Белгородский Государственный Технологический Университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

## Лабораторная работа №5 по теме: «Структуры данных „линейные списки“»

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

Проверил:  
профессор  
Синюк В. Г.

Белгород  
2013

Цель работы: изучить структуру данных типа «линейный список», научиться программно реализовывать и использовать.

## Задания

1. Для СД типа «линейный список» 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. Разработать СД «линейный список» 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 модуля, полученного в результате выполнения п. 2.

### Задача

Многочлен *P(x)=anxn + an-1xn-1 + ... + a1x + a0* с целыми коэффициентами можно представить в виде списка, причем если *ai = 0,* то соответствующее звено не включать в список. Определить логическую функцию *РАВНО(p, q),* проверяющую на равенство многочлены *p* и *q.*

### Модуль

ОЛС в динамической памяти (базовый тип определяется задачей). Выделение памяти под информационную часть элемента ОЛС и запись в нее значения осуществляется при выполнении процедуры *PutList.* При выполнении процедуры *GetList* память, занимаемая элементом, освобождается.

## Характеристика СД типа «линейный список»

1. Абстрактный уровень представления СД:
   1. По характеру организованности и изменчивости данный тип упорядоченный и динамический.
   2. К типу применимы такие операции, как: инициализация, включение элемента, исключение элемента, чтение текущего элемента, переход в начало списка и пр.
2. Физический уровень представления СД:
   1. Структура данных хранится связно.
   2. Размер памяти, занимаемой каждым экземпляром СД зависит от количества элементов в нее входящих (и от выбранной схемы хранения).
   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: List L;

## Текст модуля для реализации типа «линейный список»

### list.h

|  |  |
| --- | --- |
| 01 | **#ifndef** LIST\_H |
| 02 | **#define** LIST\_H |
| 03 |  |
| 04 | **#include** <stdio.h> |
| 05 | **#include** <stdlib.h> |
| 06 |  |
| 07 | **const short** ListOk = 0; |
| 08 | **const short** ListNotMem = 1; |
| 09 | **const short** ListEmpty = 2; |
| 10 | **const short** ListFull = 3; |
| 11 |  |
| 12 | **typedef** **struct** { |
| 13 | **int** power; |
| 14 | **int** value; |
| 15 | } basetype; |
| 16 | **typedef** basetype base; |
| 17 | **typedef** **struct** element \* ptrel; |
| 18 | **typedef** **struct** element { |
| 19 | base data; |
| 20 | ptrel next; |
| 21 | } element; |
| 22 | **typedef** **struct** { |
| 23 | ptrel Start; |
| 24 | ptrel ptr; |
| 25 | **unsigned** **int** N; |
| 26 | } List; |
| 27 |  |
| 28 | **static short** ListError; |
| 29 |  |
| 30 | **void** InitList(List \*L, **unsigned** **int** N); |
| 31 | **void** PutList(List \*L, base E); |
| 32 | **void** GetList(List \*L, base \*E); |
| 33 | **void** ReadList(List \*L, base \*E); |
| 34 | **int** EmptyList(List \*L); |
| 35 | **int** FullList(List \*L); |
| 36 | **unsigned** **int** Count(List \*L); |
| 37 | **void** BeginPtr(List \*L); |
| 38 | **void** EndPtr(List \*L); |
| 39 | **void** MovePtr(List \*L); |
| 40 | **void** MoveTo(List \*L, **unsigned** **int** n); |
| 41 | **void** ClearList(List \*L); |
| 42 | **void** CopyList(List \*L1, List \*L2); |
| 43 |  |
| 44 | **#endif** |

### list.c

|  |  |
| --- | --- |
| 001 | **#include** *"list.h"* |
| 002 |  |
| 003 | **void** InitList(List \*L, **unsigned** **int** N) { |
| 004 | L->Start = (ptrel)malloc(sizeof(element)); |
| 005 |  |
| 006 | **if** (L->Start == NULL) { |
| 007 | ListError = ListNotMem; |
| 008 | **return**; |
| 009 | } |
| 010 |  |
| 011 | L->ptr = L->Start; |
| 012 | L->ptr->next = NULL; |
| 013 | L->N = N; |
| 014 | ListError = ListOk; |
| 015 | } |
| 016 |  |
| 017 | **void** PutList(List \*L, base E) { |
| 018 | ptrel tmp = (ptrel)malloc(sizeof(element)); |
| 019 |  |
| 020 | **if** (tmp == NULL) { |
| 021 | ListError = ListNotMem; |
| 022 | **return**; |
| 023 | } |
| 024 | **if** (Count(L) == L->N) { |
| 025 | ListError = ListFull; |
| 026 | **free**(tmp); |
| 027 | **return**; |
| 028 | } |
| 029 | tmp->data = E; |
| 030 | tmp->next = L->ptr->next; |
| 031 | L->ptr->next = tmp; |
| 032 |  |
| 033 | L->ptr = L->ptr->next; |
| 034 | } |
| 035 |  |
| 036 | **void** GetList(List \*L, base \*E) { |
| 037 | ptrel tmp = L->ptr, tmpnext; |
| 038 |  |
| 039 | **if** (L->Start == NULL) { |
| 040 | ListError = ListEmpty; |
| 041 | **return**; |
| 042 | } |
| 043 | tmpnext = L->ptr->next->next; |
| 044 | \*E = L->ptr->next->data; |
| 045 | **free**(L->ptr->next); |
| 046 | L->ptr->next = tmpnext; |
| 047 | L->ptr = tmp; |
| 048 | } |
| 049 |  |
| 050 | **void** ReadList(List \*L, base \*E) { |
| 051 | **if** (L->Start == NULL) { |
| 052 | ListError = ListEmpty; |
| 053 | **return**; |
| 054 | } |
| 055 | \*E = L->ptr->data; |
| 056 | } |
| 057 |  |
| 058 | **int** EmptyList(List \*L) { |
| 059 | **return** (L->Start == NULL) ? 1 : 0; |
| 060 | } |
| 061 |  |
| 062 | **int** FullList(List \*L) { |
| 063 | **return** (Count(L) == L->N) ? 1 : 0; |
| 064 | } |
| 065 |  |
| 066 | **unsigned** **int** Count(List \*L) { |
| 067 | **unsigned** **int** counter = 0; |
| 068 | ptrel tmp = L->Start; |
| 069 |  |
| 070 | **while** (tmp->next != NULL) { |
| 071 | counter++; |
| 072 | tmp = tmp->next; |
| 073 | } |
| 074 |  |
| 075 | **return** counter; |
| 076 | } |
| 077 |  |
| 078 | **void** BeginPtr(List \*L) { |
| 079 | L->ptr = L->Start; |
| 080 | } |
| 081 |  |
| 082 | **void** EndPtr(List \*L) { |
| 083 | L->ptr = L->Start; |
| 084 | **if** (L->ptr->next == NULL) { |
| 085 | ListError = ListEmpty; |
| 086 | **return**; |
| 087 | } |
| 088 | **while** (L->ptr->next != NULL) { |
| 089 | L->ptr = L->ptr->next; |
| 090 | } |
| 091 | } |
| 092 |  |
| 093 | **void** MovePtr(List \*L) { |
| 094 | **if** (L->ptr->next == NULL) { |
| 095 | ListError = ListFull; |
| 096 | **return**; |
| 097 | } |
| 098 | L->ptr = L->ptr->next; |
| 099 | } |
| 100 |  |
| 101 | **void** MoveTo(List \*L, **unsigned** **int** n) { |
| 102 | **unsigned** **int** counter = 1; |
| 103 | ptrel tmp = L->Start; |
| 104 |  |
| 105 | **while** (counter++ != n) { |
| 106 | tmp = tmp->next; |
| 107 | **if** (tmp == NULL) { |
| 108 | ListError = ListFull; |
| 109 | **return**; |
| 110 | } |
| 111 | } |
| 112 | L->ptr = tmp; |
| 113 | } |
| 114 |  |
| 115 | **void** ClearList(List \*L) { |
| 116 | ptrel tmp, tmpnext; |
| 117 | tmp = L->Start; |
| 118 | tmpnext = tmp->next; |
| 119 |  |
| 120 | **while** (tmpnext != NULL) { |
| 121 | tmp = tmpnext; |
| 122 | tmpnext = tmpnext->next; |
| 123 | **free**(tmp); |
| 124 | } |
| 125 |  |
| 126 | **free**(L->Start); |
| 127 | ListError = ListEmpty; |
| 128 | } |
| 129 |  |
| 130 | **void** CopyList(List \*L1, List \*L2) { |
| 131 | ptrel tmp = L1->Start; |
| 132 |  |
| 133 | **while**(tmp->next != NULL) { |
| 134 | PutList(L2, tmp->next->data); |
| 135 | tmp = tmp->next; |
| 136 | **if** (FullList(L2)) { |
| 137 | ListError = ListFull; |
| 138 | **return**; |
| 139 | } |
| 140 | } |
| 141 | } |

## Текст программы для решения задачи

|  |  |
| --- | --- |
| 01 | **#include** *"list.h"* |
| 02 |  |
| 03 | **bool** isEqual(List \*L1, List \*L2); |
| 04 | **void** input(List \*L); |
| 05 |  |
| 06 | **int** main() { |
| 07 | List l, l2; |
| 08 | input(&l); |
| 09 | input(&l2); |
| 10 | **if** (isEqual(&l, &l2)) { |
| 11 | **printf**(*"%s\n"*, *"true"*); |
| 12 | } **else** { |
| 13 | **printf**(*"%s\n"*, *"false"*); |
| 14 | } |
| 15 |  |
| 16 | **return** 0; |
| 17 | } |
| 18 |  |
| 19 | **bool** isEqual(List \*L1, List \*L2) { |
| 20 | ptrel tmpl1 = L1->ptr, tmpl2 = L2->ptr; |
| 21 |  |
| 22 | BeginPtr(L1); |
| 23 | BeginPtr(L2); |
| 24 |  |
| 25 | **while** ((L1->ptr->next != NULL) && (L2->ptr->next !=NULL)) { |
| 26 | **if** ((L1->ptr->data.power != L2->ptr->data.power) || (L1->ptr->data.value != L2->ptr->data.value)) { |
| 27 | L1->ptr = tmpl1; |
| 28 | L2->ptr = tmpl2; |
| 29 | **return** **false**; |
| 30 | } |
| 31 | L1->ptr = L1->ptr->next; |
| 32 | L2->ptr = L2->ptr->next; |
| 33 | } |
| 34 |  |
| 35 | **if** ((L1->ptr->next != NULL) || (L2->ptr->next !=NULL)) { |
| 36 | L1->ptr = tmpl1; |
| 37 | L2->ptr = tmpl2; |
| 38 | **return** **false**; |
| 39 | } |
| 40 |  |
| 41 | **return** **true**; |
| 42 | } |
| 43 |  |
| 44 | **void** input(List \*L) { |
| 45 | **int** power, value; |
| 46 | base e; |
| 47 |  |
| 48 | **printf**(*"%s\n"*, *"Enter the power and coefficient,*  *separated by space.\nEnter -1 for exit."*); |
| 49 | **scanf**(*"%d"*, &e.power); |
| 50 | InitList(L, 300); |
| 51 | **while** (e.power > -1) { |
| 52 | **scanf**(*"%d"*, &e.value); |
| 53 | **if** (e.value != 0) { |
| 54 | PutList(L, e); |
| 55 | } |
| 56 | **scanf**(*"%d"*, &e.power); |
| 57 | } |
| 58 | EndPtr(L); |
| 59 | } |