CS5800: Algorithms — — Virgil Pavlu

Homework 7cd
Submit via Gradescope

Name: Xuran Feng
Collaborators:

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

1. ***(30 points, Mandatory)*** *Write up a max two-page summary of all concepts and techniques in CLRS Chapter 10 (Simple Data Structures)*

Stack:

- LIFO(last in, first out)
- PUSH() – > to insert elements
- POP() – > to delete elements
- Implementation using an size n array:
  stack consists of elements S[1..S.top] where S.top indexes the most recently inserted element
  when S.top=0: the stack is empty and POP() will cause an underflow
  when S.top=n: the stack is full and PUSH() will cause an overflow
- PUSH(S,x)-O(1):
  S.top=S.top+1
  S[S.top]=x
- POP(S)-O(1):
  if(S.top==0) error ”underflow”
  else S.top=S.top-1
  return S[S.top+1]

Queue:

- FIFO(first in, first out)
- ENQUEUE() – > to insert elements
- DEQUEUE() – > to delete elements
- Implementation using an size n array:
  queue Q[1..n] consists of Q.head and Q.tail indexing the position element inserted at and element dequeued from respectively
  when Q.head=Q.tail: the stack is empty and DEQUEUE() will cause an underflow
  when Q.head=Q.tail+1: the stack is full and ENQUEUE() will cause an overflow
- ENQUEUE(Q,x)-O(1):
  Q[Q.tail]=x
  if Q.tail==Q.length: Q.tail=1
  else: Q.tail=Q.tail+1
- DEQUEUE(Q)-O(1):
  x=Q[Q.head]
  if Q.head==Q.length: Q.head=1
  else: Q.head=Q.head+1
  return x

LinkedLists:

- Objects are arranged in linear order and the order is determined by a pointer.
- singly linked list/doubly linked list
- sorted linked list/circular linked list
- LIST-SEARCH(L,k):
  x=L.head
  while x!=NULL and x.key!=k: x=x.next
  return x

- LIST-INSERT(L,x)-O(1):
    x.next=L.head
    if L.head!=NULL: L.head.prev=x
    L.head=x
    x.prev=NULL
- LIST-DELETE(L,x)-O(1):
    if x.prev!=NULL: x.prev.next=x.next
    else: L.head=x.next
    if x.next!=NULL: x.next.prev=x.prev
- Sentinel: dummy object than can simplify boundary conditions

Tree:
- Binary tree: has at most two children
- Unbounded branching: can be converted into a binary tree

2. ***(30 points, Mandatory)*** *Write up a max two-page summary of all concepts and techniques in CLRS Chapter 12 (Binary Search Trees)*

- Search tree structure supports many dynamic-set operations and can be used as a dictionary and a priority queue.
- Operations on a binary search tree is proportional to the tree height-O(h).Red-black tree ensures height O(lgn).
- BST property: nodes in the left subtree of x then node.key≤ x.key, nodes in the right subtree of x then node.key≥x.key
- inorder tree traversal– >print out BST in sorted order

RECURSIVE-INORDER-TREE-WALK(x)-$\theta(n)$:
```
    if x != NIL
        INORDER-TREE-WALK(x.left)
        print x.key
        INORDER-TREE-WALK(x.right)
```
RECURSIVE-TREE-SEARCH(x,k)-O(height):
```
    if x==NIL or k==x.key
        return x
    if k<x.key
        return RECURSIVE-TREE-SEARCH(x.left,k)
    else return RECURSIVE-TREE-SEARCH(x.right,k)
```
ITERATIVE-TREE-SEARCH(x,k):
```
    while x!=NIL and k!=x.key
        if k<x.key
            x=x.left
        else x=x.right
    return x
```
TREE-MINIMUM(x)-O(height):
```
    while x.left!=NIL
        x=x.left
    return x
```

```
TREE-MAXIMUM(x)-O(height):
    while x.right!=NIL
        x=x.right
    return x
TREE-SUCCESSOR(x)-O(height):
    if x.right!=NIL
        return TREE-MINIMUM(x.right)
    y=x.p
    while y!=NIL and x==y.right
        x=y
        y=y.p
    return y
```

Predecessor is opposite to successor

```
TREE-PREDECESSOR(x)-O(height):
    if x.left!=NIL
        return TREE-MAXIMUM(x.left)
    y=x.p
    while y!=NIL and x==y.left
        x=y
        y=y.p
    return y
```

Insertion: use another trailing pointer y to track parent

```
TREE-INSERT(T,z)-O(height):
    y=NIL
    x=T.root
    while x!=NIL
        y=x
        if z.key<x.key
            x=x.left
        else x=x.right
    z.p=y
    if y==NIL
        T.root=z
    else if z.key<y.key
        y.left=z
    else y.right=z
TREE-DELETION(T,z)-O(height):
    if z.left==NIL
        TRANSPLANT(T,z,z.right)
    else if z.right==NIL
        TRANSPLANT(T,z,z.left)
    else y=TREE-MINIMUM(z.right)
        if y.p!=z
            TRANSPLANT(T,y,y.right)
            y.right=z.right
```

```
            y.right.p=y
        TRANSPLANT(T,z,y)
        y.left=z.left
        y.left.p=y
```

Subroutine that replaces the subtree rooted at node u with the subtree rooted at node v

```
TRANSPLANT(T,u,v):
    if u.p==NIL
        T.root=v
    else if u==u.p.left
        u.p.left=v
    else u.p.right=v
    if v!=NIL
        v.p=u.p
```

3. ***(10 points)*** *Exercise 10.1-1.*

The empty stack has six cells to be filled in – >[, , , , ,], PUSH(S,4) – > [4, , , , ,], PUSH(S,1) – > [4,1, , , ,], PUSH(S,3) – > [4,1,3, , ,], POP(S) – > [4,1, , , ,], PUSH(S,8) – > [4,1,8, , ,], POP(S) – > [4,1, , , ,].

4. ***(10 points)*** *Exercise 10.1-4.*

- ENQUEUE(Q,x):

```
 if(Q.head==Q.tail+1): error ”overflow”
 else: Q[Q.tail]=x
        if Q.tail==Q.length: Q.tail=1
        else: Q.tail=Q.tail+1
```

- DEQUEUE(Q):

```
    if(Q.head==Q.tail): error ”underflow”
    else: x=Q[Q.head]
        if Q.head==Q.length: Q.head=1
        else: Q.head=Q.head+1
        return x
```

5. ***(10 points)*** *Exercise 10.1-6.*

let stack1 and stack2 be two empty stacks.

- ENQUEUE(x):

```
    while(!isEmpty(stack1)) push(stack2, pop(stack1));
    push(stack1,x);
    while(!isEmpty(stack2)) push(stack1, pop(stack2));
```

- DEQUEUE():

```
    return pop(stack1);
```

Use a second stack2 to store all elements in reversed order and then pushed back into stack1, so

stack1 behaves like a queue that has elements in correct order. The running time of DEQUEUE is O(1) and ENQUEUE is O(n) because each time elements need to pop out and then push back again.

6. ***10 points*** *Exercise 10.1-7.*

let queue1 and queue2 be two empty queues.

- PUSH(x):

```
enqueue(queue2,x);
while(!isEmpty(queue1)) enqueue(queue2,dequeue(queue1));
swap(queue1,queue2);
```

- POP():

```
return pop(queue1);
```

Use a second queue2 to store all elements in reversed order and then swap queue1 and queue2, so queue1 behaves like a stack that has elements in correct order. The running time of POP is O(1) and PUSH is O(n) because each time elements need to dequeue and then enqueue again.

7. ***(10 points)*** *Exercise 10.2-2.*

let L be a singly linked list with a sentinel node dummyHead

- PUSH(L,x):

```
x.next=dummyHead.next
dummyHead.next=x
```

- POP(L):

```
if(dummyHead.next==NULL) error ”underflow”
else:
    x=dummyHead.next
    dummyHead.next=dummyHead.next.next
    return x
```

8. ***(10 points)*** *Exercise 10.2-6.*

Assume we have two singly linked lists and they are respectively used to store elements in S1 and S2. What is needed is to connect the last element of s1 with the first element of s2. First we need to iterate to the last element of s1.

- UNION(s1,s2):

```
last=ITERATE(s1)
if s2 has a sentinel: last.next=s2.head.next
else: last.next=s2.head
```

- ITERATE(s1):

```
x=s1.head
while(x.next!=NULL): x=x.next
return x
```

9. ***(10 points)*** *Exercise 10.4-2.*

```
PRINT-TREE(T):
    if(T==NULL): return;
    PRINT-TREE(T.left)
    output T.key
    PRINT-TREE(T.right)
```

Runs in $\theta(n)$ because each object gets printed once and no edges are walked more than twice.

10. ***(10 points)*** *Problem 10-1.*

	unsorted singly	sorted singly	unsorted doubly	sorted doubly
SEARCH	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
INSERT	O(1)	$\theta(n)$	O(1)	$\theta(n)$
DELETE	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
SUCCESSOR	$\theta(n)$	$\theta(1)$	$\theta(n)$	$\theta(1)$
PREDECESSOR	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(1)$
MINIMUM	$\theta(n)$	$\theta(1)$	$\theta(n)$	$\theta(1)$
MAXIMUM	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(1)$

For Search operation: all kinds of linked lists need to go through all nodes to find the wanted key; for Insert operation: unsorted lists only costs O(1) time because only some constant pointers need to be adjusted, sorted lists however need to first call Search operation to find the appropriate position to insert; for Delete operation, in the worst case if we want to delete some key, we need to call Search operation to locate the position; for Successor operation, sorted linked lists can simply refer to the next node behind that key and unsorted linked lists need to iterate whole lists to find; for Predecessor, because sorted singly linked list can only go next instead of going bidirectionally, so it needs to iterate whole list again while sorted doubly can refer to both directions; for Minimum operation, sorted linked list can simply return the first element within constant time; for Maximum operation, because doubly linked list can go back to last element within constant time while sorted singly linked list still need to iterate the whole list till the end.

11. ***(10 points)*** *Exercise 12.2-5.*

The node x has two children, then the successor is the minimum key of its right child, if the successor has left child meaning successor cannot be the minimum key of x's right subtree because the left child key of the successor must be less than the successor so there is a contradiction; same applies for predecessor, because predecessor is the maximum key of x's left subtree, if the predecessor has right child, then current predecessor cannot be the maximum key of x's left subtree because the right child key of the predecessor must be great than the predecessor so there is a contradiction.

12. ***(10 points)*** *Exercise 12.2-7.*

Since INORDER-TREE-WALK visits all n nodes of the subtree, $T(n) = \Omega(n)$ is valid, still needs to show $T(n) = O(n)$. We can prove this by showing each edge is walked no more than twice. According to the Successor pseudocode, if the current node has no right child, then we need to traverse up back to a qualified lowest ancestor, in other words, the edge that is traversed first to reach current node must be traversed again to float up and will only be retraversed once. For each edge there will only be a back-and-forth walk and will cost 2*(n-1) running time thus it is O(n).

13. ***(10 points)*** *Exercise 12.3-3*

TREE-INSERT has a runnint time of O(height), and the best case of O(height) is a almost balanced BST which is $\theta(logn)$ ($2^h = \theta(n)$) and thus inserting n items costs n*$\theta(logn)$=$\theta(nlogn)$ time; the worst case of O(height) is a linearly sorted BST which is $\theta(n)$ and thus inserting n items costs n*$\theta(n)$=$\theta(n^2)$ time.

14. ***(Extra Credit)*** *Problem 12-3.*

15. ***(Extra Credit)*** *Problem 10-2.*

16. ***(Extra Credit)*** *Problem 15-6.*