|  |  |
| --- | --- |
|  | **Отчёт по лабораторной работе** №24 по курсу Языки и методы программирования­  студента группы М08-107Б-18 Хренова Геннадия Николаевича , № по списку 24  Адреса www, e-mail, jabber, skype [khrenov.gena@yandex.ru](mailto:khrenov.gena@yandex.ru)  Работа выполнена: “31“ мая 2019г.  Преподаватель: аспирант каф.806 Ридли А. Н.  Входной контроль знаний с оценкой  Отчёт сдан “ “ 2019г., итоговая оценка  Подпись преподавателя |

1. **Тема**: Деревья выражений
2. **Цель работы**: Составить программу выполнения заданных преобразований арифметических выражений с применением деревьев.
3. **Задание** (*вариант №* 36): Подсчитать количество переменных, используемых в данном выражении.
4. **Оборудование** (*лабораторное*):

ЭВМ Pentium , процессор 2.7 ГГц , имя узла сети с ОП 4096 МБ

НМД 2 ГБ. Терминал адрес . Принтер

Другие устройства

*Оборудование ПЭВМ студента, если использовалось:*

Процессор intel core i3 , ОП 3072 , НМД 50 ГБ. Монитор

Другие устройства

1. **Программное обеспечение** (*лабораторное*):

Операционная система семейства Unix , наименование версия

Интерпретатор команд bash версия

Система программирования версия

Редактор текстов Emacs версия

Утилиты операционной системы

Прикладные системы и программы

Местонахождения и имена файлов программ и данных

*Программное обеспечение ЭВМ студента, если использовалось:*

Операционная система семейства Unix , наименование Ubuntu версия 18.04LTS

Интерпретатор команд bash версия

Система программирования версия

Редактор текстов Emacs версия

Утилиты операционной системы

Прикладные системы и программы

Местонахождения и имена файлов программ и данных

1. **Идея, метод, алгоритм** решения задачи (в формах: словесной, псевдокода, графической [блок-схема, диаграмма, рисунок, таблица] или формальное описание с пред- и постусловиями)

Программа состоит из следующих составляющих:

1) **list.h & list.c** - описание структуры список и всех функций, нужных для работы

2) **tree.h & tree.c** — описание структуры дерево и всех функций, нужных для работы

3) **stack.h & stack.c** - описание структуры стек и всех функций, нужных для работы

4) **sortstation.h & sortstation.c** - описание алгоритма сортировочной станции

5) **main.c** — главная часть, вызывающая функции

Алгоритм:

* + - 1. Перевести входное выражение в обратную польскую нотацию. Здесь нужно применить алгоритм Сортировочной станции, основанный на стеке и очереди. Я воспользовался списком с типом данных char \* для удобства, чтобы не реализовывать стек и очередь.
      2. Преобразовать обратную польскую нотацию в дерево. Вместо обычного подсчета выражения на каждом шагу создаем узел дерева и у операции дочерние элементы будут выражениями. В этом пункте необходимо реализовать стек с типом данных узел дерева
      3. Выполнить задание
      4. Привести выражение из дерева в обычную инфиксную нотацию.

**Алгоритм сортировочной станции:**

Пока не все токены обработаны:

* Прочитать [токен](https://ru.wikipedia.org/wiki/Токен_(лексический_анализ)).
* Если токен — *число*, то добавить его в очередь вывода.
* Если токен — [*функция*](https://ru.wikipedia.org/wiki/Функция_(программирование)), то поместить его в стек.
* Если токен — *разделитель аргументов функции* (например запятая):
* Пока токен на вершине стека не *открывающая скобка*, перекладывать операторы из стека в выходную очередь. Если в стеке не было *открывающей скобки*, то в выражении пропущен *разделитель аргументов функции* (запятая), либо пропущена *открывающая скобка*.
* Если токен — *оператор* op1, то:
* Пока присутствует на вершине стека токен *оператор* op2, и

Либо *оператор* op1 [*лево-ассоциативен*](https://ru.wikipedia.org/wiki/Ассоциативность_(программирование)) и его приоритет меньше, чем у *оператора* op2 либо равен,

или *оператор* op1 право-ассоциативен и его приоритет меньше, чем у op2,

переложить op2 из стека в выходную очередь;

* (Иначе, когда стек операторов пуст или содержит открывающую скобку)
  + положить op1 в стек.
* Если токен — *открывающая скобка*, то положить его в стек.
* Если токен — *закрывающая скобка*:
* Пока токен на вершине стека не является *открывающей скобкой*, перекладывать операторы из стека в выходную очередь.
* Выкинуть *открывающую скобку* из стека, но не добавлять в очередь вывода.
* Если токен на вершине стека — функция, добавить её в выходную очередь.
* Если стек закончился до того, как был встречен токен *открывающая скобка*, то в выражении пропущена скобка.
* Если больше не осталось токенов на входе:
* Пока есть токены операторы в стеке:
* Если токен оператор на вершине стека — скобка, то в выражении присутствует незакрытая скобка.
* Переложить оператор из стека в выходную очередь.
* Конец.

1. **Сценарий выполнения работы** [план работы, первоначальный текст программы в черновике (можно на отдельном листе) и тесты, либо соображения по тестированию].

Тесты:

1) a

2) a+1

3) (z\*x+1)/w

4)1+2+4+0

5)1+2+w

6) a\*a\*a\*a\*a

7) (a+b)\* (a+b)

8) (a+a)\*b

9) (a+a) \* 3

10) (a\*b\*c\*s+9)-(a\*q\*w\*e+1)-8

11) присутствует незакрытая скобка

*Пункты 1-7 отчёта составляются* ***строго до*** *начала лабораторной работы.*

Допущен к выполнению работы. Подпись преподавателя

1. **Распечатка протокола** (подклеить листинг окончательного варианта программы с текстовыми примерами, подписанный преподавателем)

|  |  |
| --- | --- |
| **sortstation.h**  #ifndef \_DAYK\_H\_  #define \_DAYK\_H\_  #include <string.h>  #include "list.h"  List StrToRPN(const char\*);  int is\_op(char a);  int is\_num(char a);  int is\_alpha(char a);  int have\_br(List \*l);  #endif  **list.h**  #ifndef List\_h  #define List\_h  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  typedef struct {  struct Item \*head;  int size;  } List;  struct Item {  struct Item \*next;  struct Item \*prev;  char\* data;  };  typedef struct {  struct Item \*node;  } Iterator;  void list\_create(List \*l);  Iterator list\_first(const List \*l);  Iterator list\_last(const List \*l);  bool list\_empty(const List \*l);  int list\_size(const List \*l);  void list\_insert\_front(List \*l,char\* val);  void list\_insert\_after(List \*l, char\* val);  void list\_delete\_last(List \*l);  void list\_delete\_front(List \*l);  void list\_destroy(List \*l);  void list\_print( List \*l);  **tree.h**  #ifndef BINTREE\_H\_  #define BINTREE\_H\_  #include <stdlib.h>  #include <stdio.h>  #include <string.h>  typedef char\* tree\_data\_type;  typedef struct Node {  tree\_data\_type data;  struct Node \*left;  struct Node \*right;  } Node;  typedef struct {  Node \*root;  int size;  } Bintree;  Node\* createTree(tree\_data\_type value);  void copyTree(Node\*\* out, Node\* tree);  void addLeftTree(Node\* tree, Node\* in);  void addRightTree(Node\* tree, Node\* in);  void printTree(Node\* tree, int deep);  void treeDestroy(Node\*\* tree);  Node\* getRightSon(Node\* tree);  Node\* getLeftSon(Node\* tree);  tree\_data\_type getTreeValue(Node\* tree);  #endif  tree.c  #include "tree.h"  #include <string.h>  #include <stdlib.h>  Node\* createTree(tree\_data\_type value)  {  Node\* out = (Node\*)malloc(sizeof(Node));  out->data = (char\*)malloc(sizeof(char) \* (strlen(value) + 1));  strcpy(out->data, value);  out->left = NULL;  out->right = NULL;  return out;  }  void addLeftTree(Node\* tree, Node\* in)  {  tree->left = in;  }  void addRightTree(Node\* tree, Node\* in)  {  tree->right = in;  }  void printTree(Node\* tree, int deep)  {  if (tree != NULL) {  printTree(tree->right, deep + 5);  printf("%\*s\n", deep, tree->data);  printTree(tree->left, deep + 5);  }  }  void treeDestroy(Node\*\* tree)  {  if (\*tree != NULL) {  free((\*tree)->data);  treeDestroy(&((\*tree)->right));  treeDestroy(&((\*tree)->left));  free(\*tree);  \*tree = NULL;  }  }  Node\* getRightSon(Node\* tree)  {  return tree->right;  }  Node\* getLeftSon(Node\* tree)  {  return tree->left;  }  tree\_data\_type getTreeValue(Node\* tree)  {  return tree->data;  }  **list.c**  #include "list.h"  #include <string.h>  #include <stdio.h>  void list\_create(List \*l)  {  l->head = malloc(sizeof(struct Item));    l->head->prev = l->head->next = l->head;  l->size = 0;  }  Iterator list\_first(const List \*l)  {  Iterator res = {l->head};    return res;  }  Iterator list\_last(const List \*l)  {  Iterator res = {l->head->prev};    return res;  }  int list\_size(const List \*l)  {  return l->size;  }  void list\_insert\_front(List \*l, char\* val)  {  if (l->size == 0){  Iterator i = list\_first(l);    i.node->data = (char\*)malloc(sizeof(char) \* (strlen(val) + 1));  strcpy(i.node->data, val);  l->size++;  return;  }  Iterator res = {malloc(sizeof(struct Item))};    Iterator it = list\_first(l);  l->head = res.node;  res.node->data = (char\*)malloc(sizeof(char) \* strlen(val) + 1);  strcpy(res.node->data, val);  res.node->next = it.node;  res.node->prev = it.node->prev;  it.node->prev->next = res.node;  it.node->prev = res.node;  l->size++;  return ;  }  void list\_insert\_after(List \*l, char\* val)  {  if (l->size == 0){  Iterator i = list\_first(l);    i.node->data = (char\*)malloc(sizeof(char) \* strlen(val) + 1);  strcpy(i.node->data, val);  l->size++;  return;    }  Iterator res = {malloc(sizeof(struct Item))};  if (!res.node)  return;  Iterator i = list\_last(l);  res.node->data = (char\*)malloc(sizeof(char) \* strlen(val) + 1);  strcpy(res.node->data, val);  res.node->prev = i.node;  res.node->next = i.node->next;  i.node->next->prev = res.node;  i.node->next = res.node;  l->size++;    }  void list\_delete\_last(List \*l)  {  Iterator res = list\_first(l);  Iterator i = list\_last(l);  if (list\_size == 0){  return;  }else if (Equal(&i, &res)) {  if(list\_size(l) == 1){  i.node = 0;  l->size--;  return;  }  }else {  res.node = i.node->next;  res.node->prev = i.node->prev;  res.node->prev->next = res.node;    free(i.node);  i.node = NULL;  l->size--;    return;  }  }  void list\_delete\_front(List \*l)  {  Iterator i = list\_first(l);  Iterator res = list\_last(l);  if (list\_size == 0){  return;  }else if (Equal(&i, &res)) {  if(list\_size(l) == 1){  i.node = 0;  l->size--;  return;  }  }else {  l->head = i.node->next;  res.node = i.node->next;  res.node->prev = i.node->prev;  res.node->prev->next = res.node;    free(i.node);  i.node = NULL;  l->size--;    return;  }  }  void list\_destroy(List \*l)  {  struct Item \*i = l->head->next;  while (i != l->head) {  struct Item \*pi = i;  i = i->next;  free(pi);  }  free(l->head);  l->head = NULL;  l->size = 0;  }  void list\_print(List \*l)  {  if (list\_size(l) > 0) {  Iterator i = list\_first(l);  do {  printf("%s ", it\_fetch(&i));  i.node = i.node->next;  } while (i.node != l->head);    printf("\n");  } else {  printf("List is empty\n");  }  }  char\* list\_find(const List \*l, int k)  {  if (list\_size(l) == 0) {  return NULL;  } else{  Iterator it = list\_first(l);  if (k > 0){  for(int i = 0; i < k - 1; i++){  it = \*it\_next(&it);  }  } else {  for(int i = 0; i < -k; i++){  it = \*it\_prev(&it);  }  }  return it\_fetch(&it);  }    }  Iterator\* it\_next(Iterator \*i)  {  i->node = i->node->next;  return i;  }  Iterator\* it\_prev(Iterator \*i)  {  i-> node = i->node->prev;  return i;  }  void it\_store(const Iterator \*i, char t)  {  i->node->data = strdup(&t);  }  char\* it\_fetch( Iterator \*i)  {  return i->node->data;  } | **sortstation.c**  #include <stdio.h>  #include <stdlib.h>  #include "list.h"  #include "sortstation.h"  int have\_br(List \*l){  Iterator it = list\_first(l);  for(int i = 0; i < list\_size(l); i++){  if (it\_fetch(&it)[0] == '('){  return 1;  }  it = \*it\_next(&it);  }  return 0;  }  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;  }  List StrToRPN(const char\* string)  {  List out;  List stack;  list\_create(&out);  list\_create(&stack);  char\* str = (char\*)malloc(sizeof(char) \* (strlen(string) + 1));  strcpy(str, string);  char tmp[20];  bzero(tmp, 20);  char\* tmpC;  char c;  int num\_itr = 0;    for(int i = 0; i < strlen(str); i++) {  c = str[i];  if(is\_num(c)) {  tmp[num\_itr] = c;    list\_insert\_after(&out, tmp);    }  if(is\_alpha(c)){    tmp[0] = c;  list\_insert\_after(&out, tmp);  }  if(c == '(') {  tmp[0] = c;  list\_insert\_after(&stack, tmp);  } else if(c == ')') {  Iterator it = list\_last(&stack);    while(it\_fetch(&it)[0] != '(') {  if(list\_size(&stack) == 0){  printf("input error\n");  return out;  **}**  tmpC=it\_fetch(&it);  list\_insert\_after(&out, tmpC);  list\_delete\_last(&stack);  it = list\_last(&stack);  free (tmpC);    }  list\_delete\_last(&stack);    }else if(is\_op(c)) {  if (list\_size(&stack) != 0 && !have\_br(&stack)){  Iterator it = list\_last(&stack);  while (is\_op(it\_fetch(&it)[0]) &&\  it\_fetch(&it)[0] != c &&\  ((is\_left\_a(c) && op\_prior(c) <= op\_prior(it\_fetch(&it)[0])) ||\  (is\_right\_a(c) && op\_prior(c) < op\_prior(it\_fetch(&it)[0])))) {  tmpC = it\_fetch(&it);  list\_insert\_after(&out, tmpC);  list\_delete\_last(&stack);  it = list\_last(&stack);  free(tmpC);    }  }  tmp[0] = c;  list\_insert\_after(&stack, tmp);  }  }  while (list\_size(&stack) != 0) {  Iterator it = list\_last(&stack);  if (it\_fetch(&it)[0] == '(' || it\_fetch(&it)[0] == ')') {  printf("unclosed brake\n");  exit(1);  }  tmpC = it\_fetch(&it);  list\_delete\_last(&stack);  list\_insert\_after(&out, tmpC);  free(tmpC);    }  free(str);  list\_destroy(&stack);  list\_print(&out);  return out;  }  **Stack.h**  #ifndef \_STACK\_H\_  #define \_STACK\_H\_  #include <stdlib.h>  #include "tree.h"  typedef Node\* stack\_data\_type;  typedef struct {  stack\_data\_type \*data;  size\_t size;  size\_t top;  } Stack;  Stack\* stack\_create(void);  void stack\_delete (Stack \*\*stack);  int stack\_is\_empty(Stack \*stack);  void stack\_push(Stack \*stack, stack\_data\_type value);  stack\_data\_type stack\_pop(Stack \*stack);  void stack\_print(Stack \*stack);  size\_t stack\_size(Stack \*stack);  #endif  **main.c**  #include <stdio.h>  #include <stdlib.h>  #include "list.h"  #include "sortstation.h"  #include "tree.h"  #include "Stack.h"  Node\* RPNtoTree(List\* list)  {  if (list != NULL) {  Stack \*stack = stack\_create();  Node\* node;  char\* tmp;  Iterator it = list\_first(list);  while(list\_size(list) != 0) {    tmp = it\_fetch(&it);  list\_delete\_front(list);  it = list\_first(list);  if(is\_op(tmp[0])) {  node = createTree(tmp);  addRightTree(node, stack\_pop(stack));  addLeftTree(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(Node\* tree, char\* out) {  char\* tmpStr;  tmpStr = getTreeValue(tree);  if (tmpStr[0] == '\*') {  tmpStr = getTreeValue(getLeftSon(tree));  if (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] == '+') {  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])))) {  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])))) {  strcat(out, "(");  treeToStr(getRightSon(tree), out);  strcat(out, ")");  } else {  treeToStr(getRightSon(tree), out);  }    } else if (is\_num(tmpStr[0]) || is\_alpha(tmpStr[0])) {  strcat(out, tmpStr);  }  return out;  }  void variable\_rec(Node\* tree, char\* mas, int\* m)  {  if (tree != NULL){  //char tmp[10];  //bzero(tmp, 10);  if(is\_alpha(tree->data[0])){  for (int i = 1; i <= m[0]; i++){  if (tree->data[0] == mas[i]){  variable\_rec(tree->left, mas, m);  variable\_rec(tree->right, mas, m);  return;  }  }    m[0]++;  mas[m[0]] = tree->data[0];  }  variable\_rec(tree->left, mas, m);  variable\_rec(tree->right, mas, m);  }  }  int variable(Node\* tree){  char mas[50];  bzero(mas, 50);  int m[1];  m[0] = 0;  variable\_rec(tree, mas, m);  printf("number of variables: %d\n", m[0]);  return 0;  }  int main()  {  List list;  Iterator it;  Bintree tree;  char c;  char\* value;  int pos, m;  char sp[128];  char out[128];  bzero(out, 128);  list\_create(&list);  printf("list created\n");      scanf("%s", sp);  list = StrToRPN(sp);  printf("done\n");  Node\* eT = RPNtoTree(&list);  printTree(eT, 0);    printf("\n");  variable(eT);  printf("\n");  printf("to str again: \n");  treeToStr(eT, out);  printf("%s\n", out);  list\_destroy(&list);  treeDestroy(&eT);  return 0;  } |

**9.Дневник отладки** должен содержать дату и время сеансов отладки, и основные ошибки (ошибки в сценарии и программе, не стандартные операции) и краткие комментарии к ним. В дневнике отладки приводятся сведения об использовании других ЭВМ, существенном участии преподавателя и других лиц в написании и отладке программы.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № | Лаб.  или  дом. | Дата | Время | Событие | Действие по исправлению | Примечание |
|  |  |  |  |  |  |  |

1. Замечание автора по существу работы
2. Выводы: Алгоритм сортировочной станции — способ разбора математических выражений, представленных в инфиксной нотации. Может быть использован для получения вывода в виде обратной польской нотации или в виде а[бстрактного синтакfического дерева](https://ru.wikipedia.org/wiki/Абстрактное_синтаксическое_дерево). Алгоритм изобретен [Эдсгером Дейкстрой](https://ru.wikipedia.org/wiki/Эдсгер_Дейкстра) и назван им «алгоритм сортировочной станции», поскольку напоминает действие железнодорожной сортировочной станции. Каждый токен-число, оператор выводится только один раз, а также каждый оператор или круглая скобка будет добавлен и удален из стека по одному разу. Постоянное количество операций на токен, линейная сложность алгоритма O(*n*).

