```
Alexs-MacBook-Air:24 alex$ cat Dayk.h
#ifndef _DAYK_H_
#define _DAYK_H_

#include <string.h>
#include "Lin-2-list-barrier.h"

List* StrToRPN(const char*);
int is_op(char a);
int is_num(char a);
int is_alpha(char a);

#endif
Alexs-MacBook-Air:24 alex$ cat dayk.c
#include "Dayk.h"

int is_num(char a);
int is_alpha(char a);
int is_op(char a);
int is_left_a(char a);
int is_right_a(char a);
int op_prior(char a);

List* StrToRPN(const char* string)
{
      List* out = list_create();
      List* stack = list_create();
      char* str = (char*)malloc(sizeof(char) * (strlen(string) + 1));
    strcpy(str, string);
      char tmp[20];
      char* tmpC;
      char c;
      int num_itr = 0;
      for(int i = 0; i < strlen(str) - 1; i++) {
            if((str[i] == '(' && str[i + 1] == '-') || (str[i] == '^' && str[i + 1] == '-')) {
                  str[i + 1] = '~';
            } else if(str[0] == '-') {
                  str[0] = '~';
            }
      }
      for(int i = 0; i < strlen(str); i++) {
            c = str[i];
            if(is_num(c)) {
                  tmp[num_itr] = c;
                  num_itr++;
                  continue;
            }else if (num_itr != 0) {
                  tmp[num_itr] = '\0';
                  num_itr = 0;
                  list_push_front(out, tmp);
            }
            if(is_alpha(c)){
                  if (i == 0) {
                        tmp[0] = c;
                        tmp[1] = '\0';
                        list_push_front(out, tmp);
                  } else if (is_op(str[i-1]) || str[i-1] == '(') {
                        tmp[0] = c;
                        tmp[1] = '\0';
                        list_push_front(out, tmp);
                  } else {
                        list_destroy(&stack);
                        list_destroy(&out);
                        return NULL;
                  }
            }
            if(c == '(') {
                  tmp[0] = c;
                  tmp[1] = '\0';
                  list_push_front(stack, tmp);
            } else if(c == ')') {
                  while(list_peak(stack, -1)[0] != '(') {
                        if(list_peak(stack, -1) != NULL) {
                              tmpC = list_pop_front(stack);
                              list_push_front(out, tmpC);
                              free(tmpC);
                        } else {
                              list_destroy(&stack);
                              list_destroy(&out);
                              return NULL;
                        }
```

```
                }
                free(list_pop_front(stack));

            } else if(is_op(c)) {
                if(list_peak(stack, -1) != NULL) {
                    while(is_op(list_peak(stack, -1)[0]) &&\
                                list_peak(stack, -1)[0] != c &&\
                                ((is_left_a(c) && op_prior(c) <= op_prior(list_peak(stack, -1)
[0])) ||\
                                (is_right_a(c) && op_prior(c) < op_prior(list_peak(stack, -1)[0])
)))
                    {
                        tmpC = list_pop_front(stack);
                        list_push_front(out, tmpC);
                        free(tmpC);
                        if(list_peak(stack, -1) == NULL) {
                            break;
                        }
                    }
                }
                tmp[0] = c;
                tmp[1] = '\0';
                list_push_front(stack, tmp);
            }
        }
        if(num_itr != 0) {
            tmp[num_itr] = '\0';
            num_itr = 0;
            list_push_front(out, tmp);
        }
        if(list_peak(stack, -1) != NULL) {
            while(list_peak(stack, -1)[0] != '(' && list_peak(stack, -1)[0] != ')') {
                tmpC = list_pop_front(stack);
                list_push_front(out, tmpC);
                free(tmpC);
                if(list_peak(stack, -1) == NULL) {
                    break;
                }
            }
        }
        free(str);
        list_destroy(&stack);
        return out;
}

int is_num(char a)
{
        return ((a - '0') >= 0 && (a - '0') <= 9) ? 1 : 0;
}
int is_alpha(char a)
{
        return (a >= 'a' && a <= 'z') ? 1 : 0;
}
int is_op(char a)
{
        return a == '+' || a == '-' || a == '*' || a == '/' || a == '^' || a == '~' ? 1 : 0;
}
int is_left_a(char a)
{
        return a == '+' || a == '-' || a == '*' || a == '/' ? 1 : 0;
}
int is_right_a(char a)
{
        return a == '^' ? 1 : 0;
}
int op_prior(char a)
{
        if(a == '+' || a == '-') {
                return 1;
        } else if(a == '*' || a == '/') {
                return 2;
        } else if(a == '^') {
                return 3;
        } else if(a == '~') {
                return 999;
        }
        return 0;
}
Alexs-MacBook-Air:24 alex$ cat main.c
#include <stdio.h>
#include <stdlib.h>
```

```
#include "Dayk.h"
#include "Stack.h"
#include "Bitree.h"
#include "Lin-2-list-barrier.h"

void itoa(int n, char* s);
void reverse(char* s);
void reduceCh(BTNode** polynomial,char ch);
void reduceNum(BTNode** polynomial);
BTNode** findNumInFactorsNE(BTNode** tree, BTNode* eq);

BTNode* pop1addend(BTNode** tree)
{
    if(*tree != NULL) {
        BTNode* tmpTreeR;
        BTNode* tmpTreeL;
        char* tmpStr = getTreeValue(*tree);
        if(tmpStr[0] == '+') {
            tmpTreeL = getLeftSon(*tree);
            tmpStr = getTreeValue(tmpTreeL);
            if(tmpStr[0] != '+' && tmpStr[0] != '-') {
                tmpTreeR = getRightSon(*tree);
                free(getTreeValue(*tree));
                free(*tree);
                *tree = tmpTreeR;
                return tmpTreeL;
            } else {
                  tmpTreeR = getRightSon(*tree);
                  tmpStr = getTreeValue(tmpTreeR);
                  if(tmpStr[0] != '+' && tmpStr[0] != '-') {
                  tmpTreeL = getLeftSon(*tree);
                  free(getTreeValue(*tree));
                  free(*tree);
                  *tree = tmpTreeL;
                  return tmpTreeR;
                  } else {
                        tmpTreeR = pop1addend(&((*tree)->right));
                        if(tmpTreeR) {
                              return tmpTreeR;
                        }
                        tmpTreeL = pop1addend(&((*tree)->left));
                        if(tmpTreeL) {
                              return tmpTreeL;
                        }
                  }
            }
        } else if(tmpStr[0] == '-') {
            tmpTreeR = getRightSon(*tree);
            tmpStr = getTreeValue(tmpTreeR);
            if(tmpStr[0] != '+' && tmpStr[0] != '-') {
                tmpTreeL = getLeftSon(*tree);
                addRightTree(*tree, NULL);
                addLeftTree(*tree, NULL);
                treeDestroy(tree);
                *tree = tmpTreeL;
                tmpTreeL = createTree("~");
                addRightTree(tmpTreeL, tmpTreeR);
                return tmpTreeL;
            } else {
                tmpTreeR = pop1addend(&((*tree)->right));
                if(tmpTreeR) {
                    return tmpTreeR;
                }
                tmpTreeL = pop1addend(&((*tree)->left));
                if(tmpTreeL) {
                    return tmpTreeL;
                }
            }
        } else {
            BTNode* tmpTree = *tree;
            *tree = NULL;
            return tmpTree;
        }
    } else {
        return NULL;
    }
    return NULL;
}

void CutOne(BTNode** tree) {
    char* tmpStr = getTreeValue(*tree);
```

```
    BTNode* tmpT;
    if(tmpStr[0] == '*') {
        tmpStr = getTreeValue(getRightSon(*tree));
        if(strcmp(tmpStr, "1") == 0) {
            tmpT = *tree;
            *tree = getLeftSon(*tree);
            addLeftTree(tmpT, NULL);
            treeDestroy(&tmpT);
        } else {
            CutOne(&((*tree)->right));
        }
        tmpStr = getTreeValue(getLeftSon(*tree));
        if(strcmp(tmpStr, "1") == 0) {
            tmpT = *tree;
            *tree = getRightSon(*tree);
            addRightTree(tmpT, NULL);
            treeDestroy(&tmpT);
        } else {
            CutOne(&((*tree)->left));
        }
    } else if(tmpStr[0] == '~') {
        CutOne(&((*tree)->right));
    }
}

int GetPowCh(BTNode* tree, char ch)
{
    char* tmpStr = getTreeValue(tree);
    if(tmpStr[0] == '*') {
        int out;
        out = GetPowCh(tree->left, ch);
        if(out) {
            return out;
        }
        out = GetPowCh(tree->right, ch);
        if(out) {
            return out;
        }
    } else if(tmpStr[0] == ch) {
        return 1;
    } else if(tmpStr[0] == '~') {
        tmpStr = getTreeValue(getRightSon(tree));
        if(tmpStr[0] == ch)
            return 1;
        int out = GetPowCh(getRightSon(tree), ch);
        if(out) {
            return out;
        }
    } else if(tmpStr[0] == '^') {
        tmpStr = getTreeValue(getLeftSon(tree));
        if(tmpStr[0] == ch) {
            tmpStr = getTreeValue(getRightSon(tree));
            if(tmpStr[0] == '~')
                return -atoi(getTreeValue(getRightSon(getRightSon(tree))));
            return atoi(getTreeValue(getRightSon(tree)));
        }
        if(tmpStr[0] == ch) {
            tmpStr = getTreeValue(getLeftSon(getRightSon(tree)));
            if(tmpStr[0] == ch) {
                tmpStr = getTreeValue(getRightSon(tree));
                if(tmpStr[0] == '~')
                    return -atoi(getTreeValue(getRightSon(getRightSon(tree))));
                return atoi(getTreeValue(getRightSon(tree)));
            }
        }
    } else  {
        return 0;
    }
    return 0;
}

BTNode* addPolynomials(BTNode** polynomial1, BTNode** polynomial2)
{
    int min = 1;
    int min1 = 1;
    int min2 = 1;
    int num1;
    int num2;
    int k = 1;
    int is_same;
    char* tmpStr;
```

```
BTNode* out = NULL;
BTNode* pop1;
BTNode* pop2;
BTNode** tmpPop1;
BTNode** tmpPop2;
Stack* pol11 = stack_create();
Stack* pol12 = stack_create();
Stack* pol21 = stack_create();
Stack* pol22 = stack_create();
BTNode* tmpT;
pop1 = pop1addend(polynomial1);

while(pop1) {
    stack_push(pol11, pop1);
    pop1 = pop1addend(polynomial1);
}
pop2 = pop1addend(polynomial2);
while(pop2) {
    stack_push(pol21, pop2);
    pop2 = pop1addend(polynomial2);
}
pop1 = stack_pop(pol11);
while(pop1) {
    if(k == 1)
        pop2 = stack_pop(pol21);
    if(k == -1)
        pop2 = stack_pop(pol22);
    while(pop2) {
        is_same = 1;
        for (int i = 97; i <= 122; i++) {
            if(GetPowCh(pop1, i) != GetPowCh(pop2, i)) {
                is_same = 0;
                break;
            }
        }
        if(is_same) {
            tmpStr = getTreeValue(pop1);
            if(tmpStr[0] == '~') {
                min1 *= -1;
                tmpT = getRightSon(pop1);
                addRightTree(pop1, NULL);
                treeDestroy(&pop1);
                pop1 = tmpT;
            }

            tmpStr = getTreeValue(pop2);
            if(tmpStr[0] == '~') {
                min2 *= -1;
                tmpT = getRightSon(pop2);
                addRightTree(pop2, NULL);
                treeDestroy(&pop2);
                pop2 = tmpT;
            }
            tmpPop1 = findNumInFactorsNE(&pop1, NULL);
            if(tmpPop1 == NULL) {
                tmpT = createTree("*");
                addRightTree(tmpT, pop1);
                addLeftTree(tmpT, createTree("1"));
                pop1 = tmpT;
            }
            tmpPop1 = findNumInFactorsNE(&pop1, NULL);
            tmpStr = getTreeValue(*tmpPop1);
            if(tmpStr[0] == '~') {
                min1 *= -1;
                tmpT = getRightSon(*tmpPop1);
                addRightTree(*tmpPop1, NULL);
                treeDestroy(tmpPop1);
                *tmpPop1 = tmpT;
            }

            tmpPop2 = findNumInFactorsNE(&pop2, NULL);
            if(tmpPop2 == NULL) {
                tmpT = createTree("*");
                addRightTree(tmpT, pop2);
                addLeftTree(tmpT, createTree("1"));
                pop2 = tmpT;
            }
            tmpPop2 = findNumInFactorsNE(&pop2, NULL);
            tmpT = *(tmpPop2);
            *tmpPop2 = NULL;
            treeDestroy(&pop2);
```

```
            pop2 = tmpT;

            if(tmpStr[0] == '~') {
                min2 *= -1;
                tmpT = getRightSon(pop2);
                addRightTree(pop2, NULL);
                treeDestroy(&pop2);
                pop2 = tmpT;
            }
            num1 = atoi(getTreeValue(*tmpPop1));
            num2 = atoi(getTreeValue(pop2));

            char p[17];
            int tmpRes = num1 * min1 + num2 * min2;
            if(tmpRes < 0) {
                tmpRes = -tmpRes;
                min = -1;
            }
            itoa(tmpRes, p);
            (*tmpPop1)->data = (char*)realloc((*tmpPop1)->data, sizeof(char) * (strlen(p) + 1));
            strcpy(getTreeValue(*tmpPop1), p);

            if(min == -1) {
                tmpT = createTree("~");
                addRightTree(tmpT, pop1);
                pop1 = tmpT;
            }
            min1 = 1;
            min2 = 1;
            min = 1;

        } else {
            if(k == 1)
                stack_push(pol22, pop2);
            if(k == -1)
                stack_push(pol21, pop2);
        }
        if(k == 1)
            pop2 = stack_pop(pol21);
        if(k == -1)
            pop2 = stack_pop(pol22);
    }
    k *= -1;
    if(findNumInFactorsNE(&pop1, NULL)) {
        tmpStr = getTreeValue(*findNumInFactorsNE(&pop1, NULL));
        if(tmpStr[0] == '0') {
            treeDestroy(&pop1);
        } else if(tmpStr[0] == '1') {
            CutOne(&pop1);
            stack_push(pol12, pop1);
        } else {
            stack_push(pol12, pop1);
        }
    } else {
        stack_push(pol12, pop1);
    }
    pop1 = stack_pop(pol11);
}
out = stack_pop(pol12);
if(out == NULL) {
    if(k == 1)
        out = stack_pop(pol21);
    if(k == -1)
        out = stack_pop(pol22);
}
pop2 = stack_pop(pol12);
if(pop2 == NULL) {
    if(k == 1)
        pop2 = stack_pop(pol21);
    if(k == -1)
        pop2 = stack_pop(pol22);
}
while(pop2) {
    tmpStr = getTreeValue(pop2);
    if(tmpStr[0] == '~') {
        tmpT = createTree("-");
        addLeftTree(tmpT, out);
        addRightTree(tmpT, getRightSon(pop2));
        addRightTree(pop2, NULL);
        treeDestroy(&pop2);
    } else {
```

```
            tmpT = createTree("+");
            addLeftTree(tmpT, out);
            addRightTree(tmpT, pop2);
        }
        out = tmpT;
        pop2 = stack_pop(pol12);
        if(pop2 == NULL) {
            if(k == 1)
                pop2 = stack_pop(pol21);
            if(k == -1)
                pop2 = stack_pop(pol22);
        }
    }
    stack_delete (&pol11);
    stack_delete (&pol12);
    stack_delete (&pol21);
    stack_delete (&pol22);
    return out;
}

void reverse(char* s)
{
    int i, j;
    char c;

    for (i = 0, j = strlen(s)-1; i<j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}

void itoa(int n, char* s)
{
    int i, sign;

    if ((sign = n) < 0)
        n = -n;
    i = 0;
    do {
        s[i++] = n % 10 + '0';
    } while ((n /= 10) > 0);
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}

int extractNode(BTNode** tree, BTNode* eq)
{
    if(*tree != NULL) {
        if(*tree == eq) {
            treeDestroy(tree);
            return -1;
        } else {
            if (extractNode(&((*tree)->left), eq) == -1) {
                BTNode* tmpR = (*tree)->right;
                free((*tree)->data);
                free(*tree);
                *tree = tmpR;
            } else if (extractNode(&((*tree)->right), eq) == -1) {
                BTNode* tmpL = (*tree)->left;
                free((*tree)->data);
                free(*tree);
                *tree = tmpL;
            } else if (extractNode(&((*tree)->right), eq) == 0 || extractNode(&((*tree)->left), eq)
== 0 ){
                return 0;
            } else {
                return 1;
            }
        }
    }
    return 1;
}

BTNode** findChPowInFactorsNE(BTNode** tree,BTNode* eq, char ch)
{
    char* tmpStr = getTreeValue(*tree);
    if(tmpStr[0] == '*') {
        BTNode** tmpTree;
```

```
        tmpTree = findChPowInFactorsNE(&((*tree)->left), eq, ch);
        if(tmpTree) {
            return tmpTree;
        }
        tmpTree =findChPowInFactorsNE(&((*tree)->right), eq, ch);
        if(tmpTree) {
            return tmpTree;
        }
    } else if(tmpStr[0] == ch && *tree != eq) {
        return tree;
    } else if(tmpStr[0] == '~') {
        BTNode* tmpTree;
        tmpTree = getRightSon(*tree);
        tmpStr = getTreeValue(tmpTree);
        if(tmpStr[0] == ch && tmpTree != eq)
            return tree;
    } else if(tmpStr[0] == '^' && *tree != eq) {
        BTNode* tmpTree;
        tmpTree = getLeftSon(*tree);
        tmpStr = getTreeValue(tmpTree);
        if(tmpStr[0] == ch) {
            return tree;
        }
        if(tmpStr[0] == '~') {
            tmpTree = getRightSon(tmpTree);
            tmpStr = getTreeValue(tmpTree);
            if(tmpStr[0] == ch)
                return tree;
        }
    } else  {
        return NULL;
    }
    return NULL;
}

int reduceOne(BTNode** tree)
{
    BTNode* tmpT;
    char* tmpStr = getTreeValue(*tree);
    if(tmpStr[0] == '*') {
        tmpStr = getTreeValue(getRightSon(*tree));
        if(!strcmp(tmpStr, "1")) {
            tmpT = *tree;
            *tree = getLeftSon(*tree);
            addLeftTree(tmpT, NULL);
            treeDestroy(&tmpT);
            return 0;
        } else
            reduceOne(&((*tree)->right));
        tmpStr = getTreeValue(getLeftSon(*tree));
        if(!strcmp(tmpStr, "1")) {
            tmpT = *tree;
            *tree = getRightSon(*tree);
            addRightTree(tmpT, NULL);
            treeDestroy(&tmpT);
            return 0;
        } else
            reduceOne(&((*tree)->left));
    }
    return 1;
}

void reduceCh(BTNode** polynomial,char ch)
{
    int min = 1;
    int minx = 1;
    int min1 = 1;
    int min2 = 1;
    char* tmpStr;
    BTNode** ch1T = NULL;
    BTNode** ch2T = NULL;
    ch1T = findChPowInFactorsNE(polynomial, NULL, ch);
    if(ch1T != NULL) {
        ch2T = findChPowInFactorsNE(polynomial, *ch1T, ch);
    }
    while(ch2T) {
        tmpStr = getTreeValue(*ch1T);
        if(tmpStr[0] != '^') {

            BTNode* powTmp = createTree("^");
            BTNode* powNTmp = createTree("1");
```

```
            addRightTree(powTmp, powNTmp);
            addLeftTree(powTmp, *ch1T);
            *ch1T = powTmp;
        }
        tmpStr = getTreeValue(*ch2T);
        if(tmpStr[0] != '^'){

            BTNode* powTmp = createTree("^");
            BTNode* powNTmp = createTree("1");
            addRightTree(powTmp, powNTmp);
            addLeftTree(powTmp, *ch2T);
            *ch2T = powTmp;
        }
        tmpStr = getTreeValue(getLeftSon(*ch1T));
        if(tmpStr[0] == '~') {
            minx *= -1;
        }

           BTNode* tmpT1;
        BTNode* tmpT2;
        tmpStr = getTreeValue(getRightSon(*ch1T));
        if(tmpStr[0] == '~') {
            min1 *= -1;
            tmpT1 = getRightSon(getRightSon(*ch1T));
        } else {
            tmpT1 = getRightSon(*ch1T);
        }

        tmpStr = getTreeValue(getRightSon(*ch2T));
        if(tmpStr[0] == '~') {
            min2 *= -1;
            tmpT2 = getRightSon(getRightSon(*ch2T));
        } else {
            tmpT2 = getRightSon(*ch2T);
        }


        int num1 = atoi(getTreeValue(tmpT1));
        int num2 = atoi(getTreeValue(tmpT2));

        char p[17];
        int tmpRes = num1 * min1 + num2 * min2;
        if(tmpRes < 0) {
            tmpRes = -tmpRes;
            min = -1;
        }
        itoa(tmpRes, p);
        treeDestroy(&(*ch1T)->right);
        if(min == -1) {
            tmpT1 = createTree("~");
            addRightTree(tmpT1, createTree(p));
            addRightTree(*ch1T, tmpT1);
        } else {
            addRightTree(*ch1T, createTree(p));
        }
        extractNode(polynomial, *ch2T);

        ch1T = findChPowInFactorsNE(polynomial, NULL, ch);
      if(ch1T != NULL) {
            ch2T = findChPowInFactorsNE(polynomial, *ch1T, ch);
      }
        min1 = 1;
        min2 = 1;
        min = 1;
    }
    if(minx == -1){
        tmpStr = getTreeValue(getRightSon(*ch2T));
        BTNode* tmp = getRightSon(*ch2T);
        if(tmpStr[0] == '~') {
            tmp = getRightSon(*ch2T);
            addRightTree(*ch2T, getRightSon(getRightSon(*ch2T)));
            addRightTree(tmp, NULL);
            treeDestroy(&tmp);
        } else {
            tmp = createTree("~");
            addRightTree(tmp, getRightSon(*ch2T));
            addLeftTree(*ch2T, tmp);
        }
    }

}
```

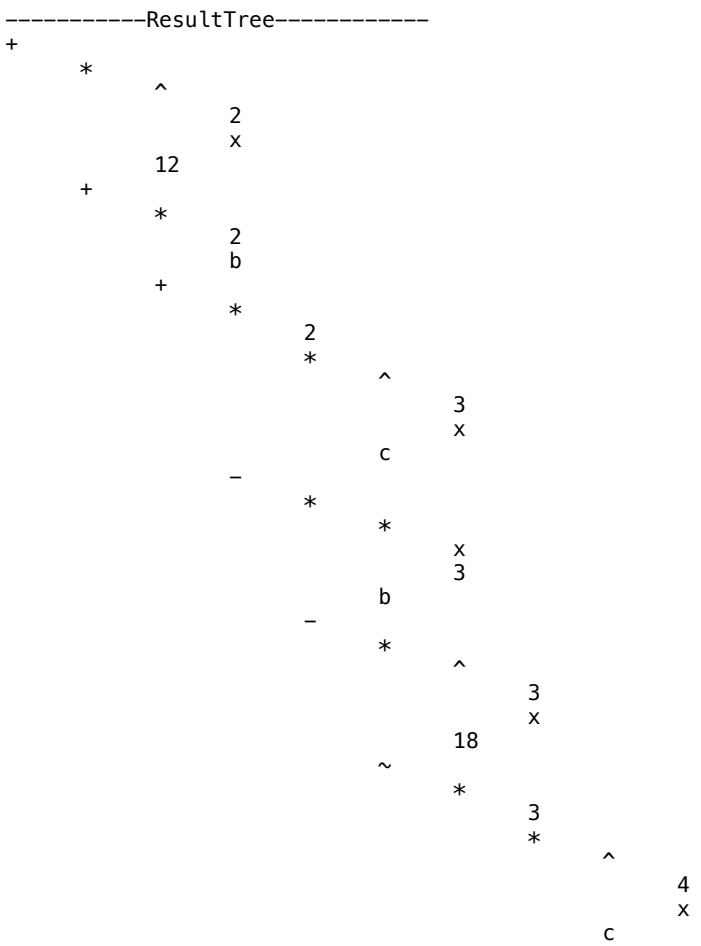
```
BTNode** findNumInFactorsNE(BTNode** tree, BTNode* eq)
{
    char* tmpStr = getTreeValue(*tree);
    if(tmpStr[0] == '*') {
        BTNode** tmpTree;
        tmpTree = findNumInFactorsNE(&((*tree)->left), eq);
        if(tmpTree) {
            return tmpTree;
        }
        tmpTree =findNumInFactorsNE(&((*tree)->right), eq);
        if(tmpTree) {
            return tmpTree;
        }
    } else if(is_num(tmpStr[0]) && *tree != eq) {
        return tree;
    } else if(tmpStr[0] == '~') {
        BTNode* tmpTree;
        tmpTree = getRightSon(*tree);
        tmpStr = getTreeValue(tmpTree);

        if(is_num(tmpStr[0]) && *tree != eq) {
            return tree;
        } else if(is_num(tmpStr[0])) {
           return NULL;
        }
        BTNode** tmpTree2;
        tmpTree2 = findNumInFactorsNE(&((*tree)->right), eq);
        if(tmpTree2) {
            return tmpTree2;
        }
    } else {
        return NULL;
    }
    return NULL;

}

void reduceMin(BTNode** addend)
{
      char* tmpStr;
      BTNode* tmpT;
      tmpStr = getTreeValue(*addend);
      if(tmpStr[0] == '~') {;
            tmpStr = getTreeValue(getRightSon(*addend));
            if(tmpStr[0] == '+') {
                  tmpT= getRightSon(*addend);
                  addRightTree(*addend, NULL);
                  treeDestroy(addend);
                  *addend = createTree("-");
                  addRightTree(*addend, getRightSon(tmpT));
                  addLeftTree(*addend, createTree("~"));
                  addRightTree(getLeftSon(*addend), getLeftSon(tmpT));
                  addLeftTree(tmpT, NULL);
                  addRightTree(tmpT, NULL);
                  reduceMin(&((*addend)->left));
            }
      }
}

void reduceNum(BTNode** polynomial)
{
    int min = 1;
    char* tmpStr;
    int num1 = 0;
    int num2 = 0;
    BTNode** num1T = NULL;
    BTNode** num2T = NULL;
    BTNode* num1add;
    BTNode* num2add;
    num1T = findNumInFactorsNE(polynomial, NULL);

    if(num1T != NULL) {
        num2T = findNumInFactorsNE(polynomial, *num1T);
    }
    while(num2T) {
        tmpStr = getTreeValue(*num1T);
        if(tmpStr[0] == '~') {
            min *= -1;
            num1add = getRightSon(*num1T);
        } else {
```

```
            num1add = *num1T;
        }

        tmpStr = getTreeValue(*num2T);
        if(tmpStr[0] == '~') {
            min *= -1;
            num2add = getRightSon(*num2T);
        } else {
            num2add = *num2T;
        }
        num1 = atoi(getTreeValue(num1add));
        num2 = atoi(getTreeValue(num2add));

        char p[17];
        itoa(num1*num2, p);
        treeDestroy(num1T);
        if(min == -1) {
            *num1T = createTree("~");
            addRightTree(*num1T, createTree(p));

        } else {
            *num1T = createTree(p);
        }
        extractNode(polynomial, *num2T);

        num1T = findNumInFactorsNE(polynomial, NULL);
      if(num1T != NULL) {
            num2T = findNumInFactorsNE(polynomial, *num1T);
      }
      min = 1;
    }
}

void multiplyBranchToAddend(BTNode** polynomial, BTNode* addend)
{
    int min = 1;
    char* tmpStr;
    BTNode* tmpT;
    BTNode* out;
    Stack* pol1 = stack_create();
    Stack* pol2 = stack_create();
    BTNode* pop1 = pop1addend(polynomial);
    while(pop1) {
      reduceMin(&pop1);
        stack_push(pol1, pop1);
        pop1 = pop1addend(polynomial);
    }
    pop1 = stack_pop(pol1);
    tmpStr = getTreeValue(pop1);
    if(tmpStr[0] == '~') {
        min *= -1;
        tmpT = getRightSon(pop1);
        addRightTree(pop1, NULL);
        treeDestroy(&pop1);
        pop1 = tmpT;
    }
    while(pop1) {
        out = createTree("*");
        tmpStr = getTreeValue(addend);
        BTNode* numAd = NULL;
        if(tmpStr[0] == '~') {
            min *= -1;
            copyTree(&numAd, getRightSon(addend));
        } else {
            copyTree(&numAd, addend);
        }
        addRightTree(out, numAd);
        addLeftTree(out, pop1);
        pop1 = out;
        reduceNum(&pop1);
        reduceOne(&pop1);
        for(int i = 'a'; i <= 'z'; i++)
            reduceCh(&pop1, i);
        if(min == -1) {
            out = createTree("~");
            addRightTree(out, pop1);
            pop1 = out;
        }
        min = 1;
        stack_push(pol2, pop1);
        pop1 = stack_pop(pol1);
```

```
        if(pop1) {
            tmpStr = getTreeValue(pop1);
        }
        if(tmpStr[0] == '~' && pop1) {
            min *= -1;
            tmpT = getRightSon(pop1);
            addRightTree(pop1, NULL);
            treeDestroy(&pop1);
            pop1 = tmpT;
        }
    }
    *polynomial = stack_pop(pol2);
    out = stack_pop(pol2);
    while(out) {
        tmpStr = getTreeValue(out);
        if(tmpStr[0] == '~') {
            tmpT = createTree("-");
            addLeftTree(tmpT, *polynomial);
            addRightTree(tmpT, getRightSon(out));
            addRightTree(out, NULL);
            treeDestroy(&out);
        } else {
            tmpT = createTree("+");
            addLeftTree(tmpT, *polynomial);
            addRightTree(tmpT, out);
        }
        *polynomial = tmpT;
        out = stack_pop(pol2);
    }
    stack_delete (&pol1);
    stack_delete (&pol2);
}

BTNode* multiplyPolynomialToAddend(BTNode* polynomial, BTNode* addend)
{
    BTNode* out = NULL;
    copyTree(&out, polynomial);
    multiplyBranchToAddend(&out, addend);
    return out;
}

BTNode* multiplyPolynomials(BTNode* polynomial1, BTNode* polynomial2)
{
    BTNode* addendsTree = NULL;
    BTNode** tmpT =(BTNode**)malloc(sizeof(BTNode**));
    BTNode* out;
    copyTree(&addendsTree, polynomial2);
    BTNode* addend  = pop1addend(&addendsTree);
    reduceMin(&addend);
    out = multiplyPolynomialToAddend(polynomial1, addend);
    treeDestroy(&addend);
    addend  = pop1addend(&addendsTree);
    while(addend != NULL) {
      reduceMin(&addend);
        *tmpT = multiplyPolynomialToAddend(polynomial1, addend);
        out =  addPolynomials(&out, tmpT);
        treeDestroy(&addend);
        addend  = pop1addend(&addendsTree);
    }
    free(tmpT);
    return out;
}

BTNode* RPNtoTree(List* list)
{
    if (list != NULL) {
        Stack *stack = stack_create();
        BTNode* node;
        char* tmp;
        while(list_peak(list, 1) != NULL) {
            tmp = list_pop_back(list);
            if(is_op(tmp[0]) && tmp[0] != '~') {
                node = createTree(tmp);
                addRightTree(node,stack_pop(stack));
                addLeftTree(node,stack_pop(stack));
                stack_push(stack, node);
            } else if(tmp[0] == '~') {
                node = createTree(tmp);
                addRightTree(node,stack_pop(stack));
                stack_push(stack, node);
            } else {
```

```
                stack_push(stack, createTree(tmp));
            }
            free(tmp);
        }
        node = stack_pop(stack);
        stack_delete(&stack);
        return node;
    } else {
        return NULL;
    }
}

char* treeToStr(BTNode* tree, char* out) {
      char* tmpStr;
      tmpStr = getTreeValue(tree);
      if (tmpStr[0] == '*') {
            tmpStr = getTreeValue(getLeftSon(tree));
            if (tmpStr[0] == '-' || tmpStr[0] == '+' || tmpStr[0] == '~') {
                  strcat(out, "(");
                  treeToStr(getLeftSon(tree), out);
                  strcat(out, ")");
            } else {
                  treeToStr(getLeftSon(tree), out);
            }
            strcat(out, "*");
            tmpStr = getTreeValue(getRightSon(tree));
            if (tmpStr[0] == '-' || tmpStr[0] == '+' || tmpStr[0] == '~') {
                  strcat(out, "(");
                  treeToStr(getRightSon(tree), out);
                  strcat(out, ")");
            } else {
                  treeToStr(getRightSon(tree), out);
            }
      } else if (tmpStr[0] == '+' || tmpStr[0] == '-') {
            treeToStr(getLeftSon(tree), out);
            strcat(out, getTreeValue(tree));
            treeToStr(getRightSon(tree), out);
      } else if (tmpStr[0] == '/' || tmpStr[0] == '^') {
            tmpStr = getTreeValue(getLeftSon(tree));
            if (!((is_num(tmpStr[0]) || is_alpha(tmpStr[0])) || tmpStr[0] == '~')) {
                  strcat(out, "(");
                  treeToStr(getLeftSon(tree), out);
                  strcat(out, ")");
            } else {
                  treeToStr(getLeftSon(tree), out);
            }
            strcat(out, getTreeValue(tree));
            tmpStr = getTreeValue(getRightSon(tree));
            if (!((is_num(tmpStr[0]) || is_alpha(tmpStr[0])) || tmpStr[0] == '~')) {
                  strcat(out, "(");
                  treeToStr(getRightSon(tree), out);
                  strcat(out, ")");
            } else {
                  treeToStr(getRightSon(tree), out);
            }
      } else if (tmpStr[0] == '~') {
            tmpStr = getTreeValue(getRightSon(tree));
            //if (tmpStr[0] == '*') {
                  //strcat(out, "-");
            //} else {
                  strcat(out, "-");
                  strcat(out, "(");
                  treeToStr(getRightSon(tree), out);
                  strcat(out, ")");
            //}
      } else if (is_num(tmpStr[0]) || is_alpha(tmpStr[0])) {
            strcat(out, tmpStr);
      }
      return out;
}

int main (void)
{

    char e1[1024];
    char e2[1024];
    scanf("%s %s", e1, e2);
    char out[1024];
    out[0] = '\0';
    List* e1L = StrToRPN(e1);
    List* e2L = StrToRPN(e2);
```

```
    puts("----RevercePolishNotationOfE1----");
    list_print(e1L);
    puts("----RevercePolishNotationOfE2----");
    list_print(e2L);
    BTNode* e1T = RPNtoTree(e1L);
    BTNode* e2T = RPNtoTree(e2L);
    BTNode* outT;
    puts("------------TreeOfE1-------------");
    printTree(e1T, 0);
    puts("------------TreeOfE2-------------");
    printTree(e2T, 0);
    outT = multiplyPolynomials(e1T, e2T);
    puts("-----------ResultTree------------");
    printTree(outT, 0);
    puts("-------ResultExpression----------");
    treeToStr(outT, out);
    printf("%s\n", out);
    puts("--------------------------------");
    list_destroy(&e1L);
    treeDestroy(&e1T);
    list_destroy(&e2L);
    treeDestroy(&e2T);
    treeDestroy(&outT);
    return 0;
}

Alexs-MacBook-Air:24 alex$ cat makefile
CC = gcc
LD = gcc
CCFLAGS = -Wall -pedantic  -std=c99
LDFLAGS =

main.out: main.o lin-2-list-barrier.o dayk.o bitree.o stack.o
      $(LD) $(LDFLAGS) -o main.out main.o lin-2-list-barrier.o dayk.o stack.o bitree.o
main.o: main.c bitree.c Bitree.h lin-2-list-barrier.c Lin-2-list-barrier.h stack.c Stack.h Dayk.h
dayk.c
      $(CC) $(CCFLAGS) -c main.c
dayk.o: Dayk.h dayk.c lin-2-list-barrier.c Lin-2-list-barrier.h
      $(CC) $(CCFLAGS) -c dayk.c
stack.o: stack.c Stack.h Bitree.h
      $(CC) $(CCFLAGS) -c stack.c
bitree.o: bitree.c Bitree.h
      $(CC) $(CCFLAGS) -c bitree.c
lin-2-list-barrier.o: lin-2-list-barrier.c Lin-2-list-barrier.h
      $(CC) $(CCFLAGS) -c lin-2-list-barrier.c
clean:
      rm *.o
Alexs-MacBook-Air:24 alex$ make
gcc -Wall -pedantic  -std=c99 -c main.c
gcc -Wall -pedantic  -std=c99 -c lin-2-list-barrier.c
gcc -Wall -pedantic  -std=c99 -c dayk.c
gcc -Wall -pedantic  -std=c99 -c bitree.c
gcc -Wall -pedantic  -std=c99 -c stack.c
gcc  -o main.out main.o lin-2-list-barrier.o dayk.o stack.o bitree.o
Alexs-MacBook-Air:24 alex$ ./main.out
b+6*x^2+c*x^3
2-3*x
----RevercePolishNotationOfE1----
b 6 x 2 ^ * c x 3 ^ * + +
----RevercePolishNotationOfE2----
2 3 x * -
------------TreeOfE1-------------
+
      +
            *
                  ^
                        3
                        x
                  c
            *
                  ^
                        2
                        x
                  6
      b
------------TreeOfE2-------------
-
      *
            x
            3
      2
```



Alexs-MacBook-Air:24 alex$