МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ УНИВЕРСИТЕТ
имени академика С.П. КОРОЛЕВА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Списки и деревья

Электронные методические указания
к лабораторной работе № 2

Самара
2011

Составители: МЯСНИКОВ Евгений Валерьевич
ПОПОВ Артем Борисович

В лабораторной работе № 2 по дисциплине "Языки и методы программирования" изучаются принципы работы на C++ с динамическими структурами данных: списками и деревьями. Приводятся краткие теоретические и справочные сведения, необходимые для выполнения лабораторных работ. Дан пример выполнения лабораторной работы.

Методические указания предназначены для студентов факультета информатики, направление 010400 – Прикладная математика и информатика, бакалавриат (010400.62)/магистратура (010400.68, магистерская программа – Технологии параллельного программирования и суперкомпьютинг).

Содержание

Цель работы: Приобретение навыков работы с динамической памятью и указателями на C++. Изучение принципов работы с динамическими структурами данных: списками и деревьями.

1 Теоретические основы лабораторной работы

1.1 Списки

Линейным списком называют последовательность однотипных элементов, возможно, с повторами. Обычно линейный список записывают в следующем виде:

$L=(a_1, a_2, \ldots, a_n)$,

где a_i, i=1..n – элементы списка.

Графически список обозначают следующим образом:

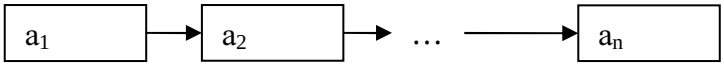


Рис. 1 – Графическое представление списка

Длиной списка называют количество элементов списка. При n=0 список пуст. Первый элемент списка a_1 называют также ***головным*** элементом списка. Последний элемент a_n – ***концевым*** элементом списка или ***хвостом***.

При работе со списками используются следующие типовые операции:

- обращение к элементу списка,
- поиск элемента списка,
- вставка элемента списка,
- удаление элемента списка,
- упорядочивание элементов списка.

При этом эффективность выполнения приведенных выше операций зависит от способа хранения списка в памяти ЭВМ. Выделяют два основных способа хранения списков: последовательное хранение и связное хранение списков. При ***последовательном хранении*** элементы списка хранятся в памяти последовательно, в смежных областях. Реализуется такой способ с использованием массива. При ***связном хранении*** элементы списка представляют собой отдельные объекты (структуры), размещаемые в произвольных областях памяти и связываемые в единую последовательность с использованием специальных полей связи.

Реализовать последовательный способ хранения списков не представляет большой сложности, и мы оставим это в качестве упражнения. Рассмотрим, как можно организовать связный способ хранения списков.

1.1.1 Односвязные списки

При использовании ***односвязных списков*** для хранения элементов объявляют структуру, имеющую единственное поле связи:

```
struct ListElm{
    T info;
    ListElm *link;
};
```

Здесь info – поле некоторого типа T, содержащее информацию, хранящуюся в элементе списка, link – поле связи со следующим элементом, представляющее собой типизированный указатель. При связном способе хранения элементов для получения доступа к списку используют указатель на головной элемент:

```
ListElm *head;
```

Графически изобразить односвязный список можно следующим образом:

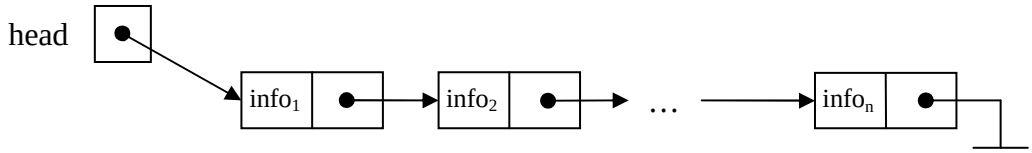
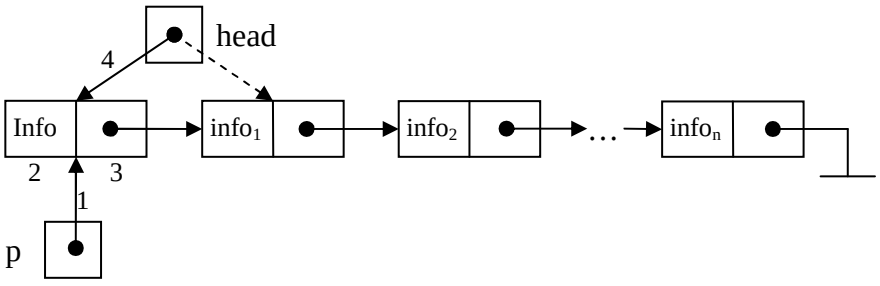


Рис. 2 – Графическое представление списка

Приведем примеры реализации некоторых операций при работе с односвязными списками.

Добавление элемента в голову списка

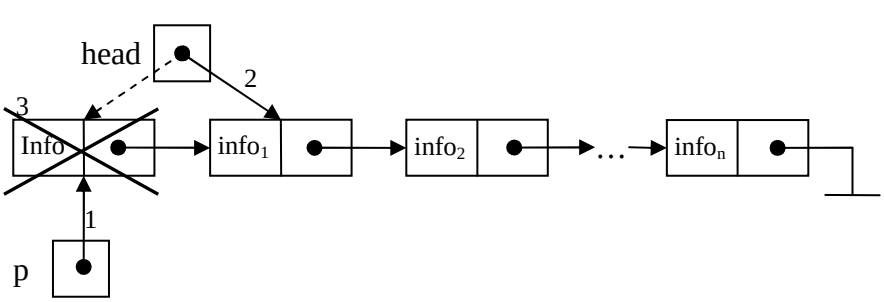



```
ListElm* p =
    new ListElm; //1
p->info = Info;  //2
p->link = head;  //3
head = p;        //4
```

Рис. 3 – Добавление элемента в голову списка

Здесь и далее на рисунке и в программе цифрами помечены совершаемые действия, Info – значение информационного поля добавляемого элемента.

Удаление элемента из головы списка




```
ListElm* p = head; //1
head = p->link;    //2
delete p;          //3
p = NULL;
```

Рис. 4 – Удаление элемента из головы списка

Поиск элемента в списке по информационному полю

```
ListElm* p = head;  // 1
while (p && (p->info != Info))  p = p->link;    //2
```

По выходу из цикла указатель p будет содержать адрес искомого элемента. В том случае, если элемент не будет найден, указатель будет нулевым.

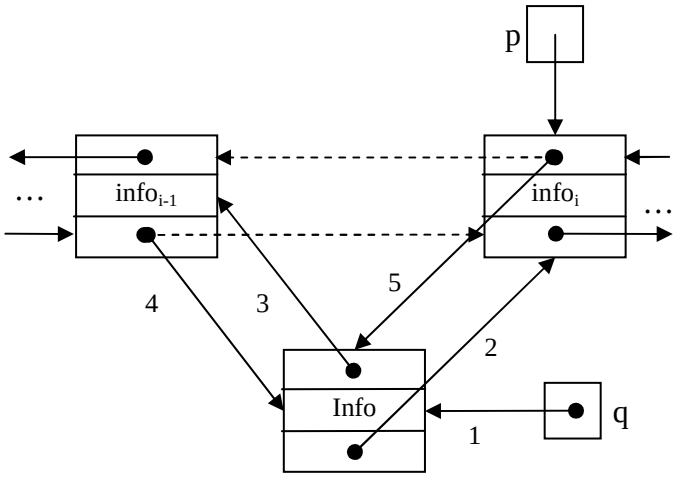
Помимо приведенных выше операций с односвязными списками также часто используются такие операции, как добавление и удаление элементов, следующих за указанным. Реализовать такие операции не представляет никаких трудностей. Однако, например, вставку элемента перед указанным или удаление указанного элемента выполнить уже сложнее, так как для этого необходимо получить указатель на предыдущий элемент списка.

1.1.2 Двусвязные списки

Помимо односвязных списков распространение получили и двусвязные списки. Каждый элемент такого списка содержит два поля связи, связывающих его с предыдущим и последующим элементами.

```
struct DListElm{
    T info;
    ListElm *prev, *next;
};
```

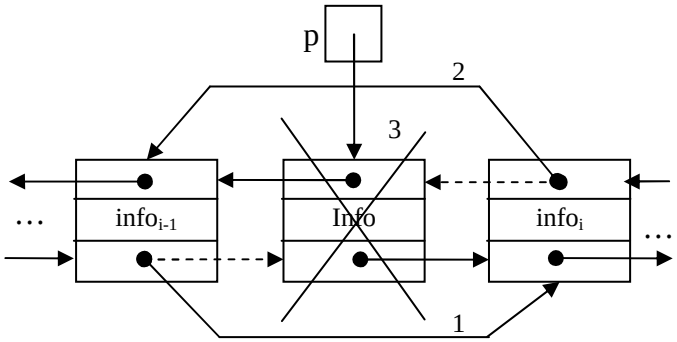
В связи с тем, что в таких списках от любого элемента можно получить доступ, как к предыдущему, так и последующему элементу, некоторые операции выполняются проще, чем в односвязном списке. В частности, облегчается выполнение таких операций, как удаление элемента, на котором установлен указатель, и вставка элемента перед элементом, на котором установлен указатель. Приведем примеры реализации этих операций.



Вставка элемента

```
DListElm* q =
    new DListElm;  //1
q->info = Info;
q->next = p;       //2
q->prev = p->prev; //3
p->prev->next = q; //4
p->prev = q;
```

Рис. 5 – Вставка элемента в двусвязный список



Удаление элемента

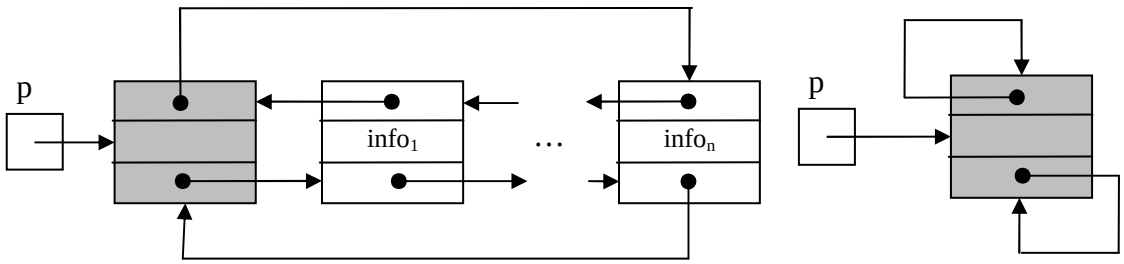
```
p->prev->next = p->next; //2
p->next->prev = p->prev; //3
delete p;
p = NULL;
```

Рис. 6 – Удаление элемента из двусвязного списка

Приведенный выше код вставки элемента будет приводить к ошибке в том случае, если q указывает на головной элемент списка. Код удаления элемента будет приводить к ошибкам, как для головного, так и для концевого элементов списка. Поэтому такие случаи должны обрабатываться отдельно.

1.2.3 Циклические списки

Избавиться от проблем с головным и концевым элементами помогает использование циклических списков. В таких списках вводится специальный головной элемент, который создается при создании списка и не удаляется. Его поле связи с предыдущим элементом содержит указатель на концевой элементом списка, а поле связи концевого элемента со следующим содержит указатель на головной элемент.



(а) (б)

Рис. 7 – Графическое представление циклического списка: а) непустого, б) пустого

Особенностью циклического списка является также и то, что когда список пуст, поля связи головного элемента указывают на сам этот элемент. Таким образом, в циклических списках наличие выделенного головного элемента позволяет различить ситуации «список не существует» и «список пуст». Работа со всеми элементами циклического списка осуществляется единообразно. Следует отметить, что организовать циклический список можно и на основе односвязного списка.

1.2 Деревья

Деревом T называют конечное множество узлов, в котором имеется один выделенный узел t, называемый ***корнем***, а остальные узлы разбиты на M≥0 непересекающихся множеств $T_1,T_2,\ldots,T_M$, каждое из которых является деревом и называется ***поддеревом*** узла t.

Корень дерева t также называют предшественником, предком или родительским узлом по отношению к корням $t_1,t_2,\ldots,t_N$ своих поддеревьев $T_1,T_2,\ldots,T_N$. Узлы $t_1,t_2,\ldots,t_N$ называют преемниками, потомками или дочерними узлами.

Степенью узла называются число потомков данного узла. ***Степенью дерева*** называют наибольшую из степеней всех его узлов. Узлы нулевой степени не имеют потомков и называются ***листьями*** (или ***концевыми узлами***) дерева.

Полным называется дерево, у которого степень всех узлов, не являющихся листьями, равна степени дерева.

Уровнем узла называется выраженная в числе узлов длина пути, соединяющая этот узел с корнем дерева. ***Высотой дерева*** называют максимальный уровень для узлов дерева.

Различают упорядоченные и неупорядоченные деревья. Дерево называют ***упорядоченным***, если для любого узла дерева, за исключением корня, известно, каким по счету потомком является этот узел.

Для решения многих задач требуется организовать ***обход*** (прохождение) дерева - просмотр всех узлов дерева в определенном порядке. Различают два основных алгоритма обхода деревьев: прямой и обратный. При ***прямом обходе*** для каждого поддерева сначала просматривается корень, а затем все поддеревья данного корня в прямом порядке. При ***обратном обходе*** сначала в обратном порядке просматривается каждое поддерево, а уже затем - корень. В любом случае при обходе все узлы дерева просматриваются по одному разу, то есть порождается линейная последовательность узлов дерева.

1.2.1 Бинарные деревья

Бинарным деревом называют упорядоченное дерево второй степени. При этом первого потомка любого узла бинарного дерева называют левым поддеревом этого узла, а второго потока – правым поддеревом. Узел может не иметь потомков вообще, может иметь только левое или только правое поддерево, может иметь оба поддерева. Бинарное дерево может быть пустым.

Для бинарных деревьев могут применяться как прямой, так и обратный алгоритмы обхода. При прямом алгоритме обхода, называемом также ***нисходящим***, для каждого узла дерева, начиная с корня, сначала обрабатывается сам узел, затем обрабатывается левое поддерево в прямом порядке, а затем - правое поддерево в прямом порядке.

При обратном алгоритме обхода, называемом также ***восходящим***, сначала обрабатывается левое поддерево в обратном порядке, затем обрабатывается правое поддерево в обратном порядке, а уже затем – корень.

Для бинарных деревьев применим также ***симметричный*** (***смешанный***) алгоритм обхода, при котором для каждого узла дерева, начиная с корня, сначала просматривается левое поддерево в симметричном порядке, затем – сам узел, а затем – правое поддерево в симметричном порядке.

Следует отметить, что алгоритмы обхода часто называют по первым буквам просматриваемых узлов: прямой – КЛП, обратный – ЛПК, симметричный – ЛКП.

Хранение бинарных деревьев может быть организовано, как последовательным, так и связным способом. При последовательном хранении могут быть использованы, как массивы, так и способы, допускающие представление деревьев общего вида, например, уровневое или скобочное представления. Нас, однако, будет интересовать связный способ хранения бинарных деревьев. При таком способе каждый узел бинарного дерева представляет собой структуру, имеющую два поля связи, используемые для хранения указателей на левое и правое поддеревья узла, соответственно.

```
struct TreeElm {
    T info;
    TreeElm *left, *right;
};
```

Здесь info – поле некоторого типа T, содержащее информацию, хранящуюся в узле дерева, left и right – поля связи с поддеревьями узла, представляющие собой типизированные указатели. Для получения доступа к дереву используют указатель на головной элемент:

```
var root: PTree;
```

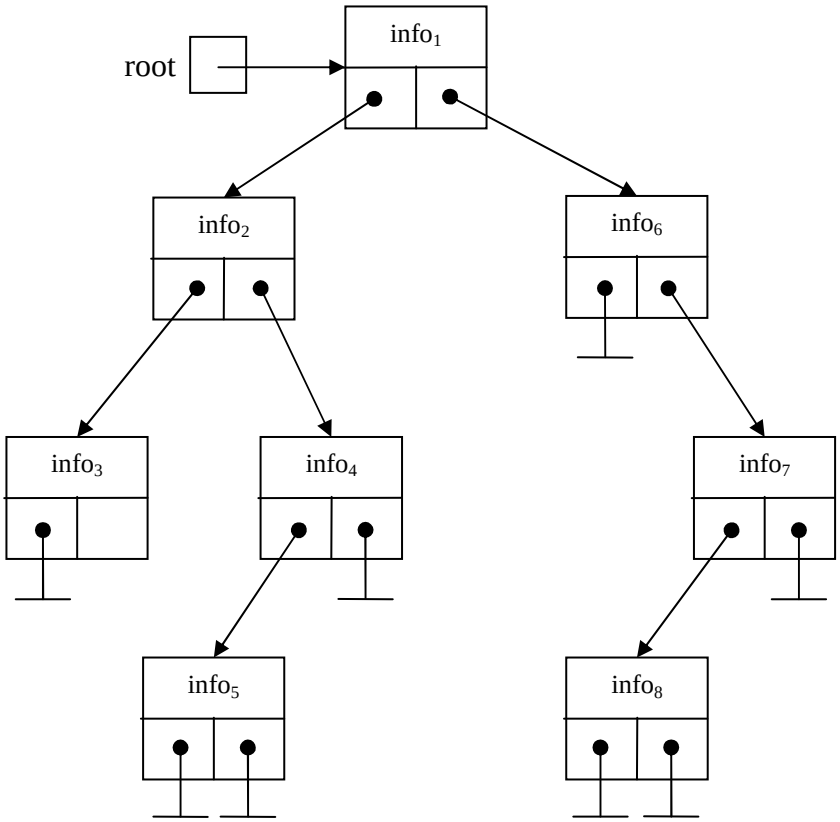


Рис. 8 – Графическое представление бинарного дерева

Следует отметить, что приведенный способ хранения не единственно возможный. На практике используются также и другие способы. Например, помимо указанных выше полей могут храниться указатели на родительские узлы.

1.2.2 Деревья поиска

Деревом поиска (***дихотомическим*** деревом) называют такое дерево, для любого узла t которого:

- значения ключевых полей всех узлов левого поддерева узла t меньше, чем у узла t,
- значения ключевых полей всех узлов правого поддерева узла t больше, чем у узла t,

Отметим, что ключевые поля узлов дерева поиска, распечатанные в симметричном порядке, образуют возрастающую последовательность.

Дерево поиска позволяет быстро находить узел дерева с заданным значением ключа. Поиск в дереве поиска начинают с корня дерева и на каждом шаге сравнивают искомое значение с ключевым полем рассматриваемого узла. В том случае, если значения совпадают, узел найден. В том случае, если искомое значение меньше значения ключевого поля, то поиск продолжают в том же порядке в левом поддереве узла, иначе – в правом поддереве. В том случае, если при поиске осуществляется переход к пустому поддереву, констатируют, что значение не найдено.

В тех случаях, когда к дереву поиска не предъявляется требований по сбалансированности, добавляемый в дерево поиска узел помещается на место того пустого поддерева, к которому привел поиск узла (поиск по ключевому значению добавляемого узла).

Удаление узла из дерева поиска осуществляется следующим образом. Если удаляемый узел является листом дерева, то удаляемый узел замещается пустым поддеревом. Если удаляемый узел имеет одного потомка, то удаляемый узел замещается своим потомком. Если удаляемый узел имеет двух потомков, то в удаляемый узел помещаются данные из узла, непосредственно предшествующего удаляемому узлу по значению ключевого поля (самый правый узел левого поддерева), либо данные из узла, непосредственно следующего за удаляемым узлом по значению ключевого поля (самый левый узел правого поддерева). После этого узел, из которого пересылались данные, удаляется. Удаление такого узла выполняется несложно, так как такой узел будет иметь не более одного потомка.

1.2.3 Сбалансированные деревья

Бинарное дерево называется ***идеально сбалансированным***, если для каждого его узла количество узлов в левом и правом поддеревьях отличается не более чем на единицу. Идеально сбалансированное дерево из n узлов имеет высоту $[\log_2 n] + 1$. Такая высота является минимальной для бинарных деревьев, состоящих из n узлов.

Алгоритм построения сбалансированного дерева из n узлов является рекурсивным и состоит из следующих шагов:

1. Взять один узел в качестве корня.
2. Построить по данному алгоритму левое поддерево с числом узлов $n_л = n \text{ div } 2$.
3. Построить по данному алгоритму правое поддерево с числом узлов $n_п = n - n \text{ div } 2 - 1$

Поиск в идеально сбалансированном дереве осуществляется за логарифмическое время. Построение идеально сбалансированного дерева поиска для упорядоченной последовательности значений не представляет большой трудности, однако поддержание сбалансированности при удалении и добавлении узлов весьма трудоемко. По этой причине для построения сбалансированных деревьев поиска используют иную формулировку сбалансированности.

Дерево называют ***сбалансированным*** тогда и только тогда, когда высоты двух поддеревьев каждой из его вершин отличается не более чем на единицу. Такие деревья называют также ***АВЛ-деревьями*** по первым буквам имен их создателей - советских ученых Г.М. Адельсона-Вельского и Е.М. Ландиса. В АВЛ деревьях для каждого узла дерева вводят показатель сбалансированности и выполняют балансировку (так называемые вращения дерева) в том случае, если показатель сбалансированности нарушается.

2 Пример выполнения лабораторной работы

Написать программу, которая вводит с клавиатуры список целых чисел, считает **длину списка**, затем, из элементов списка создает **сбалансированное** дерево и подсчитывает **среднее арифметическое** элементов дерева.

```
struct CList{
     int info;
     CList* pnext;
};
typedef CList* pCList;

struct CTree{
     int info;
     CTree* pleft, *pright;
};
typedef CTree* pCTree;

void AddToList(pCList &head, int info){
     CList * pnew = new CList();
     pnew->info = info;
     pnew->pnext = head;
     head = pnew;
}
int ListCount(pCList head){
     int res = 0;
     CList * p = head;
     while (p!=NULL){
          p = p->pnext;
          res++;
     }
     return res;
}

void DisposeList(pCList &head){
     pCList p;
```

```
     while(head != NULL){
          p = head->pnext;
          delete head;
          head = p;
     }
}
void CreateBalansedTreeByListRecourse(pCList &headCopy, pCTree
&root, int listCount){
     if (listCount==0 || headCopy==NULL){
          root = NULL;
          return;
     }
     if (listCount<0) listCount = ListCount(headCopy);
     root = new CTree;
     root->info = headCopy->info;
     headCopy = headCopy->pnext;
     CreateBalansedTreeByListRecourse(headCopy, root->pleft,
listCount / 2);
     CreateBalansedTreeByListRecourse(headCopy, root->pright,
listCount - (listCount / 2) - 1);
}
void CreateBalansedTreeByList(pCList head, pCTree &root){
     if (head==NULL){
          root = NULL;
          return;
     }
     pCList headCopy = head;
     CreateBalansedTreeByListRecourse(headCopy, root,
ListCount(headCopy));
}
void PrintTreeLKP(pCTree root, int deep){
     if(root==NULL) return;
     //L
     PrintTreeLKP(root->pleft, deep+1);
     //K
     for(int i=0; i<deep; i++)
```

```
          printf("  ");
     printf("%d\n",root->info);
     //P
     PrintTreeLKP(root->pright, deep+1);
}
void AvgTreeRecourse(pCTree root, double &sum, int &count){
     if(root==NULL) return;
     sum = sum + root->info;
     count++;
     AvgTreeRecourse(root->pleft, sum, count);
     AvgTreeRecourse(root->pright, sum, count);
}
double AvgTree(pCTree root){
     double sum = 0;
     int cnt = 0;
     AvgTreeRecourse(root, sum, cnt);
     if(cnt>0) sum = sum/cnt;
     return sum;
}
void DisposeTree(pCTree &root){
     if(root==NULL) return;
     DisposeTree(root->pleft);
     DisposeTree(root->pright);
     delete root;
     root = NULL;
}

int _tmain(int argc, _TCHAR* argv[])
{
     printf("Написать программу, которая вводит с клавиатуры список целых чисел,\n считает длину списка, затем, из элементов списка создает\n сбалансированное дерево и подсчитывает среднее арифметическое элементов дерева.\n");
     pCList list = NULL;
     int cnt, info;
     printf("Введите количество элементов списка ");
```

```
    scanf("%d", &cnt);
    for (int i=0; i<cnt; i++){
        printf("введите очередное значение элемента списка ");
        scanf("%d", &info);
        AddToList(list, info);
    }
    printf("Длина списка = %d\n\n", ListCount(list));

    pCTree treeRoot = NULL;
    CreateBalansedTreeByList(list, treeRoot);

    printf("Среднее арифметическое элементов дерева равно %lf \n",
AvgTree(treeRoot));
    printf("А вот и само дерево\n");
    PrintTreeLKP(treeRoot, 0);

    printf("Для выхода нажмите любую клавишу");
    char c;
    scanf("%c", &c);//считываем <Enter> от предыдущих вводов
    scanf("%c", &c);//собственно считывание символа

    DisposeList(list);
    DisposeTree(treeRoot);
    return 0;
}
```

3 Содержание отчета

Отчет по лабораторной работе должен содержать:

1. Титульный лист.
2. Задание на лабораторную работу.
3. Описание основных алгоритмов и структур данных, используемых в программе:
4. Описание интерфейса пользователя программы.
5. Контрольный пример и результаты тестирования.
6. Листинг программы.

4 Контрольные вопросы

1. Дайте определение списка
2. Какие списки Вы знаете? В чем состоят отличия между ними? Определите необходимые типы данных и переменные.
3. В чем особенности реализации операций вставки и удаления элементов в списках различного вида?
4. Дайте определения следующим понятиям: дерево, степень узла, степень дерева, высота дерева.
5. Дайте определение бинарному дереву. Определите необходимые типы данных и переменные.
6. Сформулируйте алгоритмы обхода бинарных деревьев и деревьев общего вида.
7. Дайте определение следующим понятиям: дерево поиска, идеально сбалансированное дерево, сбалансированное дерево.
8. Сформулируйте алгоритмы добавления, удаления и поиска узла в дереве поиска.
9. Сформулируйте алгоритм построения сбалансированного дерева.

5 Задания на лабораторную работу

5.1 Начальный уровень сложности

Общие требования: в начале программы вывести задание; в процессе работы выводить подсказки пользователю (что ему нужно ввести, чтобы продолжить выполнение программы). Основные алгоритмы, ввод/вывод списков реализовать в виде функций с необходимыми параметрами. После работы программы вся динамически выделенная память должна быть освобождена.

Варианты заданий:

1. Написать программу, которая вводит с клавиатуры список целых чисел, считает **длину списка**, затем, из элементов списка создает **сбалансированное** дерево и подсчитывает **среднее арифметическое** элементов дерева.
2. Написать программу, которая вводит с клавиатуры список целых чисел, подсчитывает **среднее арифметическое элементов списка**, затем вводит с клавиатуры **дерево поиска** и считает **количество** элементов дерева.
3. Написать программу, которая вводит с клавиатуры список вещественных чисел, подсчитывает **количество положительных и отрицательных** элементов списка, затем вводит с клавиатуры **сбалансированное дерево** и считает **сумму** элементов дерева.
4. Написать программу, которая вводит с клавиатуры список целых чисел, строит **копию списка**, затем вводит с клавиатуры **дерево поиска** и считает **количество положительных и отрицательных** элементов дерева.
5. Написать программу, которая вводит с клавиатуры список целых чисел, строит **инверсную копию списка**, затем вводит с клавиатуры **сбалансированное дерево** и считает **количество листьев** дерева.
6. Написать программу, которая вводит с клавиатуры список целых чисел, считает **суммы четных и нечетных** элементов списка, затем вводит с клавиатуры **дерево поиска** и считает **количество узлов, не являющихся листьями**.
7. Написать программу, которая вводит с клавиатуры список целых чисел, ищет **максимальный** элемент списка, затем вводит с клавиатуры **сбалансированное дерево** и считает **глубину** дерева.

5.2 Средний уровень сложности

Общие требования: в начале программы вывести задание; в процессе работы выводить подсказки пользователю (что ему нужно ввести, чтобы продолжить выполнение программы). Основные алгоритмы, ввод/вывод списков реализовать в виде функций с необходимыми

параметрами. После работы программы вся динамически выделенная память должна быть освобождена.

Взаимодействие с пользователем организовать в виде простого меню, обеспечивающего возможность переопределения исходных данных и завершение работы программы.

Варианты заданий:

8. Написать программу, обеспечивающую работу с **однонаправленным нециклическим списком**: добавление/удаление элементов в **голову**, просмотр списка, **инверсию списка**.

9. Написать программу, обеспечивающую работу с **однонаправленным циклическим списком**: добавление/удаление элементов в **хвост**, просмотр списка, **удаление элементов с позиции N по K** (не включая N и K).

10. Написать программу, обеспечивающую работу с **однонаправленным нециклическим упорядоченным списком**: добавление/удаление элементов, просмотр списка, **поиск** элемента в списке.

11. Написать программу, обеспечивающую работу с **двунаправленным нециклическим списком**: добавление/удаление элементов в **голову**, просмотр списка, реализовать **дублирование элементов** с заданным значением.

12. Написать программу, обеспечивающую работу с **двунаправленным нециклическим списком**: добавление/удаление элементов в произвольную допустимую **позицию**, введённую пользователем, просмотр списка, **инверсию** списка.

13. Написать программу, обеспечивающую работу с **двунаправленным циклическим списком**: добавление/удаление элементов в **хвост**, просмотр списка, **удаление** всех элементов **с заданным значением**.

14. Написать программу, обеспечивающую работу со **сбалансированным деревом**: **создание** нового дерева из N элементов, **добавление** элемента, **просмотр** дерева, **очистка** дерева, **подсчет глубины**.

15. Написать программу, обеспечивающую работу **с деревом поиска**: **просмотр** дерева, **очистка** дерева, **поиск** элемента по ключу, **создание копии дерева, проверку эквивалентности** текущего дерева и последней копии.

16. Написать программу, обеспечивающую работу **с деревом поиска**: **добавление/удаление** элемента по ключу с сохранением упорядоченности, **просмотр** дерева, **очистка** дерева, **поиск** элемента по ключу, **подсчет** листьев в дереве.

17. Написать программу, обеспечивающую работу со **сбалансированным деревом**: **создание** нового дерева из N элементов, **просмотр** дерева, **очистка** дерева, а также **копирование** информационных полей со значениями в диапазоне от N до K в

однонаправленный нециклический упорядоченный список, просмотр и очистку этого списка.

18. Написать программу, обеспечивающую работу **с деревом поиска**: **добавление** элемента, **просмотр** дерева, **очистка** дерева, **поиск** элемента по ключу, а также **копирование** в **однонаправленный циклический список** элемента с заданным ключом, просмотр и очистку этого списка.

19. Написать программу, обеспечивающую работу со **сбалансированным деревом**: **создание** нового дерева из N элементов, **добавление** элемента, **просмотр** дерева, **очистка** дерева, а также **создание** на основе дерева **2 списка**: в первом списке должны находиться элементы больше K, во втором все оставшиеся, обеспечить просмотр этих списков.

20. Написать программу, обеспечивающую работу **с деревом поиска**: **добавление** элемента, **просмотр** дерева, **очистка** дерева, **поиск** элемента по ключу, а также **копирование** в **двунаправленный циклический список** элементов с ключами в заданном пользователем диапазоне, просмотр и очистку этого списка.

5.3 Высокий уровень сложности

Общие требования: в начале программы вывести задание; в процессе работы выводить подсказки пользователю (что ему нужно ввести, чтобы продолжить выполнение программы). Основные алгоритмы, ввод/вывод списков реализовать в виде функций с необходимыми параметрами. После работы программы вся динамически выделенная память должна быть освобождена.

Взаимодействие с пользователем организовать в виде простого меню, обеспечивающего возможность переопределения исходных данных и завершение работы программы, предусмотреть контроль вводимых пользователем данных.

Варианты заданий:

21. Написать программу, реализующую в виде списка представление многочлена $\boldsymbol{P(x) = a_0+a_1x+\ldots+a_ix^i}$, где a_i –вещественные числа, i – целые положительные числа, причем, если $a_i=0$, то соответствующий элемент-слагаемое должен отсутствовать в списке. Пользователь должен иметь возможность произвольно добавлять элементы-слагаемые через меню. Реализовать функцию **вычисления значения** многочлена при заданном x.

22. Написать программу, реализующую в виде списка представление многочлена $\boldsymbol{P(x) = a_0+a_1x+\ldots+a_ix^i}$, где a_i –вещественные числа, i – целые положительные числа, причем, если $a_i=0$, что соответствующий элемент-слагаемое должен отсутствовать в списке. Пользователь должен иметь возможность произвольно добавлять элементы-слагаемые через меню. Реализовать процедуру **взятия производной** многочлена.

23. Написать программу, реализующую в виде списка представление многочлена $\boldsymbol{P(x) = a_0+a_1x+\ldots+a_ix^i}$, где a_i –вещественные числа, i – целые положительные числа, причем, если $a_i=0$, что соответствующий элемент-слагаемое должен отсутствовать в списке. Пользователь должен иметь возможность произвольно добавлять элементы-слагаемые через меню. Реализовать функцию **сложения двух** многочленов.

24. Написать программу, обеспечивающую редактирование трех списков слов: добавление, удаление, очистку, просмотр. Реализовать функцию **замены всех вхождений** второго списка в первый на копию третьего список, подсчитать количество замен.

25. Написать программу, преобразующую **арифметическое выражение**, допускающее скобочную запись, операции +, –, *, / и вещественные числа и переменную *x*, в бинарное дерево, структурно эквивалентное выражению. Реализовать функцию **вычисления выражения** по дереву для заданного *x*.

26. Написать программу, преобразующую **арифметическое выражение**, допускающее скобочную запись, операции +, –, *, / и вещественные числа и переменную *x*, в бинарное дерево, структурно эквивалентное выражению. Реализовать функцию **упрощения выражения,** путем выполнения соответствующих операций над листьями дерева.

27. Написать программу, преобразующую **логическое выражение**, допускающее скобочную запись, операции &&, ||, ! и логические константы и переменную *x*, в бинарное дерево, структурно эквивалентное выражению. Реализовать функцию **вычисления истинности выражения** по дереву для заданного *x*.

28. Написать программу, преобразующую **логическое выражение**, допускающее скобочную запись, операции &&, ||, ! и логические константы и переменную *x*, в бинарное дерево, структурно эквивалентное выражению. Реализовать функцию **упрощения выражения,** путем выполнения соответствующих операций над листьями дерева.

6 Библиографический список

1. Страуструп Б. Язык программирования C++. Специальное издание. М.: Радио и связь, 1991. - 349с.

2. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. - М.: Мир, 1989.

3. Язык программирования Си / Б. Керниган, Д. Ритчи. - 3-е изд., испр. - СПб. : Невский Диалект, 2004. - 351 с.

4. Практикум по программированию на языке СИ / В. В. Подбельский. - М. : Финансы и статистика, 2004.

7 Приложения

7.1 Функции для работы с динамической памятью

Библиотечные функции для работы с динамической памятью определены в стандартном заголовочном файле <***stdlib.h***>. Ниже приводятся заголовки функций с кратким описанием.

```
void *malloc( size_t size );
```

Функция выделяет область памяти размером size байт. В случае если память была успешно выделена, возвращается указатель на первый байт памяти, в противном случае - нулевой указатель. Содержимое выделенного блока памяти никак не инициализируется (не определено).

```
void *calloc( size_t num,  size_t size );
```

Функция выделяет память под массив из num элементов, каждый из которых имеет размер size и инициализирует массив нулевыми значениями. Функция возвращает указатель на первый элемент массива в случае, если функция отработала успешно. В противном случае возвращается нулевой указатель.

```
void *realloc(  void *memblock,  size_t size );
```

Функция меняет размер ранее выделенной области памяти, на которую указывает memblock на новый размер size. Функция возвращает указатель на возможно новую область памяти. Функция копирует содержимое в новую область памяти, при необходимости усекая его. Дополнительный объем памяти при этом никак не инициализируется. В случае когда memblock равен NULL, функция выделяет память подобно malloc. В случае когда size равен 0, функция освобождает ранее выделенную память, подобно функции free.

```
void free( void *memblock );
```

Функция освобождает ранее выделенную с использованием malloc, calloc или realloc область памяти на которую указывает memblock.

Пример выделения памяти под массив из 100 вещественных чисел двойной точности, заполнения массива индексами элементов и освобождения памяти:

```
double* buf = (double*)malloc(100*sizeof(double));
if (buf)  {
   for (int i = 0, double *p=buf; i < 100; i++, p++) *p=i;
   free(buf);
}
```

7.2 Операции new и delete

Альтернативным способом работы с динамической памятью является использование операций new и delete. Операции new и delete имеют операторы имеют различный синтаксис в случае выделения памяти под одиночные объекты и под массивы объектов. В случае, когда требуется выделить и освободить память под одиночный объект, операции имеют следующий вид:

```
<имя> = new <тип>[(< инициализатор>)];
delete <имя>;
```

Здесь <имя> - название указателя на создаваемый объект, <тип> - тип элементов объекта в массиве. Отметим, что при использовании операции new имеется возможность указать инициализатор в стиле вызова функции (в круглых скобках). Операции new и delete используются при работе с объектами классов. Отметим, что вызываемая таким образом операция new в случае ошибки (например, при нехватке памяти) сгенерирует исключение.

Пример выделения и освобождения памяти под объекты базовых типов:

```
double* pdbl = new double;
char * pch = new char('A');
delete pdbl;
delete pch;
```

В случае, когда требуется выделить или освободить память под одномерный массив объектов операции имеют следующий вид:

```
<имя> = new <тип> [<выражение>];
delete [] < имя >;
```

Размер динамического массива, определяемый во время выполнения программы в соответствии с выражением <выражение>, показывает количество элементов создаваемого массива. Для оператора new размер обязательно указывается в квадратных скобках, для оператора delete скобки остаются пустыми.

Пример выделения памяти под массив из 100 вещественных чисел двойной точности, заполнения массива индексами элементов и освобождения памяти:

```
int size = 100;
double* ar = new double[size];
for (int i = 0; i < 100; i++) ar[i] = i;
delete [] ar;
```

Память, отведенную с использованием операции new, следует освобождать с использованием операции delete, и, наоборот, память, выделенную с использованием функций malloc, calloc, realloc следует освобождать с использованием функции free. Кроме того, если

выделение массива производилось с использованием операции new[], то освобождать его следует с использованием операции delete[] с указанием квадратных скобок. Освобождать такой массив через delete без указания квадратных скобок является ошибкой.