Белгородский Государственный Технологический Университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

## Лабораторная работа №7 по теме: «Структура данных „дерево“»

Выполнил:  
студент группы ПВ-21  
Адаменко И. И.

Проверил:  
профессор  
Синюк В. Г.

Белгород  
2013

Цель работы: изучить структуры данных типа «дерево», научиться их программно реализовывать и использовать.

## Задания

1. Для СД типа «дерево» определить:
   1. Абстрактный уровень представления:
      1. Характер организованности и изменчивости.
      2. Набор допустимых операций.
   2. Физический уровень представления:
      1. Схему хранения.
      2. Объем памяти, занимаемый экземпляром СД.
      3. Формат внутреннего представления СД и его способ интерпретации.
      4. Характеристику допустимых значений.
      5. Тип доступа к элементам.
   3. Логический уровень представления СД. Способ описания СД и экземпляра СД на языке программирования.
2. Реализовать СД «дерево» в соответствии с вариантом индивидуального задания в виде модуля на языке C.
3. Разработать программу на языке C для решения задачи в соответствии с вариантом индивидуального задания с использованием модулей, полученных в результате выполнения п. 2.

### Задача

1. Построить дерево арифметического выражения, заданного в ОПЗ.  
   Операнды — целочисленные константы.  
   Операции — «+», «-», «\*» и взятие целого.
2. Вывести арифметическое выражение в ППЗ.
3. Вычислить значение по дереву арифметического выражения.

### Модуль для реализации дерева

Дерево в динамической памяти.

|  |  |
| --- | --- |
| 01 | **#ifndef** TREE\_H |
| 02 | **#define** TREE\_H |
| 03 |  |
| 04 | **const short** TreeOk = 0; |
| 05 | **const short** TreeNotMem = 1; |
| 06 | **const short** TreeEmpty = 2; |
| 07 | **static short** TreeError; *// Переменная ошибок* |
| 08 |  |
| 09 | **typedef** ... BaseTypeTree; *// Определить тип элемента* |
| 10 | **typedef** elementTree \*ptrelTree; |
| 11 | **typedef** struct { |
| 12 | BaseTypeTree data; |
| 13 | ptrelTree left; *// Левый сын* |
| 14 | ptrelTree right; *// Правый сын* |
| 15 | } elementTree; |
| 16 |  |
| 17 | **typedef** ptrelTree Tree; |
| 18 |  |
| 19 | *// Инициализация дерева* |
| 20 | **void** InitTree(Tree \*T); |
| 21 |  |
| 22 | *// Вставка в дерево* |
| 23 | **void** InsertTree(Tree \*T, BaseTypeTree E); |
| 24 |  |
| 25 | *// Чтение из дерева* |
| 26 | **void** ReadTree(Tree \*T, BaseTypeTree \*E); |
| 27 |  |
| 28 | *// Проверка на наличие левого сына* |
| 29 | **bool** isLeft(Tree \*T); |
| 30 |  |
| 31 | *// Проверка на наличие правого сына* |
| 32 | **bool** isRight(Tree \*T); |
| 33 |  |
| 34 | *// Переместиться к левому сыну* |
| 35 | **void** MoveToLeft(Tree \*T, Tree \*TS); |
| 36 |  |
| 37 | *// Переместиться к правому сыну* |
| 38 | **void** MoveToRight(Tree \*T, Tree \*TS); |
| 39 |  |
| 40 | *// Проверка дерева на пустоту* |
| 41 | **bool** isEmptyTree(Tree \*T); |
| 42 |  |
| 43 | *// Удалить дерево* |
| 44 | **void** MoveToRight(Tree \*T, Tree \*TS); |
| 45 |  |
| 46 | **#endif** |

## Характеристика СД типа «дерево»

1. Абстрактный уровень представления СД:
   1. По характеру организованности и изменчивости данный тип является деревом (иерархическая структура) и динамическим.
   2. К типу применимы такие операции, как: инициализация, создание корня, запись данных, чтение данных, переход к левому сыну и пр.
2. Физический уровень представления СД:
   1. Структура данных хранится связно.
   2. Размер памяти, занимаемый каждым экземпляром СД зависит от максимального возможного количества элементов в нее входящих (и от выбранной схемы хранения отдельного элемента).
   3. В зависимости от схемы хранения, меняется и формат внутреннего представления СД.
   4. Допустимыми являются все значения типа данных, определенного как базовый.
   5. Тип доступа к элементам данной СД последовательный.
3. На логическом уровне представления СД, то есть, на языке программирования С экземпляр СД описывается следующим образом: Tree T;

## Текст модуля для реализации СД типа «дерево»

### Tree.h

|  |  |
| --- | --- |
| 01 | **#ifndef** TREE\_H |
| 02 | **#define** TREE\_H |
| 03 |  |
| 04 | **#include** "StackList.h" |
| 05 |  |
| 06 | **const short** TreeOk = 0; |
| 07 | **const short** TreeNotMem = 1; |
| 08 | **const short** TreeEmpty = 2; |
| 09 | **short** TreeError; |
| 10 |  |
| 11 | **typedef** **int** BaseTypeTree; |
| 12 | **typedef** **struct** elementTree \*ptrelTree; |
| 13 | **typedef** **struct** elementTree { |
| 14 | BaseTypeTree data; |
| 15 | ptrelTree left; |
| 16 | ptrelTree right; |
| 17 | } elementTree; |
| 18 |  |
| 19 | **void** InitTree(Tree \*T); |
| 20 | **void** InsertTree(Tree \*T, BaseTypeTree E); |
| 21 | **void** ReadTree(Tree \*T, BaseTypeTree \*E); |
| 22 | **bool** isLeft(Tree \*T); |
| 23 | **bool** isRight(Tree \*T); |
| 24 | **void** MoveToLeft(Tree \*T, Tree \*TS); |
| 25 | **void** MoveToRight(Tree \*T, Tree \*TS); |
| 26 | **bool** isEmptyTree(Tree \*T); |
| 27 | **void** DelTree(Tree \*T); |
| 28 |  |
| 29 | **#endif** |

### Tree.cpp

|  |  |
| --- | --- |
| 01 | **void** InitTree(Tree \*T) { |
| 02 | \*T = (Tree)malloc(sizeof(Tree)); |
| 03 | **if** (\*T == NULL) { |
| 04 | TreeError = TreeNotMem; |
| 05 | **return**; |
| 06 | } |
| 07 | (\*T)->data = NULL; |
| 08 | (\*T)->left = NULL; |
| 09 | (\*T)->right = NULL; |
| 10 |  |
| 11 | TreeError = TreeOk; |
| 12 | } |
| 13 |  |
| 14 | **void** InsertTree(Tree \*T, BaseTypeTree E) { |
| 15 | **if** (\*T != NULL) { |
| 16 | (\*T)->data = E; |
| 17 | } |
| 18 | **else** { |
| 19 | TreeError = TreeEmpty; |
| 20 | } |
| 21 | } |
| 22 |  |
| 23 | **void** ReadTree(Tree \*T, BaseTypeTree \*E) { |
| 24 | **if** (\*T != NULL) { |
| 25 | \*E = (\*T)->data; |
| 26 | } |
| 27 | **else** { |
| 28 | TreeError = TreeEmpty; |
| 29 | } |
| 30 | } |
| 31 |  |
| 32 | **bool** isLeft(Tree \*T) { |
| 33 | **return** (\*T)->left != NULL; |
| 34 | } |
| 35 |  |
| 36 | **bool** isRight(Tree \*T) { |
| 37 | **return** (\*T)->right != NULL; |
| 38 | } |
| 39 |  |
| 40 | **void** MoveToLeft(Tree \*T, Tree \*TS) { |
| 41 | **if** (TS != NULL) { |
| 42 | **if** (\*TS == NULL) { |
| 43 | TreeError = TreeEmpty; |
| 44 | **return**; |
| 45 | } |
| 46 | (\*T)->left = \*TS; |
| 47 | } |
| 48 | (\*T) = (\*T)->left; |
| 49 | } |
| 50 |  |
| 51 | **void** MoveToRight(Tree \*T, Tree \*TS) { |
| 52 | **if** (TS != NULL) { |
| 53 | **if** (\*TS == NULL) { |
| 54 | TreeError = TreeEmpty; |
| 55 | **return**; |
| 56 | } |
| 57 | (\*T)->right = \*TS; |
| 58 | } |
| 59 | (\*T) = (\*T)->right; |
| 60 | } |
| 61 |  |
| 62 | **bool** isEmptyTree(Tree \*T) { |
| 63 | **return** (((\*T)->left == NULL) && ((\*T)->right == NULL)); |
| 64 | } |
| 65 |  |
| 66 | **void** DelTree(Tree \*T) { |
| 67 | **while** ((\*T)->left != NULL) { |
| 68 | MoveToLeft(T, NULL); |
| 69 | DelTree(T); |
| 70 | } |
| 71 | **while** ((\*T)->right != NULL) { |
| 72 | MoveToRight(T, NULL); |
| 73 | DelTree(T); |
| 74 | } |
| 75 | (\*T)->left = NULL; |
| 76 | (\*T)->right = NULL; |
| 77 | (\*T)->data = NULL; |
| 78 | } |

## Текст программы для решения задачи

### main.cpp

|  |  |
| --- | --- |
| 001 | **#include** "Tree.h" |
| 002 |  |
| 003 | **void** PrintPrefix(Tree \*T); |
| 004 | **void** PrintPostfix(Tree \*T); |
| 005 | **void** CalcTree(Tree \*T, Stack \*S); |
| 006 | **void** Parser(Tree \*T); |
| 007 |  |
| 008 | **int** main(**void**) { |
| 009 | Tree root, tmp, result; |
| 010 | Stack S; |
| 011 |  |
| 012 | InitTree(&root); |
| 013 | InitTree(&tmp); |
| 014 | Parser(&root); |
| 015 | tmp = root; |
| 016 |  |
| 017 | printf(*"Postfix: "*); |
| 018 | PrintPostfix(&root); |
| 019 | root = tmp; |
| 020 |  |
| 021 | printf(*"\nPrefix: "*); |
| 022 | PrintPrefix(&root); |
| 023 | root = tmp; |
| 024 |  |
| 025 | InitStack(&S); |
| 026 | CalcTree(&root, &S); |
| 027 |  |
| 028 | InitTree(&result); |
| 029 | GetStack(&S, &result); |
| 030 | printf(*"\nResult: %d\n"*, result->data); |
| 031 |  |
| 032 | getchar(); |
| 033 | } |
| 034 |  |
| 035 | **void** PrintPrefix(Tree \*T) { |
| 036 | **if** (T != NULL) { |
| 037 | **switch** ((\*T)->data) { |
| 038 | **case** -1: |
| 039 | printf(*"%c "*, '+'); |
| 040 | **break**; |
| 041 | **case** -2: |
| 042 | printf(*"%c "*, '-'); |
| 043 | **break**; |
| 044 | **case** -3: |
| 045 | printf(*"%c "*, '\*'); |
| 046 | **break**; |
| 047 | **case** -4: |
| 048 | printf(*"%c "*, '/'); |
| 049 | **break**; |
| 050 | **default:** |
| 051 | printf(*"%c "*, (\*T)->data + '0'); |
| 052 | **break**; |
| 053 | } |
| 054 |  |
| 055 | **if** (isRight(T)) PrintPrefix(&(\*T)->right); |
| 056 | **if** (isLeft(T)) PrintPrefix(&(\*T)->left); |
| 057 | } |
| 058 | } |
| 059 |  |
| 060 | **void** PrintPostfix(Tree \*T) { |
| 061 | **if** (T != NULL) { |
| 062 | **if** (isRight(T)) PrintPostfix(&(\*T)->right); |
| 063 | **if** (isLeft(T)) PrintPostfix(&(\*T)->left); |
| 064 |  |
| 065 | **switch** ((\*T)->data) { |
| 066 | **case** -1: |
| 067 | printf(*"%c "*, '+'); |
| 068 | **break**; |
| 069 | **case** -2: |
| 070 | printf(*"%c "*, '-'); |
| 071 | **break**; |
| 072 | **case** -3: |
| 073 | printf(*"%c "*, '\*'); |
| 074 | **break**; |
| 075 | **case** -4: |
| 076 | printf(*"%c "*, '/'); |
| 077 | **break**; |
| 078 | default: |
| 079 | printf(*"%c "*, (\*T)->data + '0'); |
| 080 | **break**; |
| 081 | } |
| 082 | } |
| 083 | } |
| 084 |  |
| 085 | **void** CalcTree(Tree \*T, Stack \*S) { |
| 086 | **if** (isRight(T)) CalcTree(&(\*T)->right, S); |
| 087 | **if** (isLeft(T)) CalcTree(&(\*T)->left, S); |
| 088 | **if** ((\*T)->data >= 0) { |
| 089 | PutStack(S, \*T); |
| 090 | } |
| 091 | **else** { |
| 092 | Tree tmpOne, tmpTwo; |
| 093 | **int** answer; |
| 094 | InitTree(&tmpOne); |
| 095 | InitTree(&tmpTwo); |
| 096 | GetStack(S, &tmpOne); |
| 097 | GetStack(S, &tmpTwo); |
| 098 | **switch** ((\*T)->data) { |
| 099 | **case** -1: |
| 100 | answer = tmpTwo->data + tmpOne->data; |
| 101 | **break**; |
| 102 | **case** -2: |
| 103 | answer = tmpTwo->data - tmpOne->data; |
| 104 | **break**; |
| 105 | **case** -3: |
| 106 | answer = tmpTwo->data \* tmpOne->data; |
| 107 | **break**; |
| 108 | **case** -4: |
| 109 | answer = tmpTwo->data / tmpOne->data; |
| 110 | **break**; |
| 111 | } |
| 112 | InsertTree(&tmpOne, answer); |
| 113 | PutStack(S, tmpOne); |
| 114 | } |
| 115 | } |
| 116 |  |
| 117 | **void** Parser(Tree \*T) { |
| 118 | char tmp = '.'; |
| 119 | Stack S; |
| 120 | Tree tmpTree, tmpLSon, tmpRSon, tmpTmp; |
| 121 |  |
| 122 | InitStack(&S); |
| 123 | **while** (tmp != '\n') { |
| 124 | tmp = getchar(); |
| 125 | **if** (tmp == ' ') continue; |
| 126 | **if** (tmp >= '0' && tmp <= '9') { |
| 127 | InitTree(&tmpTree); |
| 128 | InsertTree(&tmpTree, tmp - '0'); |
| 129 | PutStack(&S, tmpTree); |
| 130 | } |
| 131 | **if** (tmp == '+' || tmp == '-' || tmp == '\*' || tmp == '/') { |
| 132 | InitTree(&tmpTree); |
| 133 | tmpTmp = tmpTree; |
| 134 | **switch** (tmp) { |
| 135 | **case** '+': |
| 136 | InsertTree(&tmpTree, -1); |
| 137 | **break**; |
| 138 | **case** '-': |
| 139 | InsertTree(&tmpTree, -2); |
| 140 | **break**; |
| 141 | **case** '\*': |
| 142 | InsertTree(&tmpTree, -3); |
| 143 | **break**; |
| 144 | **case** '/': |
| 145 | InsertTree(&tmpTree, -4); |
| 146 | **break**; |
| 147 | } |
| 148 |  |
| 149 | GetStack(&S, &tmpLSon); |
| 150 | MoveToLeft(&tmpTree, &tmpLSon); |
| 151 | tmpTree = tmpTmp; |
| 152 |  |
| 153 | GetStack(&S, &tmpRSon); |
| 154 | MoveToRight(&tmpTree, &tmpRSon); |
| 155 | tmpTree = tmpTmp; |
| 156 |  |
| 157 | PutStack(&S, tmpTree); |
| 158 | } |
| 159 | } |
| 160 |  |
| 161 | GetStack(&S, T); |
| 162 | } |

## Результаты работы программы

|  |  |  |
| --- | --- | --- |
| № п/п | Исходные данные | Результат работы программы |
| Входная строка | Сообщение на экране |
| 1 | 8 2 5 \* + 1 3 2 \* + 4 - / | Postfix: 8 2 5 \* + 1 3 2 \* + 4 - /  Prefix: / + 8 \* 2 5 - + 1 \* 3 2 4  Result: 6 |