Data Structures and Algorithms
Assignment 4 / **Solution**

Apr 4, 2016

Abstract Data Types (ADT)

Task 1

```
#include <stdlib.h>
#include <stdio.h>

struct stack {
    int key;
    struct stack* next;
};

int isEmpty(struct stack* s)
{
    return !s;
}

void push(struct stack** s, int key)
{
    struct stack* stackNode = (struct stack*) malloc(sizeof(struct
        stack));
    stackNode->key = key;
    stackNode->next = NULL;
    stackNode->next = *s;
    *s = stackNode;
}

int pop(struct stack** s)
{
    if (isEmpty(*s))
        return -1;
    struct stack* temp = *s;
    *s = (*s)->next;
    int popped = temp->key;
    free(temp);

    return popped;
}
```

```
void printStack(struct stack* s) {
    struct stack* temp = s;
    while (temp != NULL) {
        printf("%d ", temp->key);
        temp = temp->next;
    }
    printf("\n");
}

int evaluatePostfix(char* exp)
{
    int i;
    int val1, val2;
    // Create an empty stack
    struct stack* s = NULL;

    // Scan all characters one by one
    for (i = 0; exp[i] != '\0'; i++)
    {
        if ('0' <=exp[i] && exp[i] <='9')
            // If the scanned character is an operand (number) push it to
            // the stack.
            push(&s, exp[i] - '0'); // TODO atoi
        else
        {
            // If the scanned character is an operator, pop two
            // elements from the stack and apply the operator
            val1 = pop(&s);
            val2 = pop(&s);
            switch (exp[i])
            {
                case '+': push(&s, val2 + val1); break;
                case '-': push(&s, val2 - val1); break;
                case '*': push(&s, val2 * val1); break;
                case '/': push(&s, val2/val1); break;
            }
        }
    }
    return pop(&s);
}

void main() {
    int val;
    struct stack* s = NULL;

    push(&s, 3);
    push(&s, 4);
    push(&s, 6);
    push(&s, 9);
    push(&s, 10);

    val = pop(&s);
    printf("%d\n", val);
    val = pop(&s);
```

```
    printf("%d\n", val);

    push(&s, 4);
    push(&s, 17);
    push(&s, 30);

    printStack(s);
}

// Linux, Mac: gcc task1.c -o task1; ./task1
// Windows: gcc task1.c -o task1; task1
```

Task 2. Evaluation of Postfix Expression

```
#include <stdlib.h>
#include <stdio.h>

// ... include code from Task 1 ...

int evaluatePostfix(char* exp)
{
    int i;
    int val1, val2;
    // Create an empty stack
    struct StackNode* s = NULL;

    // Scan all characters one by one
    for (i = 0; exp[i] != '\0'; i++)
    {
        if ('0' ≤exp[i] && exp[i] ≤'9')
            // If the scanned character is an operand (number) push it to
            // the stack.
            push(&s, exp[i] - '0'); // TODO atoi
        else
        {
            // If the scanned character is an operator, pop two
            // elements from the stack and apply the operator
            val1 = pop(&s);
            val2 = pop(&s);
            switch (exp[i])
            {
                case '+': push(&s, val2 + val1); break;
                case '-': push(&s, val2 - val1); break;
                case '*': push(&s, val2 * val1); break;
                case '/': push(&s, val2/val1); break;
            }
        }
    }
    return pop(&s);
}

void main() {
```

```
    char expr[100];
    int val;
    struct StackNode* s = NULL;

    printf("Type a postfix expression\n");
    scanf("%[0-9+/-*]s", expr);
    printf("Result: %d\n", evaluatePostfix(expr));
}

// Linux, Mac: gcc task2.c -o task2; ./task2
// Windows: gcc task2.c -o task2; task2
```

Binary Trees

Task 3. Insert and Deleting nodes

```
#include<stdlib.h>
#include<stdio.h>

struct TreeNode {
    int key;
    struct TreeNode* lChild;
    struct TreeNode* rChild;
};

/* See slides, there is also a recursive variation. */
void insert(int key, struct TreeNode ** root) {
    struct TreeNode *nodeToInsert = malloc(sizeof(struct TreeNode));
    nodeToInsert->key = key;
    nodeToInsert->lChild = NULL;
    nodeToInsert->rChild = NULL;

    struct TreeNode *oneDelayed = NULL;
    struct TreeNode *insertPlace = *root;
    while (insertPlace != NULL) {
        oneDelayed = insertPlace;
        if (insertPlace->key < nodeToInsert->key)
            insertPlace = insertPlace->rChild;
        else
            insertPlace = insertPlace->lChild;
    }
    if (oneDelayed == NULL)
        *root = nodeToInsert;
    else if (oneDelayed->key < nodeToInsert->key)
        oneDelayed->rChild = nodeToInsert;
    else
        oneDelayed->lChild = nodeToInsert;
}

void deleteTree(struct TreeNode ** root) {
    /* Post order tree walk */
    if(*root == NULL) return;
```

```
    deleteTree(&(*root)->lChild);
    deleteTree(&(*root)->rChild);
    free(*root);
    *root=NULL;
}

void main() {
    struct TreeNode *root1;
    root1 = NULL;
    struct TreeNode *root2;
    root2 = NULL;

    insert(4, &root1);
    insert(2, &root1);
    insert(3, &root1);
    insert(8, &root1);
    insert(6, &root1);
    insert(7, &root1);
    insert(9, &root1);
    insert(12, &root1);
    insert(1, &root1);

    insert(3, &root2);
    insert(8, &root2);
    insert(10, &root2);
    insert(1, &root2);
    insert(7, &root2);

    deleteTree(&root1);
    deleteTree(&root2);
}

// Linux, Mac: gcc task3.c -o task3; ./task3
// Windows: gcc task3.c -o task3; task3
```

Task 4. Printing the tree

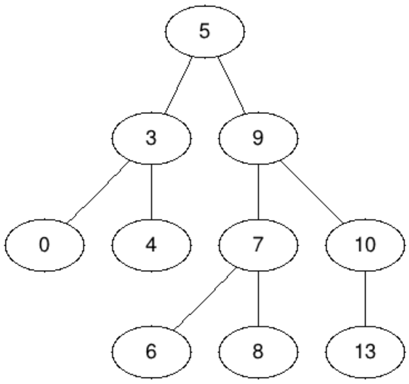
Output:

```
graph g {
  5 -- 3
  3 -- 0
  3 -- 4
  5 -- 9
  9 -- 7
  7 -- 6
  7 -- 8
  9 -- 10
  10 -- 13
}

0 3 4 5 6 7 8 9 10 13

graph g { }
```

Graph:




```c
#include <stdlib.h>
#include <stdio.h>

// ... include code from Task 3 ...

void deleteTree(struct TreeNode ** root) {
    /* Post order tree walk */
    if (*root == NULL)
        return;
    deleteTree(&(*root)->lChild);
    deleteTree(&(*root)->rChild);
    free(*root);
    *root=NULL;
}

void printTreeRecursive(struct TreeNode *root) {
    if (root == NULL)
        return;
    if (root->lChild != NULL) {
        printf(" %d -- %d\n", root->key, root->lChild->key);
        printTreeRecursive(root->lChild);
    }
    if (root->rChild != NULL) {
        printf(" %d -- %d\n", root->key, root->rChild->key);
        printTreeRecursive(root->rChild);
    }
}

void printTree(struct TreeNode *root) {
    printf("graph g {\n");
    printTreeRecursive(root);
```

```
    printf("}\n");
}

void printInOrderRecursive(struct TreeNode *root) {
    if (root == NULL)
        return;
    printInOrderRecursive(root->lChild);
    printf("%d ", root->key);
    printInOrderRecursive(root->rChild);
}

void printInOrder(struct TreeNode *root) {
    printInOrderRecursive(root);
    printf("\n");
}

void main() {
    struct TreeNode *root;
    root = NULL;
    insert(5, &root);
    insert(3, &root);
    insert(4, &root);
    insert(9, &root);
    insert(7, &root);
    insert(8, &root);
    insert(10, &root);
    insert(13, &root);
    insert(0, &root);
    insert(6, &root);

    printTree(root);

    printInOrder(root);

    deleteTree(&root);

    printTree(root);
}

// Linux, Mac: gcc task4.c -o task4; ./task4
// Windows: gcc task4.c -o task4; task4
```

Task 5. Lowest Common Ancestor of 2 nodes

```
#include <stdlib.h>
#include <stdio.h>

// ... include code from Task 3 ...

/* Function to find LCA of n1 and n2. The function assumes that both
  n1 and n2 are present in BST */
struct TreeNode *lca(struct TreeNode *root, int n1, int n2)
{
    if (root == NULL) return NULL;

    // If both n1 and n2 are smaller than root, then LCA lies in left
    if (root->key > n1 && root->key > n2)
        return lca(root->lChild, n1, n2);

    // If both n1 and n2 are greater than root, then LCA lies in right
    if (root->key < n1 && root->key < n2)
        return lca(root->rChild, n1, n2);

    return root;
}

void main() {
    int n1, n2;
    struct TreeNode *t;
    struct TreeNode *root = NULL;

    insert(7, &root);
    insert(5, &root);
    insert(6, &root);
    insert(1, &root);
    insert(9, &root);
    insert(10, &root);
    insert(8, &root);

    printTree(root);

    n1 = 8;
    n2 = 9;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->key);
    n1 = 1;
    n2 = 6;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->key);
```