Dayou Du
N13825075
dd2645@nyu.edu

Homework 2 - SA
Fundamental Algorithms

2017-09-24

# Ex 2.1

## a)

51, 6, 3, 5, 14, 7, 11
I got this question CORRECT

## b)

3, 5, 6, 7, 11, 14, 51
I got this question CORRECT

## c)

5, 3, 11, 7, 14, 6, 51
I got this question CORRECT

# Ex 2.2

Note: Here we use function $eval(op, x, y)$, which calculates $y\ op\ x$ (e.g. $eval(-, 2, 3)$ calculates $3 - 2$ and gives out $1$).

**Algorithm 1** Ex 2.2

**Input:** The current root node $R$
**Output:** The value of the subtree rooted at $R$

```
function EvaluateTree(;;R)
    if R is a leaf then
        return R.val
    else
        return eval(R.op, EvaluateTree(R.right), EvaluateTree(R.left))
    end if
end function
```

I got this question CORRECT.
Note: Here we pass R by reference to ensure we didn't copy the whole tree. If we have real "pointer"type, then we'd better pass R as value.

# Ex 2.3

## a)

**Algorithm 2** Ex 2.3 a)

**Input:** The current root node $R$
**Output:** The smallest value in the $.val$ fields of the subtree rooted by $R$

**function** $findSmallestVal(;;R)$
    $R.small \leftarrow R.val$
    **for each** child $c$ of $R$ **do**
        $childVal \leftarrow findSmallestVal(c)$
        **if** $childVal < R.small$ **then**
            $R.small \leftarrow childVal$
        **end if**
    **end for**
    **return** $R.small$
**end function**

I got this question CORRECT

## b)

**Algorithm 3** Ex 2.3 b)

**Input:** The current root node $R$
**Output:** The node pointer of which have smallest value in the $.val$ fields of the subtree rooted by $R$

**function** $findSmallestValandPointer(;;R)$
    $R.which \leftarrow R$
    **for each** child $c$ of $R$ **do**
        $childSmallestValPointer \leftarrow findSmallestValandPointer(c)$
        **if** $childSmallestValPointer.val < R.which.val$ **then**
            $R.which \leftarrow childSmallestValPointer$
        **end if**
    **end for**
    $R.small \leftarrow R.which.val$
    **return** $R.which$
**end function**

I got this question CORRECT.
Note: Since x.which always point to the vertex which have the smallest value in the subtree rooted at x. Then we have x.small == x.which.val

# EX 2.4

## a)

**Algorithm 4** Ex 2.4 a)

**Input:** The current root node $R$
**Output:** Postorder printed $.val$ fields

```
procedure PostorderTraversal(;;R)
    for each child c of R do
        PostorderTraversal(c)
    end for
    print R.val
end procedure
```

I got this question CORRECT

## b)

**Algorithm 5** Ex 2.4 b)

**Input:** The current root node $T$ and a number $X$
**Output:** Rotated Tree and updated $X$

```
procedure Rotate(;;T, X)
    for each child c of R do
        Rotate(c, X)
    end for
    temp ← T.val
    T.val ← X
    X ← temp
end procedure
```

I got this question CORRECT

## Ex 2.5

Note: Here we use function $eval(op, x, y)$, which calculates $y\ op\ x$ (e.g. $eval(-, 2, 3)$ calculates $3 - 2$ and gives out $1$).

**Algorithm 6** Ex 2.5

**Input:** $L$ is a stack that represents an arithmetic expression as described
**Output:** Returns the evaluation of $L$

**function** $StackEval(;; L)$
    $x \leftarrow PopFrom(L)$
    **if** $x$ is a number **then return** $x$
    **else**
        **return** $eval(x, StackEval(L), StackEval(L))$
    **end if**
**end function**

I got this question CORRECT

## EX 2.6

### a)

S and T are exactly reversed lists.
I got this question CORRECT

### b)

In list S, we print the child nodes from left to right, and then we print the root. In list T, we print the root, and then print child nodes from right to left. This holds true when we processing each node. Thus, they are exactly reversed.
I got this question CORRECT

## c)

Note: Here we use function $eval(op, x, y)$, which calculates $y\ op\ x$ (e.g. $eval(-, 2, 3)$ calculates $3 - 2$ and gives out 1).

**Algorithm 7** Ex 2.6c

**Input:** $L$ is a stack that represents an arithmetic expression as described
**Output:** Returns the evaluation of $L$

**function** $StackEval(;;L)$
    create an empty stack $tempStack$
    **while** $L$ is not empty **do**
        $x \leftarrow PopFrom(L)$
        **if** $x$ is a number **then**
            $tempStack.Push(x)$
        **else**
            $num1 \leftarrow PopFrom(tempStack)$
            $num2 \leftarrow PopFrom(tempStack)$
            $tempStack.Push(eval(x, num1, num2))$
▷ we can also write them in one line:
▷ $tempStack.Push(eval(x, PopFrom(tempStack), PopFrom(tempStack)))$
        **end if**
    **end while**
    **return** $PopFrom(tempStack)$
**end function**

I got this question CORRECT I'm not quite understand why in the solution sheet there's an initial move from original Stack top to the temp Stack. I know that the first element must be a number, but I think the while-loop can do it. I guess, do the initial move can save one If judgment?

## d)

Note: Here we use function $eval(op, x, y)$, which calculates $y\ op\ x$ (e.g. $eval(-, 2, 3)$ calculates $3 - 2$ and gives out 1).

**Algorithm 8** Ex 2.6d

**Input:** $L$ is a doubly linked list that represents an arithmetic expression as described
**Output:** Returns the evaluation of $L$

**function** $ListEval(;;L)$
    **while** $L.next \neq Nil$ **do** ▷ It's same to say $L.next \neq Nil$ or $L \neq EOL$, the later one reads better.
        **if** $L.val$ is a number **then**
            $L \leftarrow L.next$
        **else**
            $L.val \leftarrow eval(L.val, L.prev.val, L.prev.prev.val))$
            $L.prev \leftarrow L.prev.prev.prev$
▷ *Actually this judgment(3 lines below) is unnecessary if would just like to get the correct answer, because we will never try to access .prev.next. Do it if we would like to maintain the doubly-link structure.*
            **if** $L.prev \neq Nil$ **then**
                $L.prev.next \leftarrow L$
            **end if**
            **if** $L.next \neq Nil$ **then** ▷ *Check whether we have already reached head. Unnecessary.*
                $L \leftarrow L.next$ ▷ It's same to say $L.next \neq Nil$ or $L \neq EOL$, the later one reads better.
            **end if**
        **end if**
    **end while**
    **return** $L.val$ ▷ *The return value should be $L.prev.val$. Since at last, L point to EOL record ($EOL.next = Nil$ and $EOL.prev$ points to the last operator), thus answer is stored in the prev record of EOL*
**end function**

Current solution returned WRONG position of the record. After the last computation, $L$ points to $EOL$ record, and the answer is stored in $EOL.prev$, which is actually where the last operator was. Thus we should return $L.prev.val$ instead of $L.val$. The fault I made is that I didn't carefully checked the last return statement.
After fix and some clean up, the correct code should be:

**Algorithm 9** Ex 2.6d:revised

**Input:** $L$ is a doubly linked list that represents an arithmetic expression as described
**Output:** Returns the evaluation of $L$

```
function ListEval(; ; L)
    while L ≠ EOL do
        if L.val is a number then
            L ← L.next
        else
            L.val ← eval(L.val, L.prev.val, L.prev.prev.val))
            L.prev ← L.prev.prev.prev
            if L.prev ≠ Nil then
                L.prev.next ← L
            end if
            L ← L.next
        end if
    end while
    return L.prev.val
end function
```

Note: Actually there could be a bug in the solution sheet if we consider a single number also a legal expression: It didn't initialize B at the beginning of the program, thus will not work for a single number.

## Ex 2.8

**Algorithm 10** Ex 2.8

```
Input: The current root node R
Output: The number of leafs in the subtree rooted at R
function CountLeafs(; ; R)
    if R is a leaf then
        R.numb ← 1
    else
        R.numb ← 0
        for each child c of R do
            R.numb ← R.numb + CountLeafs(c)
        end for
    end if
    return R.numb
end function
```

I got this question CORRECT

## Ex 2.9

**Algorithm 11** Ex 2.9

```
Input: The current root node R
Output: The max depth in the subtree rooted at R, meanwhile change the .dis fields in the subtree as required
function CountDepth(; ; R)
    R.dis ← 0
    for each child c of R do                    ▷ Work Correctly for leafs
        cDis ← CountDepth(c)
        if cDis > R.dis then
            R.dis ← cDis
        end if
    end for
    return R.dis + 1
end function
```

I got this question CORRECT
Note: I moved the ”plus one”to the return statement.

# Ex 2.10

**Algorithm 12** Ex 2.10

**Input:** The current root node $R$
**Output:** The max depth in the subtree rooted at $R$, meanwhile change the $.dis1$ fields and $.dis2$ fields in the subtree as required

```
function CountDepth(; ; R)
    R.dis1 ← 0
    R.dis2 ← −∞
    for each child c of R do
        cDis ← CountDepth(c)
        if cDis > R.dis1 then
            R.dis2 ← R.dis1
            R.dis1 ← cDis
        else
            if cDis > R.dis2 then
                R.dis2 ← cDis
            end if
        end if
    end for
    return R.dis1 + 1
end function
```

I got this question CORRECT
Note:

1. I moved the ”plus one”to the return statement.
2. Since inner vertex always have at lease one child, $.dis1$ will not be $-\infty$. Thus it's fine to initialize it with 0

# Ex 2.19

## a)

Yes.
I got this question CORRECT

## b)

No.
I got this question CORRECT

## c.1)

1. At beginning we call $Parent(T, Nil)$
2. Here we assume that we have an overloaded function $print$, which can correctly accept Vertex pointers, dealing with Nil, and print the name of vertex.

**Algorithm 13** Ex 2.19c1

**Input:** The root node $R$
**Output:** The vertex names in pre-order traversal

**procedure** $Parent$(; ; $v, pv$)
    print $(v, pv)$
    **for each** child $c$ of $v$ **do**
        $Parent(c, v)$
    **end for**
**end procedure**

I got this question CORRECT.
Note: Here we pass $v$ and $pv$ by reference to ensure we didn't copy the whole tree. If we have real "pointer"type, then we'd better pass them by value.

## c.2)

1. At beginning we call $Parent(S, Nil)$
2. Here we assume that we have an overloaded function $print$, which can correctly accept Vertex pointers, dealing with Nil, and print the name of vertex.

**Algorithm 14** Ex 2.19c2

**Input:** The root node $R$
**Output:** The vertex names in pre-order traversal

**procedure** $Parent$(; ; $v, pv$)
    **if** $v == Nil$ **then**
        **return**
    **else**
        print $(v, pv)$
        $Parent(v.left, V)$
        $Parent(v.right, V)$
    **end if**
**end procedure**

I got this question CORRECT.
Note: It's better to judge the leafs before we call an additional function

### d.1)

1. At beginning we call $Parent(T, Nil)$
2. Here we assume that we have an overloaded function $print$, which can correctly accept Vertex pointers, dealing with Nil, and print the name of vertex.

**Algorithm 15** Ex 2.19d1

**Input:** The root node $R$
**Output:** The vertex names in post-order traversal

```
procedure Parent(; ; v, pv)
    for each child c of v do
        Parent(c, v)
    end for
    print (v, pv)
end procedure
```

I got this question CORRECT.
Note: Here we pass $v$ and $pv$ by reference to ensure we didn't copy the whole tree. If we have real "pointer"type, then we'd better pass them by value.

### d.2)

Note:

1. At beginning we call $Parent(S, Nil)$
2. Here we assume that we have an overloaded function $print$, which can correctly accept Vertex pointers, dealing with Nil, and print the name of vertex.
3. Actually we are doing an in-order traversal for binary tree S.

**Algorithm 16** Ex 2.19d2

**Input:** The root node $R$
**Output:** The vertex names in post-order traversal for original tree T, that is in-order traversal for S

```
procedure Parent(; ; v, pv)
    if v == Nil then
        return
    else
        Parent(v.left, V)
        print (v, pv)
        Parent(v.right, V)
    end if
end procedure
```

I got this question CORRECT.
Note: It's better to judge the leafs before we call an additional function

### e)

Note:

1. At beginning we call $Parent(T, Nil)$
2. Here we assume that we have an overloaded function $print$, which can correctly accept Vertex pointers, dealing with Nil, and print the name of vertex.

---

**Algorithm 17** Ex 2.19e

**Input:** The root node $R$
**Output:** The vertex names in post-order traversal

**procedure** $PostorderTraversal(;;R)$
    **for each** child $c$ of $R$ **do**
        $Parent(c)$
        print $(c, R)$
    **end for**
**end procedure**

---

I got this question CORRECT.

## Ex 2.20

---

**Algorithm 18** Ex 2.20

**Input:** The currant head pointer L
**Output:** The list that removed the items which $.data == 0$

**procedure** $Clean(;;L)$
    **if** $L == Nil$ **then**
        **return**
    **end if**
    **if** $L.data == 0$ **then** ▷ Current item should be removed
        $temp \leftarrow L$ ▷ Clean up removed item. Not necessary
        $L \leftarrow L.next$
        $temp.next \leftarrow Nil$ ▷ Clean up removed item. Not necessary
        $Clean(L)$
    **else**
        $Clean(L.next)$
    **end if**
**end procedure**

---

I got this question CORRECT.
Note: The solution sheet didn't cleanup the removed items. As the example on text book of delete one item from list, we should reset the pointer of the removed item to Nil in order to clean up. However, absolutely ,we will get the correct list if we don't do the clean up.

# Ex 2.23

## a)

At beginning we call $SelectionSort_{recursive1}(n, Data[1..n])$

**Algorithm 19** Ex 2.23a, Selection Sort, recursive outer loop

**Input:** The number of elements $n$, the numbers $Data[1..n]$
**Output:** $Data[1..n]$ in nondecreasing order

**procedure** $SelectionSort_{recursive1}(n; ; Data[1...n])$
    **if** $n == 1$ **then**
        **return**
    **end if**
    $IndexOfBiggest \leftarrow 1$
    $Biggest \leftarrow Data[1]$
    **for** $TestDex \leftarrow 2 : n$ **do**
        **if** $Biggest < Data[TestDex]$ **then**
            $Biggest \leftarrow Data[TestDex]$
            $IndexOfBiggest \leftarrow TestDex$
        **end if**
    **end for**
    $Swap(Data[IndexOfBiggest], Data[n])$
    $SelectionSort_{recursive1}(n-1, Data[1...n-1])$
**end procedure**

I got this question CORRECT.

## b)

At beginning we call $SelectionSort_{recursive2}(n, Data[1..n])$

**Algorithm 20** Ex 2.23b, Selection Sort, Print the sorted array and Restore to original configuration

**Input:** The number of elements $n$, the numbers $Data[1..n]$
**Output:** $Data[1..n]$ in nondecreasing order

1: **procedure** $SelectionSort_{recursive2}(n; ; Data[1...n])$
2:     **if** $n == 1$ **then**
3:         print $Data[1]$     ▷ We can also print the whole array Here.
4:         **return**
5:     **end if**
6:     $IndexOfBiggest \leftarrow 1$
7:     $Biggest \leftarrow Data[1]$
8:     **for** $TestDex \leftarrow 2 : n$ **do**
9:         **if** $Biggest < Data[TestDex]$ **then**
10:             $Biggest \leftarrow Data[TestDex]$
11:             $IndexOfBiggest \leftarrow TestDex$
12:         **end if**
13:     **end for**
14:     $Swap(Data[IndexOfBiggest], Data[n])$
15:     $SelectionSort_{recursive2}(n-1, Data[1...n-1])$
16:     print $Data[n]$     ▷ Note: We can also print the whole array in line 2.
17:     $Swap(Data[IndexOfBiggest], Data[n]$
18: **end procedure**

I got this question CORRECT.
Note: We can also print the whole array in line 2.

## Ex 2.30

```
#include "common.h"
using namespace std;
void DFS(vector<vector<int>>& childTable, int rootId)
{
        for (int i = 1; i <= childTable[rootId][0]; ++i)
                DFS(childTable, childTable[rootId][i]);
        printf_s("%d, ", rootId);
}
```

There's no solution for this question. But I think I got this question CORRECT.