x.key$
        Inorder-Tree-Walk($x.right$)
    end if
end function
```

10 Problem 10-1 Comparisons among lists

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
SEARCH(L,x)	O(n)	O(n)	O(n)	O(n)
INSERT(L,x)	O(1)	O(1)	O(1)	O(1)
DELETE(L,x)	O(n)	O(n)	O(1)	O(1)
SUCCESSOR(L,x)	O(n)	O(1)	O(n)	O(1)
MINIMUM(L)	O(n)	O(1)	O(n)	O(1)
MAXIMUM(L)	O(n)	O(n)	O(n)	O(1)

11 Exercise 12.2-5

Suppose the node x has two children, its predecessor is the maximum value in its left subtree and its successor is the minimum value in its right subtree. If the successor of node x had s left child then it wouldn't be the minimum element. Depending on the property of binary tree, the successor must not have a left child. Similarly, the predecessor must not have a right child.

12 Exercise 12.2-7

Algorithm 6 Finding the successor

```
function TREE-MINIMUM($x$)
    while $x.left \neq NIL$ do
        $x = x.left$
    end while
    return $x$
end function

function TREE-SUCCESSOR($x$)
    if $x.right \neq NIL$ then
        return TREE-MINIMUM($x.right$)
    end if
    $y \leftarrow x.p$
    while $y \neq NIL$ and $x == y.right$ do
        $x = y$
        $y = x.p$
    end while
    return $y$
end function
```

A call to TREE-MINIMUM followed by $n-1$ calls to TREE-SUCCESSOR performs the same procedure as INORDER-TREE-WALK does. The algorithm traverse each edge at most twice (once going down the tree and once going up),

which takes $\Theta(n)$ time. The only time the tree is traversed downward is in code of TREE-MINIMUM, and the only time the tree is traversed upward is in code of TREE-SUCCESSOR when we look for the successor of a node that has no right subtree. Therefore, this algorithm runs in $\Theta(n)$ time.

13 Exercise 12.3-3

Algorithm 7 Sort a given set of n numbers by first building a binary search tree

```
function TREE-INSERT($T, x$)
    $y \leftarrow NIL$
    $x \leftarrow T.root$
    while $x \neq NIL$ do
        $y \leftarrow x$
        if $z.key < x.key$ then
            $x = x.left$
        else
            $x = x.right$
        end if
    end while
    $z.p \leftarrow y$
    if $y == NIL$ then
        $T.root = x$
    else
        if $z.key < y.key$ then
            $y.left = z$
        else
            $y.right = z$
        end if
    end if
end function

function INORDER-TREE-WALK($x$)
    if $x \neq NIL$ then
        INORDER-TREE-WALK($x.left$)
        print $x.key$
        INORDER-TREE-WALK($x.right$)
    end if
end function

function TREE-SORT($x$)
    let T be an empty binary search tree
    for $i = 1$ to $n$ do
        TREE-INSERT($T, x$)
    end for
```

```
37:     Inorder-Tree-Walk($T.root$)
38: end function
```

The worst-case is that we were inserting in already sorted order, therefore, the worst-case takes $\Theta(n^2)$. In the best-case, the tree is balanced, which means the height doesn't exceed $O(lg(n))$, so the best-case takes $\Theta(nlg(n))$.