СТРУКТУРЫ ДАННЫХ И АЛГОРИТМЫ

Методические указания
к лабораторным работам для студентов
1 курса ФПМиИ
(направление 010400 – прикладная математика и информатика, направление 010500–математическое обеспечение и администрирование информационных систем)
дневного отделения

Новосибирск 2013

ВВЕДЕНИЕ

Цели лабораторных работ:

- формирование практических навыков организации и использования при решении задач динамических структур данных;
- изучение наиболее распространенных алгоритмов решения задач с использованием сложных структур данных.

Содержание работ одинаково для всех лабораторных работ.

Язык программирования – язык Си/Си++.

Порядок выполнения работы

1. Изучить необходимый материал по теме лабораторной работы, руководствуясь методическими указаниями к теме.
2. Получить допуск к работе, ответив на контрольные вопросы.
3. Разработать структуры данных для представления основных исходных данных и/или результатов для выданного преподавателем варианта задания.
4. Разработать алгоритм и написать программу решения задачи.
5. Подготовить набор тестов и отладить программу.
6. Оформить отчет по лабораторной работе. Отчет по работе включает следующие пункты:
    1. Условие задачи
    2. Анализ задачи
    3. Структуры основных входных и выходных данных
    4. Алгоритм решения задачи
    5. Текст программы
    6. Набор тестов
    7. Результаты отладки и их анализ
7. Защитить лабораторную работу. Защита состоит в обсуждении выбранной структуры данных и алгоритма решения задачи, ответе на контрольные вопросы, решении контрольных задач.

Л А Б О Р А Т О Р Н А Я Р А Б О Т А № 1

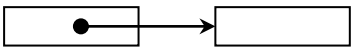
ЛИНЕЙНЫЕ СПИСКИ

Цель работы: изучить тип указатель; получить навыки в организации и обработке линейных списков.

1. Методические указания

1.1. Тип указатель

Значением типа указатель является адрес участка памяти, выделенного для объекта конкретного типа (ссылка на объект). Связь указателя P с объектом можно изобразить следующим образом:



ссылка объект

По указателю осуществляется обращение (доступ) к объекту.

Определение переменной типа указатель:

type *имя_указателя ;

где type – обозначение типа, на который будет указывать переменная с именем (идентификатором) имя_указателя. Символ ‘*’ - это унарная операция раскрытия ссылки (операция разыменования, операция обращения по адресу, операция доступа по адресу), но в данном контексте служит признаком типа указатель.

При определении переменной-указателя можно выполнить инициализацию:

type *имя_указателя инициализатор ;

Инициализатор имеет две формы записи, поэтому допустимо:

type *имя_указателя = инициализирующее_ выражение ;

type *имя_указателя (инициализирующее_ выражение) ;

Инициализирующее_ выражение – это константное выражение, которое может быть задано указателем, уже имеющим значение или выражением, позволяющим получить адрес объекта с помощью операции & - получение адреса операнда.

Пример 1:

```
char cc = ‘f’, *p;     //Определение символьной переменной cc  и неинициализи-
                         // рованного указателя на  объект типа char
int *p1, *p2;          //Определение двух неинициализированных указателей
                         // p1 и p2 на объекты типа int
char *pc = &cc;      //Инициализированный указатель на объект типа char
char *ptr (NULL);  //Нулевой указатель на объект типа char
int *mas [10];         //Массив из 10 указателей на объекты типа int
int (*point) [10];    //Указатель на массив из 10 элементов типа int
struct list *n, *pr;   //Два указателя n и pr на объекты именованного структурного
                         // типа list. Определение  типа list должно либо
                         // предшествовать данному определению, либо находиться в
                         // пределах области действия данного определения
  struct list            //Определение переменной line структурного
```

```
    {                    //типа list, один из элементов которой типа
    char b; int *k;      //указатель на объект типа int
            } line;
   int **pp = &p1;   //Указатель pp – это указатель на указатель. Он инициирован
                     //значением адреса указателя p1 на объект целого типа. Это
                     // допустимо, так как указатель – это тоже объект в памяти
```

Переменной типа указатель (в дальнейшем просто указатель) можно задать значение:

- присваивая ей адрес объекта с помощью операции & (тип объекта должен быть тот же, что и тип объекта, на который ссылается указатель);
- присваивая ей значение другой переменной или выражения типа указатель (типы объектов, на которые они ссылаются должны быть одинаковыми);
- с помощью операции new.

Над указателями допускается выполнение следующих операций:

- присваивание;
- получение адреса;
- сложение и вычитание;
- инкремент и декремент;
- операции отношения.

Все операции применимы к указателям на объекты одного типа и к указателю и целой константе, в противном случае требуется явное преобразование типов. Операция сложения допустима только для указателя и целочисленного значения. Вычитая два указателя одного типа, можно получить “расстояние” между двумя участками памяти. “Расстояние“ определяется в единицах, кратных размеру памяти объекта того типа, к которому отнесен указатель.

С помощью переменной типа указатель и операции разыменования * можно обращаться к объекту, на который она ссылается.

Пример 2 (с учетом описаний в примере 1) :

```
  int a = 10, b = 20, c = 30;
  int x[3][10];
ptr = pc;           //Указатели ptr и pc ссылаются на одну и ту же переменную
                    // cc типа char
*ptr = '+';         //Объект, на который указывают ptr и pc (т.е. cc), получил
                    // новое значение
p1 = &a;            //Указатель p1 получил значение адреса целой переменной a
p2 = &b;            //Указатель p2 – значение адреса целой переменной b
mas[0] = p1;        //Элемент массива mas[0] типа указатель получил то же значение,
                    //что и указатель p1
mas[1] = &b;        //Указатель mas[1] получил значение адреса b
*mas[0] = 15;       //Переменная типа int, на которую указывают и p1 и mas[0],
                    //т.е. переменная a получила новое значение
*p2 = 25;           //Переменная b получила новое значение
point = &x[0];      //Указатель point получил значение адреса нулевого элемента
                    //массива x, т.е. адреса элемента x[0][0]
point[0] = a;       //Элементу массива с индексом 0, на который указывает point
```

```
                    //присваивается значение  переменной a
*(point + 1) = b;  //Элементу массива, на который указывает point + 1,
                    //т.е. элементу x[0][1] присваивается значение b
n = &line;          //Указателю n присвоен адрес структуры line
pr = n;             //Указатели n и pr ссылаются на одну  переменную
(*n).b = *ptr;      //Элементу b структуры, на которую ссылается n, присваивается
                    //значение объекта, на который ссылается ptr,
                    //т.е. line.b получила значение '+'
pr -> k = p1;       //Элементу k структуры, на которую ссылается pr присваивается
                    //значение указателя p1, т.е. line.k получила значение адреса
                    //целой переменной a
```

1.2. Понятие динамической структуры данных

Описывая в программе с помощью определений структуру данных (переменную, константу, функцию), мы создаем так называемые статические объекты или заранее определяемые объекты, которые отображаются в статической памяти. Статическая структура данных характеризуется тем что:

- она имеет имя, которое используется для обращения к ней;
- ей выделяется память в процессе трансляции программы;
- количество элементов сложной структуры (размерность) фиксировано при ее описании;
- размерность структуры не может быть изменена во время выполнения программы.

Динамическая структура данных характеризуется тем что:

- количество элементов структуры может не фиксироваться;
- размерность структуры может меняться в процессе выполнения программы;
- в процессе выполнения программы может меняться характер взаимосвязи между элементами структуры.

Динамическая структура данных может отображаться как в статической, так и в динамической памяти. При отображении динамической структуры данных в статической памяти приходится вносить ограничение на размер данных (максимальное число элементов в значении структуры) и обеспечивать возможность изменения фактической размерности в процессе выполнения программы.

При отображении динамических данных в динамической памяти динамическая структура данных характеризуется еще и тем что:

- ей выделяется память в процессе выполнения программы;
- она не имеет имени.

Каждой такой динамической структуре данных сопоставляется статическая переменная типа указатель (ее значение - адрес этого объекта) посредством которой осуществляется доступ к динамической структуре. Для создания динамического объекта используется унарная операция new:

```
new  имя_типа
```

или

```
new  имя_типа  инициализатор
```

Операция new позволяет выделить свободный участок в основной памяти (динамической), размеры которого соответствуют типу данных, определяемому именем типа. В случае успешного выполнения операция new возвращает адрес начала выделенного участка памяти, таким образом, создан (порожден) динамический объект. Если участок нужного размера не может быть выделен (нет свободной памяти нужного размера), то операция new возвращает нулевое значение адреса (NULL). В случае инициализации в выделенный участок заносится значение, определяемое инициализатором, т.е. динамическому объекту присваивается начальное значение, иначе значение динамического объекта не определено.

Пример 3:

```
int *p1, *p2;
char *ptr (NULL);
int *mas [10];
int (*point) [10];
struct list *n, *pr;
  struct list
   {   char b; int *k;      }

p1 = new int;          //Создан динамический объект типа int, адрес
                       // которого присвоен указателю p1
*p1 = 25;              //Динамический объект, на который указывает p1
                       //получил значение 25
p2 = new int(15);      //Создана динамическая  переменная целого типа с начальным
                       //значением 15, указателю p2 присвоен ее адрес
mas[0] = new int;      //Создана динамическая переменная целого типа,
                       // указателю mas[0] присвоен ее адрес
mas[1] = p1;           //Указатели mas[1],  p1 ссылаются на один объект
ptr = new char('*'); //Создан динамический объект типа char, адрес которого
                       //присвоен указателю ptr, объект инициирован символьным
                       //значением '*'
point = new int[10]; //Создан динамический объект типа массив из 10 целых
                       //элементов, указателю point присвоен адрес первого элемента
                       //этого массива
n= new list;           //Создан динамический объект типа структура,
                       //указателю n присвоен его адрес
pr = n;                //Указатели pr и n ссылаются на объект типа list
n->b = *p1;            //Элементу с именем b динамической структуры, на которую
                       //указывает n присвоено значение динамического объекта,
                       //на который указывает p1, т.е. значение 25
n->k = p2;             //Элементу с именем k динамической структуры, на которую
                       //указывает n присвоено значение указателя p2, т.е. адрес
                       //динамического объекта целого типа со значением 15
*(n->k) ++;            //Значение динамического объекта целого типа, на который
                       //ссылается элемент с именем k динамической структуры,
```

```
                    //на которую ссылается указатель n, увеличено на 1, т.е. стало
                    //равно 16 (на этот объект ссылается также указатель p2)
```

Продолжительность существования динамического объекта – от точки его создания операцией new до конца программы или до явного его уничтожения.

Уничтожение динамического объекта (освобождение памяти, выделенной под динамическую переменную) осуществляется операцией delete:

```
delete  имя_указателя ;
```

Здесь указатель адресует освобождаемый участок памяти, ранее выделенный с помощью операции new. После выполнения операции delete значение указателя становится неопределенным (хотя в некоторых реализациях языка может и не меняться).

Для освобождения памяти, выделенной для массива, используется следующая модификация этой операции:

```
delete  [ ] имя_указателя ;
```

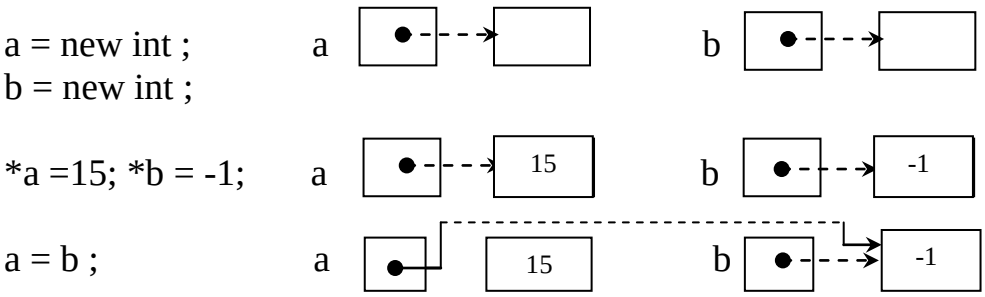
Уничтожая динамический объект не оставляйте “ висячих ” ссылок. “ Висячая ” ссылка - это ссылка на несуществующий объект.

Над динамическими объектами выполняются операции допустимые для типа данного динамического объекта.

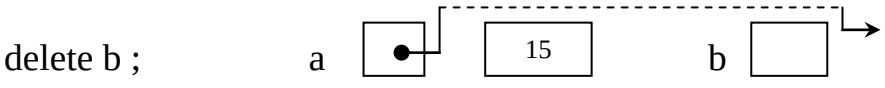
Пример 4:

```
int  *a, *b  ;
   . . . . . .
  a =  new int ;  b = new int ;
  *a = 15 ; *b = -1 ;  a = b ;  delete b  ;
    . . . . . .
```

Результаты этих действий можно продемонстрировать следующим образом:



Динамический объект со значением 15 существует, но к нему нет доступа. Этот недоступный объект называется “мусором”. Создавая мусор, вы уменьшаете объем свободной динамической памяти. Указатели a и b ссылаются на один объект.



Память, занимаемая динамическим объектом, на который указывают a и b освобождена, но указатель a сохранил значение адреса объекта. Указатель a – “висячая” ссылка.

1.3. Линейный однонаправленный (односвязный) список

Список – это множество (возможно пустое) данных, имеющих некоторый общий смысл для решаемой задачи.

В статической памяти список можно представить как множество данных, которые располагаются в списке последовательно и непрерывно друг за другом. Будем называть такой список (условно) – статический список. На языке Си статический список можно представить неоднородной структурой:

```
struct  list
        { int  n;
          type  elem [ k ];  }
```

Здесь элемент с именем n определяет фактическое количество данных (элементов) в списке, если n равно 0, то список пуст, если n равно k, то список полон; элемент с именем elem (в данном случае массив) определяет само множество элементов списка, type – тип элемента списка.

В динамической памяти список можно представить как множество данных, связанных между собой указателями. Будем называть такой список (условно) – динамический список.

В линейном списке у каждого, составляющего его данного (элемента списка) есть один предшествующий и один следующий элемент (это справедливо для всех элементов кроме первого и последнего).

В данной лабораторной работе рассматриваются списки, представляемые в динамической памяти.

Линейный односвязный список – это динамический список, каждый элемент которого состоит из двух полей. Одно поле содержит информацию (или ссылку на нее), другое поле содержит ссылку на следующий элемент списка. Элемент списка называют «звено» списка. Таким образом, список – это цепочка связанных между собой звеньев от первого до последнего.

Пример 5 (строку символов BETA представим в виде списка):

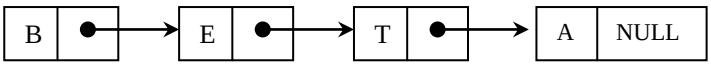


Рис.1.1.

Последнее звено (рис.1.1.) не ссылается на следующий элемент, поэтому поле ссылки имеет значение NULL - пустой указатель.

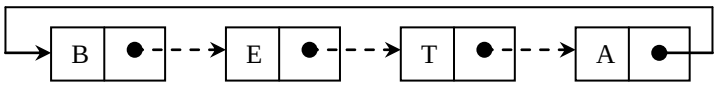
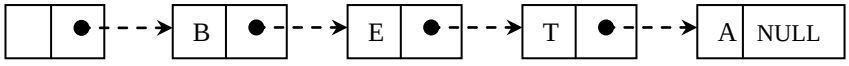


Рис.1.2.

Последнее звено ссылается на первый элемент списка (рис.1.2.) - это циклический список.



Заглавное звено

Рис.1.3.

Список имеет заглавное звено (рис.1.3.), которое ссылается на первый элемент списка. Его информационное поле, как правило, не используется. Заглавное звено позволяет обрабатывать первое звено списка также как и другие его звенья.

По однонаправленному списку можно двигаться только в одном направлении – от первого (или заглавного) звена к последнему.

Чтобы задать динамический список надо описать его звено. Так как звено состоит из полей разных типов, то описать его можно неоднородным типом – структурой.

Задание типа элемента списка:

```
struct list
    { list  *next ;
      type  elem ; }
```

Здесь type – тип информационного поля элемента списка, поле next – ссылка на аналогичную структуру типа list.

Для примера 5 элемент списка может быть определен:

```
struct list
    { list  *next ;
      char  elem ; }
```

Чтобы работать со списком как с единым объектом, надо ввести в употребление статическую переменную-указатель, значение которой – это адрес первого (или заглавного) звена списка. Если список пустой, она должна иметь значение NULL.

Определение статической переменной-указатель:

```
list  *headlist ;
```

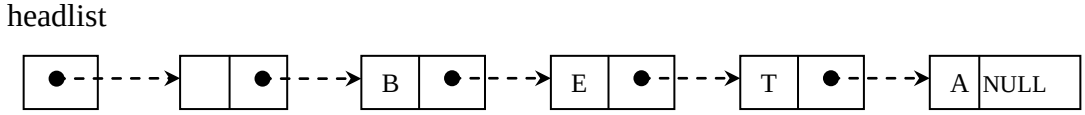


Рис.1.4.

Статическая переменная-указатель headlist, называемая заголовком (или головой) списка (не путать с заглавным звеном) определяет список как единый объект. Используя указатель, можно обращаться к элементам списка (для списка на рис.1.4.):

```
headlist . . .                    //указатель на заглавное звено списка
headlist->next . . .              //указатель на первое звено списка
headlist->next->next . . .        //указатель на второе звено списка
headlist->next->next->next . . .  //указатель на третье звено списка
headlist->next->elem . . .        //обращение к информационному
                                  //полю первого элемента списка
```

Для более удобной работы со списком лучше ввести указатель, который ссылается на текущий элемент списка.

Пример 6. Формирование списка из примера 5 (не циклического, с заглавным звеном):

```
void main ( )
```

```
        struct list
           { list  *next ;
             char  elem ; } *ph;
        list *p;
        char ch;
 { ph = new list;       //Создание заглавного звена, ph получил значение адреса
   ph->next = NULL; //заглавного звена, его полю next присвоено значение
                          //пустой ссылки, таким образом создан пустой список
                          //с заглавным звеном
   p = ph;                //текущему указателю p присвоена ссылка на заглавное звено
   while ( (ch = getchar ( )) != '\n' )
     { p->next = new list;   //Создание очередного звена, поле next текущего звена
                             //получило значение адреса вновь созданного звена
       p = p->next;          //текущему указателю p присвоена ссылка на
                             //очередное звено
       p->elem = ch; }       //информационное поле elem текущего звена
                             //получило значение символа ch
   p->next = NULL; }     //Ссылочному полю на следующий элемент списка текущего
                         //(последнего сформированного) звена присвоено значение
                         //пустой ссылки, таким образом определен конец списка
```

Пример 7. Фрагменты программ, выполняющих различные действия с односвязным списком:

```
        struct list
           { list  *next ;
             char  elem ; } *ph;      //Считаем, что ph – голова списка
        list *p, pr;
      . . . . .
for (p = ph; p != NULL; p = p->next) . . .       //просмотр всех элементов списка
for (p = ph, pr = NULL; p != NULL; pr = p, p = p->next) . . .
                          //просмотр списка с сохранением указателя
                          //на предыдущий элемент списка
for (p = ph; p->next != NULL; p = p->next) . . .
                           //переход к последнему элементу непустого списка
if (ph != NULL)
          for (p = ph; p->next != NULL; p = p->next) . . .
                          //просмотр списка (возможно пустого)  с
                          //сохранением указателя на предыдущий элемент
```

Основными операциями обработки списка являются:

1) поиск заданного элемента по его значению или порядковому номеру; операция заканчивается, когда элемент найден или просмотрен весь список (если элемента нет); результат операции должен определять, есть ли элемент в списке или нет и, если есть, то возможно его адрес или значение;

2) включение (вставка) в список нового элемента перед или после заданного

элемента (в том числе перед первым элементом или после последнего); включению, как правило, предшествует поиск элемента после и/или перед которым происходит включение; при включении элемента перед первым в список без заглавного звена меняется заголовок списка; при включении после некоторого элемента меняется ссылочное поле у элемента, после которого происходит включение, поэтому надо определять ссылку на элемент после которого происходит включение;

3) исключение (удаление) заданного элемента из списка (в том числе удаление первого элемента или последнего); исключению, как правило, предшествует поиск, исключаемого элемента; результатом поиска должна быть ссылка на элемент предшествующий исключаемому, так как при удалении элемента из списка меняется ссылочное поле у элемента, предшествующего удаляемому; при удалении первого элемента в списке без заглавного звена меняется заголовок списка;

4) определение числа звеньев списка;

5) упорядочение элементов списка по значению информационного поля.

1.4. Линейный двунаправленный (двусвязный) список

Динамический список, в котором каждый элемент (кроме возможно первого и последнего) связан с предыдущим и следующим за ним элементами называется двусвязным. Каждый элемент такого списка имеет два поля с указателями: одно поле содержит ссылку на следующий элемент, другое поле – ссылку на предыдущий и третье поле - информационное. Наличие ссылок на следующее звено и на предыдущее позволяет двигаться по списку от каждого звена в любом направлении: от звена к концу списка или от звена к началу списка, поэтому такой список называют еще и двунаправленным.

Пример 1 (строка символов BETA представлена двусвязным списком):

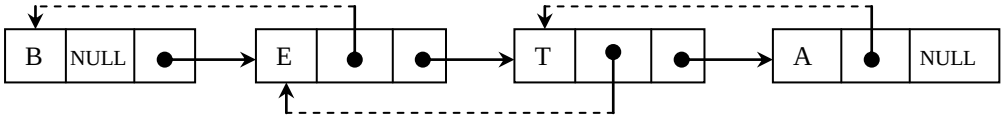


Рис.1.5.

Первое звено не имеет ссылки на предыдущее, последнее звено не имеет ссылки на следующее звено (рис.1.5.).

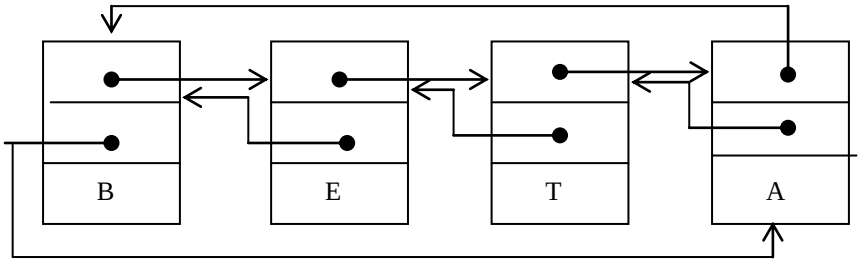


Рис.1.6.

Такой список (рис.1.6.) называют кольцевым (циклическим). Здесь первое звено имеет ссылку на последнее, а последнее звено на первое. Поэтому начинать обработку такого списка можно как с первого звена так и с последнего.

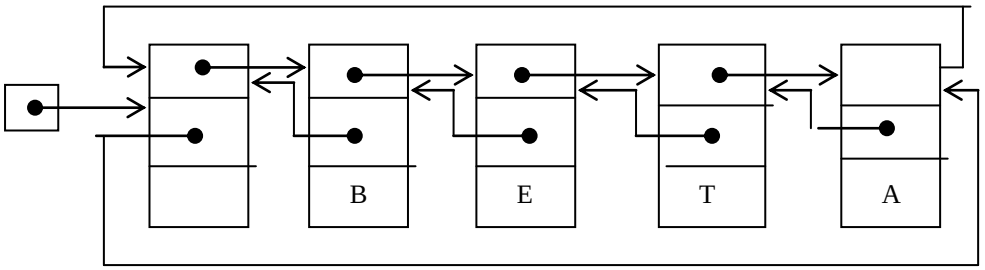


Рис.1.7.

Двусвязный список может иметь заглавное звено (рис.1.7. и рис.1.8.). Заглавное звено позволяет обрабатывать первое и последнее звенья также как другие.

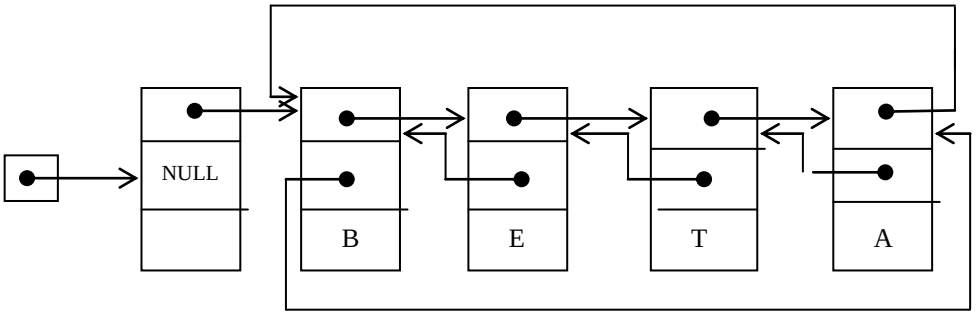


Рис.1.8.

Возможны и другие структуры двусвязных списков.

Задание двусвязного списка:

```
struct list2
        { type   elem ;
          list2   * next, * pred ;
                    }

list2   * headlist2 ;
```

Здесь type – тип информационного поля элемента списка.
Переменная-указатель headlist2 задает список как единый программный объект, ее значение - указатель на первый (или заглавный) элемент списка.

Пример 2. Формирование двунаправленного кольцевого списка

```
void  main ( )
struct list2
      { char  elem ;
        list2  * next, * pred ; }
list2 *l, *r, *x;
char sym;
```

```
{ l = new list2;                          // Создание пустого
  l -> next = l;                          //  кольцевого списка
  l -> pred = l;                          //  с заглавным  звеном
  r = l;
    while ( (sym = getchar ( )) != '\n' )
  {   x = new list2; x -> elem = sym;          // Создание очередного звена
  x -> next = r -> next;                       // Поле next - указатель на следующее звено
                                          //получило значение указателя на первое звено списка
  x -> pred = r;                          //Поле pred - указатель на предыдущее звено получило
                                        // значение указателя на текущее последнее звено списка
  r -> next = x;                          // Поле next текущего последнего элемента - указатель
                                       // на следующий элемент получило значение указателя на
                                       // новый элемент, таким образом, новое звено добавлено
                                       //  в конец списка
  x -> next -> pred = x;                       // Поле pred заглавного элемента - указатель
                                        // на последнее звено списка получило значение указателя
                                       // на добавленный в конец списка элемент, таким образом,
                                      // сформирован кольцевой список
   r = r -> next;                         // Текущий указатель получил значение ссылки на
                                        // новый последний элемент списка
            } }
```

Основные операции, выполняемые над двусвязным списком, те же, что и для односвязного списка. Так как двусвязный список более гибкий, чем односвязный, то при включении элемента в список, можно использовать указатель как на элемент, за которым происходит включение, так и указатель на элемент перед которым происходит включение. При исключении элемента из списка можно использовать как указатель на сам исключаемый элемент, так и указатель на элемент предшествующий или следующий за исключаемым. Но так как элемент двусвязного списка имеет два указателя, то при выполнении операций включения/исключения элемента надо изменять больше связей, чем в односвязном списке. Поиск элемента в двусвязном списке можно вести:

а) просматривая элементы от начала к концу списка,

б) просматривая элементы от конца списка к началу,

в) просматривая список в обоих направлениях одновременно: от начала к середине списка и от конца к середине (учитывая, что элементов в списке может быть четное или нечетное количество).

Пример 3. Фрагменты программ, выполняющих различные действия с двусвязным списком:

```
 void  exam1( list2 *p)  // Просмотр циклического (возможно пустого)
                         // списка без заглавного звена
 { list2 *ph;            // в направлении от начала к концу списка
   if ( p = = NULL ) return;
   ph = p;
  do  { . . .
  p = p -> next; }
```

```
  while ( p != ph ); //Просмотр продолжается пока текущий указатель
                     //  p не равен указателю на начало списка - ph
       }
. . .
void  exam2( list2 *p)  // Просмотр кольцевого (возможно пустого)
                        // списка с заглавным звеном
 { list2 *pr;           // в направлении от конца списка к началу
   if ( p –> next = = p ) return;
   pr = p -> pred;      // Текущий указатель pr получил значение ссылки
                        // на последний элемент списка
  while (pr != p)  //Просмотр продолжается пока текущий указатель pr
                   // не равен указателю на заглавное звено списка - p
{ . . . pr = pr -> pred; }
. . .
x = new list2;          // Включение нового элемента (в список с
x -> pred = p -> pred; // заглавным звеном) перед  элементом, на
x -> next = p;          // который ссылается p
p -> pred -> next = x;
p -> pred = x;
. . .
p -> pred -> next = p -> next;  // Исключение из списка с заглавным
p -> next -> pred = p -> pred;  // звеном элемента, на который
delete p;                       // ссылается указатель p
```

2. Контрольные вопросы

1. Понятие типа указатель.
2. Задание переменных типа указатель. Операции над указателями.
3. Понятие статического и динамического объекта.
4. Создание и уничтожение динамического объекта. Операции над динамическим объектом.
5. Понятие списка.
6. Понятие линейного односвязного списка. Задание односвязного списка.
7. Операции над односвязным списком.
8. Понятие двусвязного списка. Возможные структуры двусвязного списка.
9. Задание двусвязного списка.
10. Основные операции над двусвязным списком.

3. Варианты задания

3.1. Односвязный список

1. Задан текст, состоящий из строк, разделенных пробелом и оканчивающийся точкой.

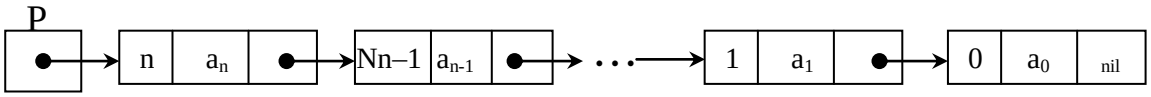
   Написать подпрограмму поиска заданного элемента в списке. Используя эту подпрограмму :

а) подсчитать количество вхождений заданного символа в каждую строку текста. Вхождение задавать номером строки и номером позиции в строке;
б) найти все вхождения (см. пункт 1.а) заданного символа в текст;
в) найти первое вхождение (см. пункт 1.а) каждой десятичной цифры в текст;
г) найти первое вхождение (см. пункт 1.а) гласных латинских букв в текст;
д) подсчитать количество вхождений четных (нечетных) десятичных цифр в каждую строку текста;
е) заменить заданный символ, если он имеется в тексте, на новое значение (символ), считая, что символ входит в каждую строку не более одного раза;
ж) удалить все вхождения заданного символа из текста;
з) после последнего вхождения каждой гласной латинской буквы в строку текста вставить цифру, изображающую число вхождений этой гласной в данную строку (в строке содержится не более девяти одинаковых гласных);
и) если в строке текста содержится заданный символ, то переместить его на место первого символа в этой строке;
к) если в строке текста содержится заданный символ, то переместить его на место последнего символа в этой строке.

2. Даны действительные числа $x_1, x_2, \ldots, x_n$ (n >= 2 и заранее неизвестно). Получить последовательность ($x_1 - x_n$) , ($x_2 - x_n$) , . . . , ($x_{n-1} - x_n$).

3. Дана последовательность действительных чисел $a_1, a_2, \ldots, a_n$ (n >= 2 и заранее неизвестно). Если последовательность упорядочена по неубыванию, то оставить ее без изменения, иначе получить последовательность $a_n, a_{n-1}, \ldots, a_1$.

4. Для заданных полиномов $P_n(x)$ и $Q_n(x)$ найти полином R - сумму полиномов P и Q. Каждый полином представить в виде списка:



Причем в список а) включать, б) не включать и коэффициенты равные нулю. Считать, что входные данные не содержат равных нулю коэффициентов.

5. Дана последовательность символов $s_1, s_2, \ldots, s_n$ (n >= 2 и заранее неизвестно). Получить те символы, принадлежащие последовательности, которые входят в нее по одному разу.

6. Дана последовательность символов $s_1, s_2, \ldots, s_n$ (n >= 2 и заранее неизвестно). Получить последовательность символов, содержащую только последние вхождения каждого символа в строку с сохранением их исходного взаимного порядка.

7. Дана последовательность символов $s_1, s_2, \ldots$. Известно, что s_1 отличен от точки и , что среди $s_2, s_3, \ldots$ имеется хотя бы одна точка. Пусть $s_1, s_2, \ldots, s_n$ - символы, предшествующие первой точке. Получить последовательность $s_1, s_3, \ldots, s_n$, если n нечетно и последовательность $s_2, s_4, \ldots, s_n$, если n четно.

3.2. Двусвязный список

1. Дана последовательность символов, оканчивающаяся точкой:
а) найти всех соседей заданного символа (первый и

последний символы считать соседями);

б) подсчитать количество символов, у которых левый сосед больше правого соседа (первый и последний элемент считать соседями);

в) удалить все символы, у которых равные соседи (первый и последний символы считать соседями);

г) переставить в обратном порядке все символы между первым и последним вхождениями заданного символа;

д) в конец последовательности добавить все ее символы, располагая их в обратном порядке (например, из последовательности 1, 2, 3 получить 1, 2, 3, 2, 1).

2. Дана последовательность латинских букв, оканчивающаяся точкой. Среди букв есть специальный символ, появление которого означает отмену предыдущей буквы; k знаков подряд отменяют k предыдущих букв, если такие есть. Преобразовать последовательность с учетом вхождения в нее специального символа.

3. Даны две разреженные квадратные матрицы A и B порядка n (разреженная матрица это матрица высокого порядка с большим количеством нулевых элементов). Получить матрицу C = A*B. Для представления разреженной матрицы использовать двусвязный циклический список. Каждое звено списка состоит из пяти полей :

- поле с номером строки ненулевого элемента,
- поле с номером столбца ненулевого элемента,
- поле со значением элемента,
- поле со ссылкой на предыдущий ненулевой элемент в этой же строке,
- поле со ссылкой на предыдущий ненулевой элемент в этом же столбце.

Каждая строка (и столбец) имеют заглавное звено, соответствующее ссылочное поле которого содержит ссылку на последний ненулевой элемент в строке (в столбце) .

4. Дан многочлен P(x) произвольной степени с целыми коэффициентами, причем его одночлены могут быть не упорядочены по степеням x , а одночлены с одинаковой степенью могут повторяться (например, $-75x + 8x^4 - x^2 + 6x^4 - 5 - x$). Привести подобные члены в этом многочлене и расположить одночлены по убыванию степеней x .

5. Даны действительные числа $x_1 , x_2 , \ldots , x_n$ (n >= 2 и заранее неизвестно). Вычислить:

а) $x_1 x_n + x_2 x_{n-1} + \ldots + x_n x_1$;

б) $(x_1 + x_n) (x_2 + x_{n-1}) \ldots (x_n + x_1)$;

в) $(x_1 + x_2 + 2x_n) (x_2 + x_3 + 2x_{n-1}) \ldots (x_{n-1} + x_n + 2x_2)$.

6. Даны действительные числа $y_1 , y_2 , \ldots , y_n$ (n >= 2 и заранее неизвестно). Получить последовательность:

а) $y_1 , y_2 , \ldots , y_n , y_1 , \ldots , y_n$;

б) $y_1 , y_2 , \ldots , y_n , y_n , \ldots , y_1$;

в) $y_n , y_{n-1} , \ldots , y_1 , y_1 , \ldots , y_n$.

7. Даны действительные числа $a_1, a_2, \ldots, a_{2n}$ (n >= 2 и заранее неизвестно). Вычислить:

а) $a_1 a_{2n} + a_2 a_{2n-1} + \ldots + a_n a_{n+1}$;

б) $\min (a_1 + a_{n+1}, a_2 + a_{n+2}, \ldots, a_n + a_{2n})$;

в) $\max (\min (a_1, a_{2n}), \min (a_3, a_{2n-2}), \ldots, \min (a_{2n-1}, a_2))$.

Л А Б О Р А Т О Р Н А Я Р А Б О Т А № 2

СТРУКТУРЫ ДАННЫХ СТЕКИ и ОЧЕРЕДИ

Цель работы: сформировать практические навыки организации таких распространенных структур как стеки и очереди и их использования при решении задач.

1. Методические указания

Стек – это список, у которого доступен один элемент (одна позиция). Этот элемент называется вершиной стека. Взять элемент можно только из вершины стека, добавить элемент можно только в вершину стека.

Очередь – это список, у которого доступны два элемента (две позиции) начало и конец очереди. Поместить элемент можно только в конец очереди, а взять элемент только из ее начала.

Стеки и очереди могут быть организованы различными способами. При этом они могут быть размещены как в статической памяти, так и в динамической.

Статический стек (стек отображается в статической памяти) можно задать следующим образом:

```
struct  stack
      { int   top;
        type  data  [ n ]  ;}
```

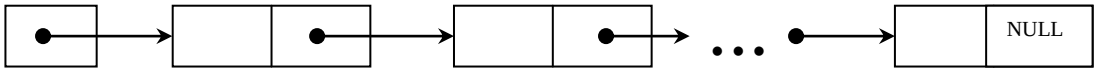
Здесь type – тип элементов, хранимых в стеке. Поле top определяет вершину стека. Ограничение на количество элементов в массиве (константа n) задает размер стека.

вершина

Вершина стека это либо позиция стека (массива), куда был помещен последний элемент либо позиция, куда будет помещен очередной элемент. Чтобы инициализировать статический стек (задать начальное значение стека), достаточно определить такое начальное значение поля top, которое показывало бы, что в стеке еще нет элементов, т.е. стек пуст. Если S.top = -1 (или 0 в случае, когда top показывает на позицию, куда будет помещен очередной элемент), то стек пуст. В таком случае при помещении в стек элемента значение S.top увеличивается на единицу, при взятии элемента уменьшается на единицу. Если top равно n-1 (или n), то стек полон. Если указатель равен 0 (или 1), то в стеке один элемент. При взятии из стека единственного элемента, он становиться пустым, поле top должно получить значение -1 (или 0).Обращение к элементу в вершине статического стека: S.data [S.top].

Динамический стек (стек отображается в динамической памяти) можно задать линейным односвязным списком:

```
struct list
   { type  pole1;
     list *pole2 ; }
typedef    list      stack ;
```



вершина стека

В стеке, представленном линейным односвязным списком, вершина стека – первый элемент списка.

Ввести в употребление переменную типа стек можно следующим образом:

```
stack *S;
```

Переменная S – это указатель на вершину стека. Чтобы инициализировать динамический стек, т.е. создать пустой динамический стек достаточно указателю на вершину стека задать значение NULL: S = NULL. Обращение к значению элемента в вершине динамического стека: S -> pole1.

Основные операции над стеком (S - переменная типа stack):

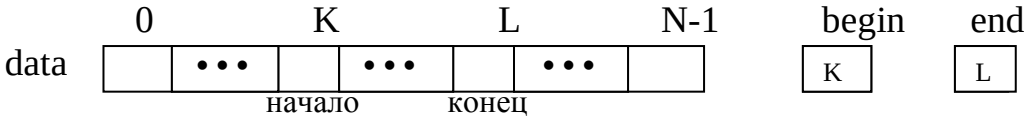
Таблица 1

Название Операции	Статистический способ определения стека	Динамический способ определения стека
Очистить стек (создать стек)	В поле S.top ← ∅	S:= NULL
Поместить элемент в стек	Если S.top < n-1 , то S.top = S.top + 1; S.data[S.top] = элемент; , иначе ошибка - «переполнение стека»	Добавить элемент в начало линейного списка (см. Лабораторная работа № 1)
Взять элемент из стека	Если стек не пуст, то элемент = S.data[S.top]; S.top = S.top – 1; , иначе ошибка - “операция не определена”	Элемент = S ->pole1; Удалить первый элемент из линейного списка, сделав первым следующий элемент списка: S = S -> pole2; Операция определена, если стек не пуст.
Проверка пуст ли стек?	пуст (S)=true, если S.top =∅ пуст(S)= false, если S.top ≠∅	пуст(S)=true, если S = NULL пуст(S)=false,если S!= NULL

Статическую очередь (очередь отображается в статической памяти) можно представить :

а) линейным массивом с двумя указателями: на начало и на конец очереди. Указатель на начало определяет позицию очереди (массива), откуда можно взять элемент, указатель на конец – позицию, куда кладем элемент:

```
struct  ch1
    { int   begin, end ;
      type   data  [ N ] ; }
```



Задать очередь – это определить переменную типа ch1:

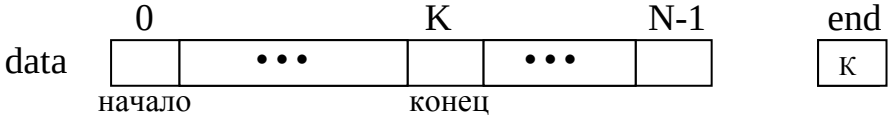
```
ch1    Q;
```

Чтобы инициализировать очередь достаточно определить начальное значение указателя на начало и/или конец очереди. Для этого полю begin дадим значение, не принадлежащее диапазону от 0 до N-1 (а полю end можно задать значение, указывающее на первый элемент очереди – это упростит работу с указателями при включении элемента в очередь). При включении элемента в непустую очередь увеличивается на единицу значение указателя end, при взятии элемента из очереди значение указателя на начало. При включении элемента в пустую очередь увеличиваются оба указателя.

Если поле end равно N-1, то очередь полна. Здесь не используются освободившиеся после взятия элементов из очереди позиции, поэтому возможно мнимое переполнение очереди. Если end равно N-1 и begin равно 0 – очередь полна, а если begin больше 0, то очередь псевдополна (т.е. из очереди были взяты элементы и в начале массива есть свободные позиции). Обращение к элементу при взятии из очереди: Q.data [Q.begin], при занесении в очередь: Q.data [Q.end]. Если в очереди один элемент, то оба указателя показывают на него, т.е. их значения равны, если из очереди взять единственный элемент, она должна стать пустой;

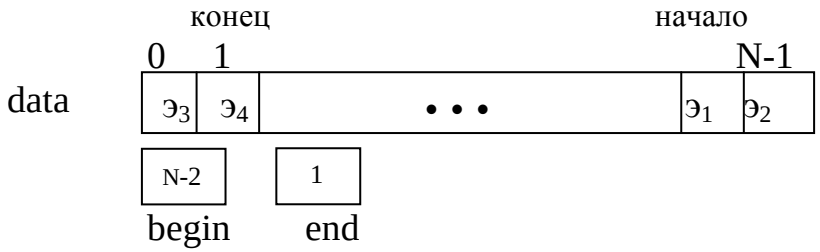
б) линейным массивом с одним указателем на начало или на конец очереди, вторая позиция очереди фиксируется (например, начало очереди это первый элемент массива. В таком случае очередь сдвигается после каждого взятия из нее элемента):

```
struct ch2
    { int  end ;
      type  data [ N ] ;
```



в) линейным массивом с двумя указателями (аналогично варианту а). При достижении конца массива (последнего элемента) очередь сдвигается на начало массива (к первому элементу), если есть свободные позиции в начале массива. Таким образом, освобождаемся от псевдопереполнения очереди;

г) циклическая очередь – линейный массив с двумя указателями (аналогично варианту а), в котором при достижении каким-либо указателем конца массива (последнего элемента) значение этого указателя очереди округляется по модулю N, где N – размер массива. Таким образом, в циклической очереди при достижении конца массива (end = = N-1) и наличии свободных позиций в начале (begin > 0), новый элемент помещается в массив на место с индексом 0, при этом указателю end присваивается значение 0.



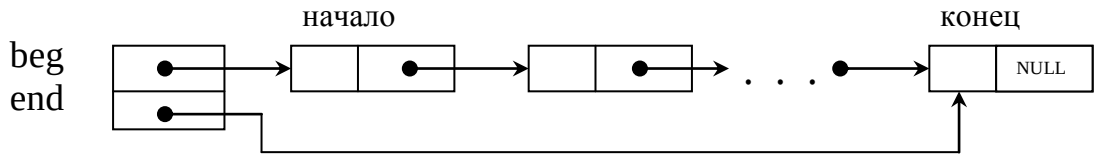
Очередь будет полна, если поле end = = N-1, а поле begin = = 0 или поле end = = K, а begin = = K+1. Здесь 0 < K < N-1.

В динамической памяти очередь можно представить:

а) линейным односвязным списком с двумя указателями:

```
struct   list1
  { type  pole1;
    list1  *pole2; }

struct   ch3
     { list1  *beg, *end ; }
```



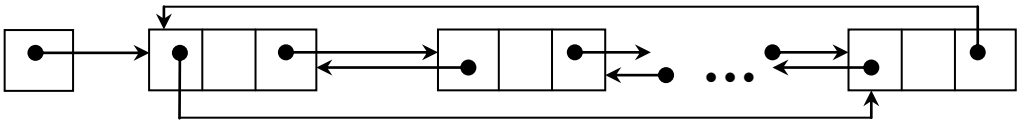
Задать динамическую очередь - это определить переменную типа ch3:

```
ch3   Q1;
```

Чтобы инициировать очередь достаточно указателю на начало Q1.beg и/или на конец Q1.end присвоить значение NULL. При взятии единственного элемента из очереди она должна стать пустой;

б) линейным двусвязным циклическим списком с одним указателем:

```
struct    list2
   {  type  pole1;
      list2 *pole1, *pole2; }
```



Здесь начало (аналогично конец) очереди можно сделать как в начале так и в конце списка.

Кроме рассмотренных вариантов простых очередей существует понятие очереди с приоритетом. Элемент такой очереди имеет не только значение, но и вес или приоритет этого значения. При включении элемента в очередь с приоритетом сначала отыскивается место элемента в соответствии с его приоритетом. Элемент с наивысшим приоритетом помещается в начало очереди, элемент с низшим весом в конец очереди. Такой очереди наилучшим образом отвечает двусвязный циклический список.

Основные операции над очередью аналогичны операциям со стеком (с учетом особенностей этой структуры).

Еще одной разновидностью этих структур данных является дек. Дек это список, у которого обе позиции: начало и конец списка доступны для добавления и взятия элемента. Таким образом, дек может быть и стеком, и очередью, и комбинацией этих структур. Наиболее часто дек представляется структурой с ограничением на вход или выход: дек с ограниченным входом (только одна позиция доступна для добавления элемента) и дек с ограниченным выходом (только одна позиция доступна для взятия элемента из дека).

2. Контрольные вопросы

1. Понятие структуры данных стек, очередь, дек.
2. Представление в памяти структур данных стек, очередь, дек.
3. Задание структур данных стек, очередь, дек.
4. Основные операции над структурами данных стек, очередь, дек.
5. Достоинства и недостатки различного представления в памяти структур данных стек, очередь, дек.
6. Использование структур данных стек, очередь для решения задач.

3. Варианты задания

Данная лабораторная работа состоит в выполнении варианта задания из пункта 1 и пункта 2.

1. Написать операции работы с заданной структурой данных, включив их в один модуль (файл). К основным операциям (см. таблицу 1) добавить операцию, показывающую содержимое структуры после выполнения какого-либо действия с ней. Эту операцию реализовать на основе базовых операций:

а) основные операции над статическим стеком;
б) основные операции над динамическим стеком;
в) операция “принадлежит ли заданный элемент ” статическому стеку;
г) операция “принадлежит ли заданный элемент ” динамическому стеку;
д) основные операции над статической очередью (вид очереди: а, б, в,г);
е) основные операции над динамической очередью (вид очереди:а, б);
ж) операция “добавить элемент в очередь” для статической очереди с приоритетом (вид очереди: а, б, в, г). На вход такой очереди подается элемент и его приоритет, определяющий место элемента в очереди. В очереди с приоритетом элементы должны располагаться в порядке возрастания или убывания приоритетов;
з) операция “добавить элемент в очередь” для динамической очереди с приоритетом (вид очереди: а, б) см. 1.ж;
и) операция “принадлежит ли заданный элемент ” статической очереди (вид очереди а, б, в, г);
к) операция “принадлежит ли заданный элемент ” динамической очереди (вид очереди: а, б);
л) основные операции над статическим деком;
м) основные операции над динамическим деком.

2. Написать программу, демонстрирующую выполнение операций над заданной структурой данных. Эту программу надо поместить в свой модуль (файл). Модуль с основными операциями включать в программу, используя директиву include.

Л А Б О Р А Т О Р Н А Я Р А Б О Т А № 3

СТРУКТУРЫ ДАННЫХ: ДЕРЕВЬЯ

Цель работы: изучить основные алгоритмы работы с деревьями; получить практические навыки разработки и использования этих структур и алгоритмов для решения задач.

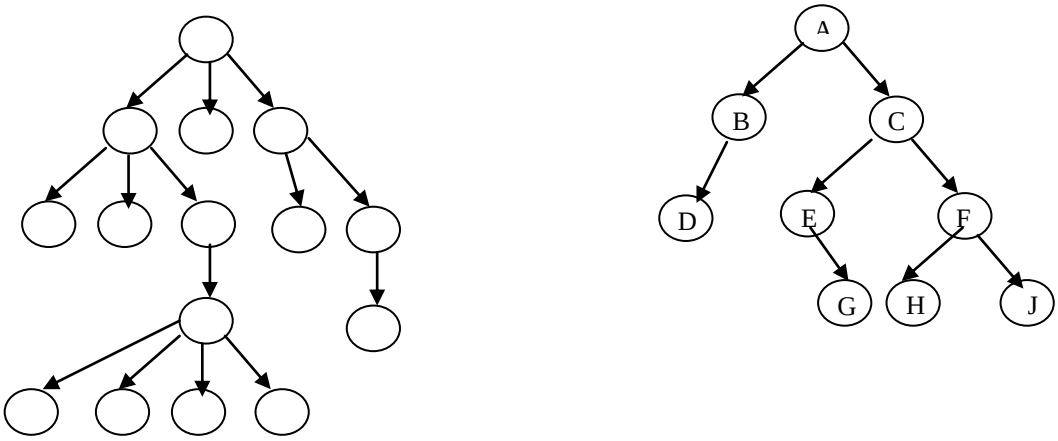
1. Методические указания

1.1. Понятие дерева. Виды деревьев

Дерево – это конечное множество T , возможно пустое, в противном случае, состоящее из одного или более элементов (узлов или вершин дерева) таких, что:

а) имеется один специально обозначенный элемент, называемый корнем данного дерева;

б) остальные элементы содержатся в $m > 0$ попарно непересекающихся множествах $T_1, \ldots, T_m$, каждое из которых в свою очередь является деревом; деревья $T_1, \ldots, T_m$ называются поддеревьями данного корня (рис.3.1.а).



а б

Рис.3.1

Если имеет значение относительный порядок поддеревьев $T_1, \ldots, T_m$, то говорят, что дерево является упорядоченным. Число поддеревьев данного узла называется степенью этого узла. Узел с нулевой степенью называется концевым узлом (или листом или терминальным узлом), все остальные элементы – внутренние узлы (нетерминальные). Максимальная степень всех вершин называется степенью дерева. Корень дерева имеет уровень равный 0. Остальные вершины имеют уровень на единицу больше уровня непосредственного предка. Максимальный уровень какой-либо из вершин называется глубиной или высотой дерева. Минимальная высота при заданном числе вершин достигается, если на всех уровнях, кроме последнего, помещается максимально возможное число вершин. Максимальное число вершин в дереве высотой h достигается в том случае, если из каждой вершины, за исключением уровня h, исходят d поддеревьев, где d –

степень дерева: на 0-м уровне 1 вершина, на 1-м – d потомков, на 2-м – d^2 потомков, на 3-м уровне d^3 потомков и т.д.

Наиболее широко используются двоичные (бинарные) деревья (рис.3.1.б). Бинарное дерево это конечное множество элементов, которое либо пусто, либо состоит из корня и из двух непересекающихся бинарных деревьев, называемых левым и правым поддеревьями данного корня. Таким образом, каждый элемент бинарного дерева имеет 0, 1 или 2 поддерева. Бинарное дерево – упорядоченное дерево, так как в нем различают левое и правое поддеревья.

Дерево поиска – это двоичное дерево, в котором выполняются следующие условия (см. рис.3.2):

а) l(u) < l(v) для всех вершин u, v, если вершина u находится в поддереве, корень которого – левый преемник v;

б) l(u) > l(v) для всех вершин u, v, если вершина u находится в поддереве, корень которого – правый преемник v;

в) для всякого значения a∈ S существует единственная вершина v, для которой l(v) = a (S – множество значений, допускающих сравнения).

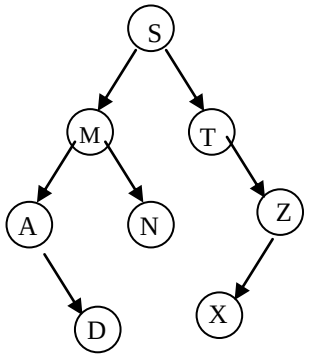


Рис.3.2

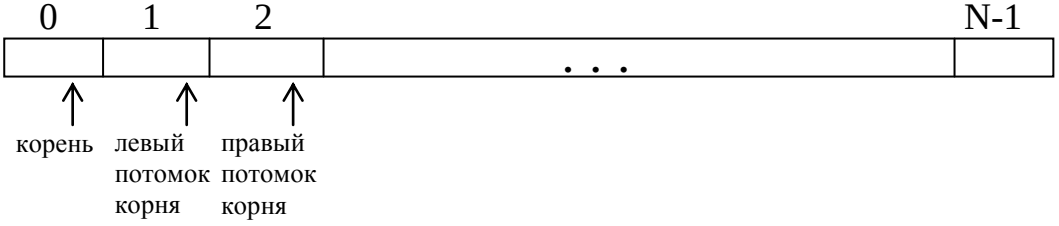
Определение структуры дерева, данное выше, является рекурсивным и отражает рекурсивную природу самой структуры.

Структуру данных – дерево можно представить как в статической, так и в динамической памяти. В статической памяти дерево можно представить массивом, для которого определено понятие пустого элемента:

```
struct stree
  { type  elem [ N ]; }
stree   d;
```

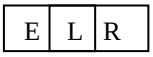
или

```
type    d [ N ];
```



Вершины двоичного дерева располагаются в массиве следующим образом: если k – индекс вершины дерева, то ее потомки находятся в элементах массива с индексами 2k + 1 и 2(k + 1); корень дерева находится в элементе с индексом 0 (при индексации в массиве от 1 до N индексы потомков k-ой вершины: 2k и 2k + 1, корень имеет индекс 1).

В динамической памяти дерево представляется иерархическим списком. Задать элемент двоичного дерева можно как элемент списка с тремя полями: два ссылочных поля, содержащие указатели на его левое (L) и правое (R) поддеревья, и информационное поле (E):



Определение типа элемента бинарного динамического дерева:

```
struct btree
    {type   elem;
     btree  *left, *right;}
```

Здесь type – тип информационного поля (если информационное поле имеет сложную структуру, то type может быть типом - указатель на объект, содержащий значение элемента дерева).

Чтобы определить дерево как единую структуру, надо задать указатель на корень дерева:

```
btree     * d ;
```

На рис.3.3 представлено двоичное динамическое дерево d в соответствии с определением типа элемента, сделанным выше. Элементы дерева со степенью 0 и 1 имеют два или одно ссылочное поле со значением равным пустому указателю (NULL).

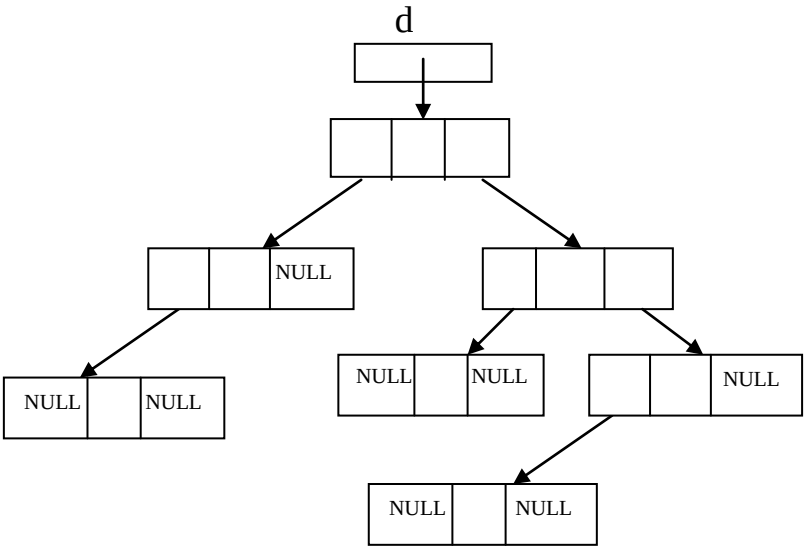


Рис.3.3

Обрабатывая дерево, приходится просматривать его элементы – эта операция называется обход дерева (или прохождение дерева).

Обход дерева – это способ методичного исследования его узлов, при котором каждый узел проходится точно один раз. Полное прохождение динамического дерева дает линейную расстановку его узлов.

Возможны два способа прохождения бинарного дерева (если дерево не пусто):

а) в глубину:

- прямой (префиксный) порядок: корень, левое поддерево в прямом порядке, правое поддерево в прямом порядке (линейная расстановка узлов дерева, представленного на рис.3.1.б: ABDCEGFHJ);

Алгоритм прямого обхода:

```
{ S = NULL;
    while ( d != NULL)
   {  обработка ( d );
      if ( d->left != NULL && d->right != NULL)
            {встек (S, d->right); d = d->left;}
         else  if ( d->left = = NULL && d->right = =NULL)
                  if ( S!= NULL) изстека (S, d); else d = NULL;
               else  if (d->left != NULL) d = d->left;
                              else d = d->right;
            }
  }
```

Рекурсивное описание алгоритма прямого обхода:

```
пр_обход ( btree *d)
{ if ( d != NULL ) { обработка ( d ); пр_обход ( d->left );
                                       пр_обход ( d->right ); }
  }
```

- обратный (инфиксный) порядок: левое поддерево в обратном порядке, корень, правое поддерево в обратном порядке (линейная расстановка узлов дерева, представленного на рис.3.1.б: DBAEGCHFJ);

Алгоритм обратного обхода:

```
{ S = NULL;  F = 1;
   while ( F )
    {     while ( d != NULL )
             { встек ( S, d ); d = d->left; }
      if ( S != NULL ) { изстека ( S, d );
                          обработка ( d ); d = d-right;}
            else F = 0;
            }
       }
```

Рекурсивное описание алгоритма обратного обхода:

```
 обр_обход ( btree *d )
{ if ( d != NULL ) { обр_обход ( d->left );  обработка ( d );
                     обр_обход ( d->right ); }
  }
```

- концевой (постфиксный) порядок: левое поддерево в концевом порядке, правое поддерево в концевом порядке, корень (линейная расстановка узлов дерева, представленного на рис.3.1.б: DBGEHJFCA);

Алгоритм концевого обхода:

```
{ S = NULL;
  do
        while ( d != NULL )
            { встек ( S, d, 0 ); d = d->left; }
     if ( S != NULL )
            { do
                    изстека ( S, d, F );  if ( F ) обработка ( d );
                while ( F && S != NULL );
              if ( ! F ) { встек ( S, d, 1 ); d = d->right; }
   while ( S != NULL );
     }
```

Рекурсивное описание алгоритма концевого обхода:

```
 конц_обход ( btree *d )
{ if ( d != NULL ) { конц_обход ( d->left );  конц_обход ( d->right );
                                        обработка ( d ); }
        }
```

б) в ширину: узлы дерева посещаются слева направо, уровень за уровнем вниз от корня (линейная расстановка узлов дерева, представленного на рис.3.1.б: ABCDEFGHJ).

Алгоритм обхода в ширину:

```
{ Q = NULL;
  if ( d != NULL ) { в_очередь ( Q, d );
          do
              из_очереди ( Q, d ); обработка ( d );
              if ( d->left != NULL ) в_очередь ( Q, d->left );
              if ( d->right != NULL ) в_очередь ( Q, d->right );
          while ( ! пуста_очередь ( Q ) );
                      }
     }
```

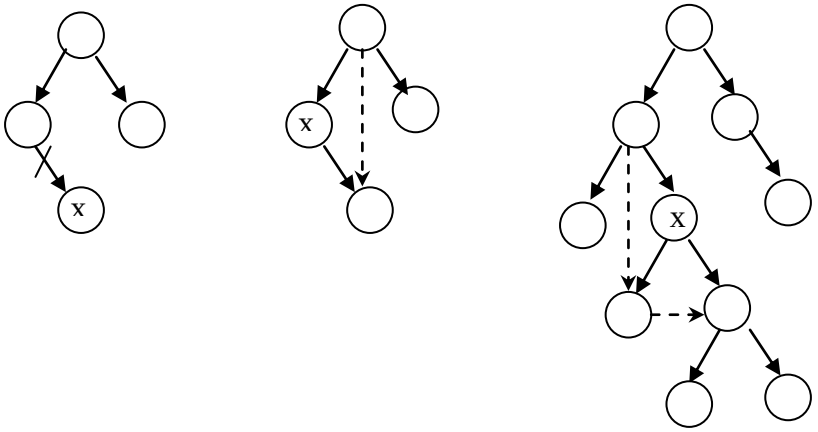
Основные операции при работе с деревьями:

а) поиск элемента в дереве. Операция заключается в прохождении узлов дерева в одном из рассмотренных выше порядке. При прохождении дерева поиска достаточно пройти только то поддерево, которое возможно содержит искомый элемент;

б) включение элемента в дерево. Операция заключается в добавлении листа (исключая сбалансированное дерево – включение элемента в сбалансированное дерево приводит, как правило, к его перестройке) в какое-то поддерево, если такого элемента нет в исходном дереве. При включении нового элемента в дерево поиска лист добавляется в то поддерево, в котором не нарушается отношение порядка;

в) удаление элемента из дерева. Операция заключается в изменении связей между дочерними и родительскими, по отношению к удаляемому, элементами (исключая сбалансированное дерево – удаление элемента из сбалансированного дерева приводит, как правило, к его перестройке); здесь необходимо рассматривать три случая: удаление листа (см. рис.3.4.а), удаление элемента с одним потомком (см. рис.3.4.б), удаление элемента с двумя потомками (см. рис.3.4.в).

При удалении элемента степени 2 из дерева поиска изменять связи необходимо так, чтобы не нарушалось установленное в дереве отношение порядка. Лучшим вариантом в этом случае будет перемещение на место удаляемого элемента либо самого правого листа из левого поддерева удаляемого элемента, либо самого левого листа из правого поддерева удаляемого элемента;



а б в

Рис.3.4

г) сравнение деревьев (проверка деревьев на равенство). Операция заключается в прохождении обоих деревьев в одном порядке до тех пор, пока либо не будут пройдены оба дерева, либо одно из них, либо соответствующие элементы окажутся не равными, либо в одном из деревьев не будет обнаружено отсутствие соответствующего элемента. Только в случае равенства деревьев оба дерева будут пройдены одновременно;

д) копирование дерева. Операция заключается в прохождении исходного дерева и построении нового дерева с элементами, информационные поля которых равны информационным полям соответствующих элементов исходного дерева.

1.2. Ввод дерева

Чтобы ввести дерево надо выбрать какой-то способ перечисления его узлов. Одним из возможных способов является так называемая списочная запись (представление дерева в виде списка). Список – это конечная последовательность атомов, число которых, может быть и равно нулю. Атом – это понятие, определяющее элемент из некоторого множества объектов, либо список. Используя скобки, список можно задать перечислением через запятую его атомов

(порядок перечисления атомов определяет их упорядочение). Таким образом, дерево, представленное на рис.3.1.б можно записать в виде следующего списка: (A, (B, (D, 0, 0), 0), (C, (E, 0, (G, 0,0)), (F, (H, 0, 0), (I, 0, 0))))

Ввод дерева, представленного таким списком, можно описать рекурсивной функцией (тип дерева btree описан выше, здесь type – тип информационного поля элемента – char):

```
btree * build_tree ( )
  { char  sym;
    btree *d;
        scanf (“%c”, &sym );
        switch ( sym )
      {case ‘ (‘ : { d =  new btree;
                     scanf (“%c”, &sym );  d->elem  = sym;
                      d->left =  build_tree (  );
                      d->right = build_tree (  ); scanf( “%c”, &sym );
                       return  d  ;  }
        case  ‘0’  :  return NULL ;
        case  ‘,’   : d  = build_tree (  );  return d ;
                }
                   }
```

2.Контрольные вопросы

1. Понятие дерева, двоичного дерева, упорядоченного дерева, дерева поиска.
2. Способы задания дерева.
3. Способы обхода дерева (в глубину: прямой, обратный, концевой; в ширину).
4. Показать, можно ли построить бинарное дерево, если заданы: а) прямой и обратный порядки расположения его узлов; б) прямой и концевой порядки; в) обратный и концевой порядки? Если построить дерево можно, то приведите алгоритм построения.
5. Найдите все бинарные деревья, узлы которых располагаются точно в одной и той же последовательности: а) в прямом и обратном порядках; б) в прямом и концевом порядках; в) в обратном и концевом порядках.
6. Каково наибольшее число вершин, находящихся одновременно в стеке при обходе двоичного дерева, содержащего n элементов: а) в прямом, б) обратном, в) концевом порядках?

2. Варианты задания

Во всех задачах тип значений элементов дерева простой. Исходное дерево после ввода распечатать в одном из порядков.

1. В заданном бинарном дереве подсчитать число его листьев и напечатать их значения:

а) при прямом обходе дерева;
б) при обратном обходе дерева;
в) при концевом обходе дерева;
г) реализуя обход, рекурсивно.

2. В заданном бинарном дереве найти первое вхождение заданного элемента и напечатать пройденные при поиске узлы дерева:
   а) при прямом обходе дерева;
   б) при обратном обходе дерева;
   в) при концевом обходе дерева;
   г) реализуя обход, рекурсивно.
3. В заданном непустом бинарном дереве найти длину (число ветвей) пути от корня до ближайшей вершины со значением равным заданному :
   а) при прямом обходе дерева;
   б) реализуя обход, рекурсивно.
4. В заданном непустом бинарном дереве подсчитать число вершин на n- ом уровне, считая корень вершиной 0-го уровня.
5. Задано бинарное дерево. Определить, есть ли в этом дереве хотя бы два одинаковых элемента.
6. Распечатать все элементы заданного непустого бинарного дерева по уровням: сначала из корня дерева, затем (слева направо) из вершин, дочерних по отношению к корню, затем (слева направо) из вершин, дочерних по отношению к этим вершинам, и т.д.
7. Формулу вида:

< формула > ::= < терминал > | (< формула > < знак > < формула >)
< знак > ::= + | - | *
< терминал > ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

можно представить в виде двоичного дерева (‘дерева – формулы ‘):

- формула из одного терминала (цифры) представляется деревом из одной вершины (корнем) с этим терминалом;
- формула вида (f_1 s f_2) - деревом, в котором корень – это знак s, а левое и правое поддеревья – это соответствующие представления f_1 и f_2 :

а) по заданной формуле построить дерево – формулу, обходя дерево – формулу в 1)прямом, 2)обратном, 3)концевом порядке, напечатать его элементы и вычислить (как целое число) значение;

б) проверить, является ли заданное двоичное дерево (с элементами типа char) деревом – формулой и напечатать его элементы в порядке 1)прямого, 2)обратного, 3)концевого обхода;

в) пусть в дереве – формуле в качестве терминалов используются не только цифры, но и буквы, играющие роль переменных; преобразовать заданное дерево – формулу, заменяя в нем все поддеревья, соответствующие формулам (($f_1 \pm f_2$) * f_3) и (f_1 *($f_2 \pm f_3$)) на поддеревья, соответствующие формулам ((f_1 * f_3) ± (f_2 * f_3)) и ((f_1 * f_2) ± (f_1 * f_3)); распечатать преобразованное дерево в 1)прямом, 2)обратном, 3)концевом порядке;

г) выполнить в заданном дереве – формуле преобразования, обратные преобразованиям из пункта в; распечатать преобразованное дерево в 1)прямом, 2)обратном, 3)концевом порядке.

8. Заданную последовательность из строчных латинских букв, оканчивающуюся точкой, представить в виде дерева поиска, затем преобразовать его, удалив символ с а) наименьшим, б) наибольшим значением; распечатать преобразованное дерево в 1)прямом, 2)обратном, 3)концевом порядке.

9. Заданную последовательность целых чисел, оканчивающуюся нулем, представить в виде дерева поиска, затем преобразовать его, добавив еще одно целое число; распечатать преобразованное дерево в 1)прямом, 2)обратном, 3)концевом порядке.

Л А Б О Р А Т О Р Н А Я Р А Б О Т А № 4

УПРАВЛЕНИЕ ТАБЛИЦАМИ

Цель работы: получить практические навыки организации таблиц, обработки таблиц и их использования при решении задач.

1. Методические указания

Таблица – это список, состоящий из конечного множества элементов, причем каждый элемент характеризуется рядом признаков (свойств). Один из признаков, называемый ключом, позволяет отличить один элемент от другого (идентифицировать элемент). Ключ может однозначно определять элемент таблицы (ключи всех элементов различны) или неоднозначно (в таблице есть элементы с равными ключами).

Все действия над элементами выполняются в соответствии с их ключами: по ключу элементы выбираются из таблицы и добавляются в нее.

Таблицы можно хранить как в статической, так и в динамической памяти.

В данной работе рассматриваются статические таблицы (таблица отображается в статической памяти). Таблица в статической памяти отображается:

а) массивом и величиной, задающей фактическое количество элементов таблицы, её можно представить следующим образом (см. рис.4.1):

элемент 1	признак 1	признак 2	. . .	признак m
элемент 2				
	⋮			
элемент n				
	⋮			
N				

n

Рис.4.1

где N - максимальное количество элементов в таблице (размер таблицы);
n - фактическое количество элементов в таблице.

Задать такую таблицу можно следующим образом:

```
struct table
  { type  elem [ N ];
    int   n;  }
```

table T;

Если поле n = N , таблица полна, если поле n < N , в таблице есть свободные позиции. Если поле n = 0, таблица пуста.

б) вместо задания n можно ввести, если это возможно, понятие пустого элемента. Пустой элемент позволяет определить пуста таблица, полна таблица или в ней есть еще свободные позиции. Например, если в таблице нет ни одного пустого элемента, она полна, если первый элемент – пустой элемент, таблица пуста.

	признак 1	признак 2	. . .	признак m
элемент 1				
элемент 2				
	⋮			
элемент n				
	⋮			
N				

Рис. 4.2

В этом случае таблица задается (рис.4.2.):

type T [N];

Здесь type – тип элемента таблицы, который, как правило, является сложным типом.

Неупорядоченная таблица – это таблица, элементы которой располагаются в порядке их поступления в таблицу.

Упорядоченная таблица – это таблица, между элементами которой установлено отношение порядка. Это отношение, как правило, устанавливается на признаке ключ. Поэтому говорят, что таблица упорядочена по ключам элементов. Таблица упорядочена по возрастанию значений ключа, если ключ(T_i) < ключ(T_{i+1}) для всех i = 1, 2 , . . . , n-1 (здесь T_i - i -й элемент таблицы T). Таблица упорядочена по убыванию значений ключа, если ключ(T_i) > ключ(T_{i+1}) для всех i = 1, 2, . . . , n-1.

Таблица с вычисляемым входом (хеш-таблица) – это таблица, элементы которой располагаются в соответствии с некоторой функцией расстановки (хеш-функцией). Функция расстановки f (ключ) вычисляет для каждого элемента таблицы по его ключу номер (позицию) элемента в массиве. Таким образом, диапазон значений функции f (ключ) –

$$0 \div N-1 \text{ или } 1 \div N.$$

При работе с хеш-таблицей необходимо определять, находится ли в позиции с номером равным f (ключ) элемент или нет. Поэтому в памяти хеш-таблица отображается вторым способом, т.е. массивом с заданием значения «пустой элемент». Хеш-таблица должна быть инициирована так, чтобы все элементы таблицы получили значение «пусто». Задание хеш-таблицы:

type T [N];

Если ключ(T_i) ≠ ключ (T_j) и f (ключ (T_i)) ≠ f (ключ (T_j)) при $i \neq j$, для всех i и j = 0, 1, 2, . . . , n , то элементы таблицы взаимно - однозначно отображаются в элементы массива. Организованная таким образом таблица называется таблицей с прямой адресацией. Прямая адресация применима, если количество возможных ключей невелико. Если в такой таблице число реально присутствующих элементов мало по сравнению с N, то в таблице много пустых элементов (т.е. много памяти тратится зря).

Если ключ (T_i) ≠ ключ (T_j), а f (ключ (T_i)) = f (ключ (T_j)) при $i \neq j$, то нет однозначного отображения элемента таблицы в элемент массива. Эта ситуация называется коллизией (столкновением). Если заранее неизвестен диапазон значений ключа, то выбрать функцию расстановки, дающую однозначное отображение практически невозможно. В таких таблицах возникает проблема разрешения коллизий, которая включает две задачи:

1) выбор функции расстановки;
2) выбор способа разрешения коллизий (способа рехеширования).

Функция расстановки подбирается в каждом случае в зависимости от ключа так, чтобы алгоритм ее вычисления был несложным. Функцию расстановки подбирают из условия случайного и возможно более равномерного отображения (перемешивания) ключей в адреса хранения, но «случайная» хеш-функция должна быть детерминированной в том смысле, что при ее повторных вычислениях с одним и тем же ключом, она должна давать одно и то же значение.

Известно несколько методов получения хеш-функции, при этом обычно предполагают, что область определения хеш-функции – целые неотрицательные числа, если ключи не являются такими их всегда можно преобразовать к целому типу:

а) метод деления: ключу ставится в соответствие остаток от деления на N – значение функции вычисляется по формуле:

$$f(\text{ключ}) = \varphi(\text{ключ}) \bmod N,$$

здесь φ (ключ) – функция, преобразующая ключ элемента в целое число;
(mod N) - означает по модулю N, т.е. остаток от деления на N. Здесь хорошо задавать N простым числом, плохо, если $N = 2^p$ или $N = 10^p$.

б) метод умножения: значение хеш-функции – это какая-то часть преобразованного значения ключа, либо часть самого значения ключа. Значение хеш-функции вычисляется по формуле:

$$f(\text{ключ}) = [N \cdot (\text{ключ} \cdot A \bmod 1)],$$

здесь A – константа из интервала (0,1). Выбор константы зависит от исходных данных, но любое значение константы дает не плохие результаты. Выражение (ключ·A mod 1) – это дробная часть ключ·A.
Квадратные скобки – это взятие целой части $N \cdot (\text{ключ} \cdot A \bmod 1)$.
Если $N = 2^m$, (степень двойки) или $N = 10^p$, то операция умножения на N – сдвиг значения на m (или p) разрядов. Таким образом, значением хеш-функции будет m (или p) разрядов выражения (ключ·A mod 1). В некоторых случаях при $N = 2^m$ в

качестве значения функции можно взять m - разрядное двоичное число либо методом выделения каких-либо m битов из значения ключа, либо применяя логическую операцию над некоторыми частями ключа, либо, разделив ключ на m частей, просуммировать их и в качестве значения функции взять младшие m разрядов (это допустимо, если эта часть ключа уникальна или повторяется редко).

Разрешать коллизии можно в исходной или в дополнительной таблице. Если в исходной, то такая таблица называется таблицей с перемешиванием или с открытой адресацией. В таблицах с перемешиванием применяют линейное (метод последовательных испытаний или проб), случайное или квадратичное рехеширование. Наиболее простой метод - линейное рехеширование, который заключается в том, что функция вторичной расстановки $f^{|}$ определяется как

$$f^{|}(\text{ключ}, i) = (f (\text{ключ}) + i) \bmod N ,$$

здесь значение i равно 1, 2, . . . , а f (ключ) это значение, дающее коллизию. Рехеширование продолжается до тех пор, пока очередная позиция массива с номером $f^{|}$(ключ, i) не окажется пустой или равной снова своему начальному значению (в этом случае таблица полна). Линейное рехеширование применяют, если таблица никогда не бывает полностью заполненной (лучше, если она заполнена не более чем на 70%). Если допускается операция удаления элемента из таблицы, то в таблице с перемешиванием надо различать еще одно состояние: элемент удален из таблицы.

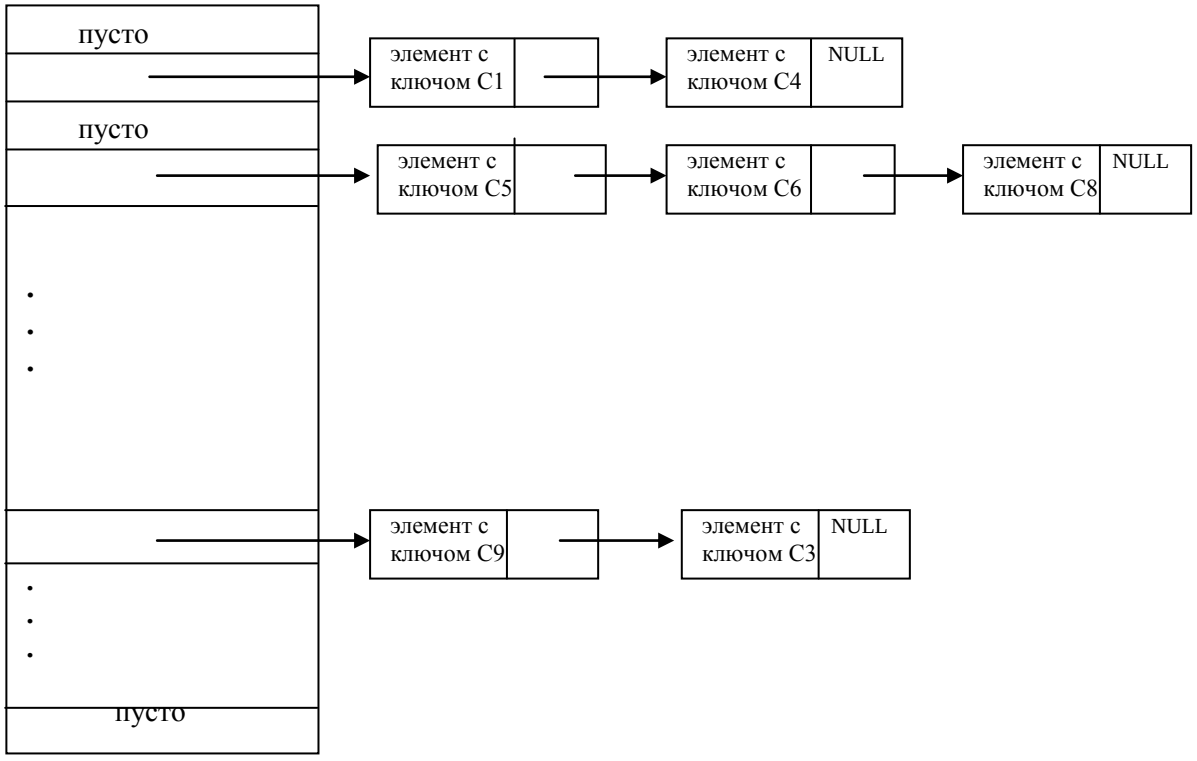


Рис.4.2.

В случае разрешения коллизии в дополнительной таблице удобно использовать хеширование с цепочками. В таких таблицах элементы, которым соответствует

одно и то же хеш-значение, связываются в цепочку – список, а в самом элементе таблицы хранится ссылка на список (см. рис.4.2).

Основные операции над таблицами:

1. Поиск в таблице элемента с заданным ключом.
2. Включение заданного элемента в таблицу.
3. Исключение из таблицы элемента с заданным ключом.

Различные способы организации таблиц принято сравнивать по времени выполнения в них основных операций и прежде всего по времени поиска. В неупорядоченной таблице поиск элемента в среднем будет выполнен за n/2 сравнений, так как поиск заключается в последовательном просмотре элементов таблицы (максимальное число сравнений n). В упорядоченной таблице кроме последовательного просмотра можно применять бинарный поиск (поиск делением пополам), максимальное число сравнений при котором $1 + \log_2 n$.

Включение элемента в неупорядоченную таблицу заключается в добавлении элемента в таблицу (если $n < N$) на очередное свободное место. Включению элемента в упорядоченную таблицу предшествует поиск места k для нового элемента с учетом отношения порядка, затем освобождение этого места (если оно занято) смещением всех последующих элементов, начиная с k-го вниз на одну позицию. Таким образом, включение в упорядоченную таблицу требует больше времени, чем в неупорядоченную таблицу.

Исключению элемента из таблицы также предшествует поиск по ключу элемента и, если элемент в таблице есть, то исключение заключается в смещении всех последующих элементов на одну позицию вверх.

Время поиска в хеш-таблице без коллизий постоянно – это время вычисления функции расстановки. Время поиска в таблицах с перемешиванием для метода линейного рехеширования зависит от коэффициента загрузки таблицы $k = n/N$, среднее число сравнений в этом случае $e = (1 - k/2) (1 - k)$. В хеш-таблице с цепочками время поиска зависит от длины списка. При включении элемента в хеш-таблицу вычисляется значение функции расстановки. Если позиция с вычисленным по хеш-функции номером «пуста», то элемент помещается в данную позицию, иначе необходимо разрешать коллизию. При исключении элемента из хеш-таблицы по ключу вычисляется номер его позиции в таблице. Если позиция не «пуста» и ключ элемента находящегося в этой позиции совпадает с ключом удаляемого элемента, то чтобы исключить элемент, достаточно сделать данную позицию «пустой». Если ключи не совпадают, то поиск элемента осуществляется с учетом метода разрешения коллизии.

2. Варианты задания

1. В файле PROG содержится текст программы на языке Си. Файл разделен на строки произвольной длины (но не более 256). Используя определения и описания, построить:
   1) таблицу имен. Элемент таблицы должен содержать имя программного объекта, его тип, размер памяти, требуемой объекту, число компонент, тип компонент (последние два признака для сложных типов). Организовать таблицу как:

а) неупорядоченную;
б) упорядоченную;
в) таблицу с вычисляемым входом (таблица с перемешиванием);

2) таблицу констант. Элемент таблицы содержит: значение константы, имя константы, тип константы, размер памяти. Организовать таблицу как:
а) неупорядоченную;
б) упорядоченную;
в) таблицу с вычисляемым входом, считая, что тип констант целый и диапазон значений от -255 до 255.

2. В файле WORK содержатся результаты работы цеха за день. Элемент файла включает: шифр изделия (8 - символьный код), наименование изделия, количество (штук). Построить таблицу, содержащую результаты работы за день, считая ключом шифр изделия. Элемент таблицы имеет ту же структуру, что и элемент файла. Содержащаяся в файле информация с равными ключами должны быть помещена в таблицу один раз с общим количеством штук изделия. Организовать таблицу как:
а) неупорядоченную;
б) упорядоченную;
в) таблицу с вычисляемым входом (таблица с перемешиванием).

3. В спортивных соревнованиях участвуют n команд. В файле SPORT содержатся прогнозы результатов соревнований. Каждый прогноз включает номер команды, занявшей первое место, номер команды, занявшей последнее место, номера команд, входящих в первую тройку сильнейших команд. Построить таблицу, содержащую проценты голосов, отданных командам - претендентам на первое место, командам - претендентам на последнее место и проценты голосов, отданных командам - претендентам на первую тройку. Организовать таблицу как:
а) неупорядоченную;
б) упорядоченную;
в) таблицу с вычисляемым входом.

3. Контрольные вопросы

1. Понятие таблицы.
2. Способы организации таблиц.
3. Операции над таблицами.
4. Чем отличается последовательный поиск в упорядоченной таблице от последовательного поиска в неупорядоченной таблице?
5. Чем отличается включение элемента в упорядоченную таблицу от включения элемента в неупорядоченную таблицу?
6. Хеш-функция - как ее построить?
7. Методы рехеширования.
8. Сравнительные оценки операций работы с таблицами.

Л А Б О Р А Т О Р Н А Я Р А Б О Т А № 5

УПОРЯДОЧЕНИЕ ДАННЫХ

Цель работы: изучить основные алгоритмы упорядочения (сортировки) данных.

1. Методические указания

Сортировкой называется процесс (операция) перегруппировки объектов в некотором определенном порядке. Различают внутреннюю сортировку и внешнюю.

Алгоритмы внутренней сортировки упорядочивают данные, хранимые в оперативной памяти, алгоритмы внешней сортировки обрабатывают данные, хранимые во внешней памяти. В лабораторной работе № 6 рассматриваются алгоритмы внутренней сортировки данных, хранимых в статических таблицах.

Если таблица представлена множеством элементов $a_1, a_2, \ldots, a_n$, тогда сортировка есть перестановка элементов $a_{k1}, a_{k2}, \ldots, a_{kn}$, в которой над элементами установлено отношение порядка:

$$f(a_{k1}) <= f(a_{k2}) <= \ldots <= f(a_{kn})$$

или

$$f(a_{k1}) >= f(a_{k2}) >= \ldots >= f(a_{kn})$$

или какое-то другое отношение порядка (здесь f - упорядочивающая функция). В простом случае f задается явно как компонент элемента таблицы, обычно этот компонент – ключ элемента. В таком случае говорят об упорядочении по ключам.

Сравнительная оценка сложности алгоритмов внутренней сортировки осуществляется по следующим параметрам:

- число сравнений ключей элементов;
- число перемещений элементов.

В эффективных алгоритмах стремятся сократить число сравнений и перестановок элементов, а также экономно использовать память. Поэтому алгоритмы сортировки производят перегруппировку элементов в исходном массиве. Алгоритм сортировки n данных, оперирующий сравнениями, имеет минимальную сложность n или выше (т.к. должно быть выполнено минимально n – 1 сравнений), максимальная сложность алгоритма, оперирующего сравнениями, не меньше $n*\log n$.

Основные алгоритмы сортировки базируются на одном из трех способов:

1. Сортировка включением (вставками - by insertion). Исходное множество элементов делят на две части: уже упорядоченную и неупорядоченную. Вначале упорядоченная часть состоит, как правило, из одного элемента – первого элемента, а все остальные элементы находятся в неупорядоченной части. Из неупорядоченной части берут очередной элемент и включают его в упорядоченную часть, помещая (вставляя) на нужное место (т.е. так, чтобы выполнялось отношение порядка). Так продолжают до тех пор, пока последний элемент из неупорядоченной части не будет включен в упорядоченную часть.

Простой вариант сортировки включением – это метод прямого (простого) включения, улучшенный – сортировка методом Шелла.

Алгоритм прямого включения – здесь рассматриваются элементы таблицы (ее упорядоченной и неупорядоченной частей), отстоящие друг от друга на шаг равный 1:

```
{ for (i = 1; i <  T.n ; i ++ )
    { x = T.elem_i ;  c = ключ ( T.elem_i ) ;  j = i - 1;
        while ( j  > = 0 && ключ ( T.elem_j ) > c )
            { T.elem_j+1 = T.elem_j ;  j = j - 1; }
      T.elem_j+1 = x; }
        }
```

Включаемый элемент сравнивается с элементами упорядоченной части, начиная с ее последнего элемента. Число просмотров (упорядоченной части) равно n – 1.

2. Сортировка выбором (by selection). Исходное множество элементов делят также на две части: уже упорядоченную и неупорядоченную. Из неупорядоченной части выбирают нужный элемент (например, с минимальным или максимальным значением в соответствии с отношением порядка) и помещают на очередное место в упорядоченную часть массива. Процесс продолжают до тех пор, пока в неупорядоченной части останется один элемент. Вначале неупорядоченная часть – весь исходный массив, а очередное место в упорядоченной части – первый (или последний) элемент. Простой вариант сортировки выбором – это метод прямого (простого) выбора; улучшенный вариант – сортировка выбором с использованием представления таблицы двоичным деревом (при этом дерево отображается в статической памяти) – сортировка с использованием структуры данных – дерево.

Алгоритм прямого выбора – здесь среди всех элементов неупорядоченной части осуществляется поиск нужного элемента:

```
{ for (i = 0; i <  T.n - 1; i ++)
    { min = T.elem_i ;  k = i;
        for (j = i + 1; j <  T.n ; j++)
            if ( ключ ( T.elem_j ) < ключ ( min ))
                {min = T.elem_j ;   k = j; }
      T.elem_k = T.elem_i ;  T.elem_i = min;  }
        }
```

Число просмотров таблицы (неупорядоченной части) равно n – 1.

3. Сортировка обменом (by exchange). В исходном множестве элементов рассматриваются пары элементов. Если пара содержит инверсию, то она устраняется выполнением обмена этих элементов (инверсия – это пара индексов,

на которой нарушено отношение порядка. Например, пусть отношение порядка – возрастание значений ключей элементов. Если i < j, а ключ (i) >= ключ (j) , то эта пара i и j содержит инверсию). Таким образом, после каждого просмотра хотя бы один элемент окажется на своем месте. Просмотры продолжаются до тех пор, пока в массиве не останется ни одной инверсии. Простой вариант сортировки обменом – метод прямого (простого) обмена («пузырьковая» сортировка), улучшенный вариант – быстрая сортировка.

Алгоритм прямого обмена – здесь сравниваются пары соседних элементов:

вариант 1:

```
{ for ( i = 0; i < T.n - 1; i ++  )
      for ( j =0; j < T.n - 1 - i ; j ++ )
         if ( ключ( T.elem_j )  > ключ ( T.elem_{j+1}) )
                        { x = T.elem_j ;   T.elem_j = T.elem_{j+1} ;
                              T.elem_{j+1} = x; }
          }
```

В данном варианте всегда выполняется максимальное число сравнений. Число просмотров таблицы (неупорядоченной части) равно n – 1.

В случае, когда исходное множество является частично упорядоченным можно улучшить алгоритм прямого обмена, учитывая имеющийся частичный порядок:

вариант 2:

```
{ инверсия = 1; j =0;
   while ( инверсия )
    { инверсия = 0;
      for ( i = 0; i < T.n - 1 - j; i ++ )
         if ( ключ (T.elem_i ) >  ключ ( T.elem_{i+1} )
                 {  r = T_i ;  T_i = T_{i+1} ;  T_{i+1} = r ;
                    инверсия = 1; j ++ ;  }
       }  }
```

В данных алгоритмах таблица T определена следующим образом:

```
struct table
    { type  elem [ N ];
      int    n;  }
  table  T;
```

Здесь поле n определяет фактическое число элементов в таблице.

2. Контрольные вопросы

1. Понятие упорядочения.
2. Характеристика прямых методов сортировки: включения, выбора, обмена.
3. Сравнительная оценка сложности прямых методов сортировки по числу сравнений и числу перемещений элементов.
4. Характеристика улучшенных методов сортировки, оценки их сложности.

3. Варианты задания

1. Упорядочить таблицу, построенную в лабораторной работе № 4 варианты 1.1.а, 1.2.а по ключу (ключ для 1.1.а – имя объекта; для 1.2.а – значение константы) методом:
   а) прямого включения;

   б) прямого выбора;

   в) прямого обмена;

   г) методом Шелла;

   д) сортировки с использованием структуры дерево.

   Используя раздел операторов, дополнить элементы таблицы числом раз использования каждого ключа. Для поиска элементов в таблице использовать:

   а) последовательный поиск;

   б) бинарный поиск.
2. Упорядочить таблицу, построенную в лабораторной работе № 4 варианты 1.1.а, 1.2.а по ключу (ключ для 1.1.а – имя объекта; для 1.2.а – значение константы) методом:
   а) прямого включения;

   б) прямого выбора;

   в) прямого обмена;

   г) методом Шелла.

   Используя раздел операторов, напечатать в порядке возрастания ключей элементы таблицы, описанные, но не используемые в программе. Для поиска элементов в таблице использовать:

   а) последовательный поиск;

   б) бинарный поиск.
3. Упорядочить таблицу, построенную в лабораторной работе № 4 варианты 1.1.б, 1.1.в, 1.2.б, 1.2.в по новому ключу - по не возрастанию частоты использования элемента методом:
   а) прямого включения;

   б) прямого выбора;

в) прямого обмена;

г) методом Шелла.

Используя раздел операторов, дополнить элементы таблицы числом раз использования каждого ключа. Для поиска элементов в таблице использовать:

а) последовательный поиск;

б) бинарный поиск.

4. Упорядочить таблицу, построенную в лабораторной работе № 4 вариант 2.а по не убыванию значений ключа методом:
а) быстрой сортировки;

б) сортировки с использованием структуры дерево;

в) методом Шелла.

Включить информацию, хранимую в этой таблице, в таблицу продукции, имеющейся на складе. Таблица продукции упорядочена по возрастанию ключа. Элемент таблицы включает: шифр изделия (это ключ), наименование, количество (штук), цена (за штуку). Цену изделия брать из таблицы – прейскурант, элемент которой содержит: шифр изделия, цена (за штуку). Эта таблица также упорядочена по возрастанию шифров изделий. Для поиска элементов в таблице использовать:

а) последовательный поиск;

б) бинарный поиск.

5. Дополнить таблицу, построенную в лабораторной работе № 4 варианты 2.б, 2.в информацией о цене изделия. Цену изделия брать из таблицы – прейскурант, элемент которой содержит: шифр изделия, цена (за штуку). Эта таблица упорядочена по возрастанию шифров изделий. Упорядочить преобразованную таблицу по новому ключу – количество изделий методом:
а) прямого включения;

б) прямого обмена;

в) методом Шелла;

г) сортировки с использованием структуры дерево.

Для поиска элементов в таблице использовать:

а) последовательный поиск;

б) бинарный поиск.

6. Упорядочить таблицу, построенную в лабораторной работе № 4 вариант 3.а по убыванию процентов голосов, отданных командам – претендентам:
а) на первое место;

б) на последнее место;

в) на первую тройку.

Для упорядочения использовать метод:

а) быстрой сортировки;

б) сортировки с использованием структуры дерево;

в) методом Шелла.

7. Упорядочить таблицу, построенную в лабораторной работе № 5 варианты 3.б, 3.в по новому ключу – по возрастанию номеров команд, не вошедших в претенденты ни на первое, ни на последнее место.
   Для упорядочения использовать метод:

   а) быстрой сортировки;

   б) сортировки с использованием структуры дерево;

   в) методом Шелла.

Литература

1. Хусаинов Б.С. Структуры и алгоритмы обработки данных. – М.: Финансы и статистика, 2004
2. Вирт Н. Алгоритмы и структуры данных. – М.: Мир, 1989.
3. Кнут Д. Искусство программирования для ЭВМ. Основные алгоритмы. – М.: Мир, 1976.
4. Кнут Д. Искусство программирования для ЭВМ. Сортировка и поиск. – М.: Мир, 1978.
5. Подбельский В.В., Фомин С.С. Программирование на языке Си. – М.: Финансы и статистика, 2004-2006
6. Павловская Т.А Программирование на языке высокого уровня. – СПб.: Питер, 2006