МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение
высшего образования «Национальный исследовательский Нижегородский
государственный университет им. Н.И. Лобачевского»
Институт информационных технологий, математики и механики

Е.А. Кумагина
Н.Н. Чернышова

ВВЕДЕНИЕ В СТРУКТУРЫ ДАННЫХ

Учебно-методическое пособие

Рекомендовано методической комиссией ИТММ
для студентов ННГУ, обучающихся по направлению подготовки 09.04.03
«Прикладная информатика»

Нижний Новгород
2016

УДК 004.021(076.5)
ББК 3971я73-4
К 88

К 88 Кумагина, Е.А., Чернышова, Н.Н. Введение в структуры данных: учебно-методическое пособие / Е.А. Кумагина, Н.Н. Чернышова. – Нижний Новгород: Изд-во ННГУ, 2016. – 36 с.

Рецензент: к.ф.-м.н., доцент К.А. Баркалов

Пособие предназначено для студентов ИИТММ направления подготовки «Прикладная информатика», изучающих курсы «Программирование на языке С», «Основы алгоритмизации и алгоритмические языки», «Разработка Windows-приложений».

В данном пособии основное внимание уделяется таким структурам данных как списки и бинарные деревья. Рассмотрены вопросы реализации основных операций для работы с этими структурами данных и применение их для анализа арифметических выражений. По каждой теме разработаны лабораторные работы.

УДК 004.021(076.5)
ББК 3971я73-4
К 88

Оглавление

Введение

Структуры данных предназначены для удобного хранения и доступа к информации.

Простейшая структура данных это **массив**. Доступ к элементу массива происходит по номеру элемента через операцию квадратные скобки [ ]. Физически массив реализован в виде непрерывной области памяти. На этом свойстве массивов основывается связь массивов и указателей в языке Си [6].

```
//объявили массив, пока значений у элементов нет, но выделено sizeof(mass)=40 байт
памяти
int mass[10];
int *p; //объявили указатель
p=mass; //установили указатель на первый элемент массива, аналог p=&mass[0];
```

Теперь mass[5] и *(p+5) обращаются к одной и той же области памяти, в которую, кстати, до сих пор не занесли никакое значение.

Массив относят к **статическим структурам** данных. Под этим подразумевается, что память под массив выделяется один раз. Даже если рассматривать динамический массив, то объем памяти, выделенной при его создании остается неизменным до уничтожения массива.

Динамические структуры данных характеризуются тем, что элементы этой структуры расположены по произвольным адресам памяти. Для установления связи между элементами используются указатели.

Элемент динамической структуры состоит как минимум из полей двух типов:

1. информационного поля, в котором содержатся те данные, ради которых создается структура;
2. поля связок, в которых содержатся один или несколько указателей, связывающий данный элемент с другими элементами структуры.

Основными структурами данных являются **массивы, списки, деревья**. В данном пособии основное внимание уделяется спискам и бинарным деревьям.

Таблица 1

Сложность основных операций

Структура данных	**Поиск элемента**	**Вставка элемента**	**Удаление элемента**
Статический массив	O(n)	-	-
Динамический массив	O(n)	O(n)	O(n)
Линейный список	O(n)	O(1)	O(1)
Бинарное дерево	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$

1. Связный список

1.1. Виды связных списков

Если в программе необходимо хранить неупорядоченное множество элементов, число которых заранее известно, то логично использовать массив. Но если число элементов постоянно меняется, то в таких случаях для хранения применяют динамические структуры, которые представляют собой отдельные элементы, связанные с помощью ссылок.

По типу связности выделяют односвязные, двусвязные, XOR-связные, однонаправленные, кольцевые и некоторые другие списки.

В случае **односвязного списка** поле связок next, будет указывать на следующий элемент (поле содержит адрес следующего элемента). Такие списки иногда называют однонаправленными. В **двусвязном** списке добавляется еще поле prev, содержащее указатель на предыдущий элемент.

Если элемент не связан ни с каким другим, то в поле указателя на записывают значение NULL.

Указателем на первый элемент списка является особый элемент, называемый head.

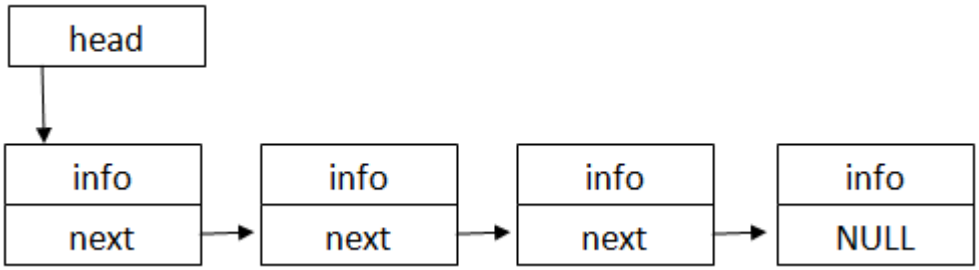

Рис. 1. Структура односвязного линейного списка

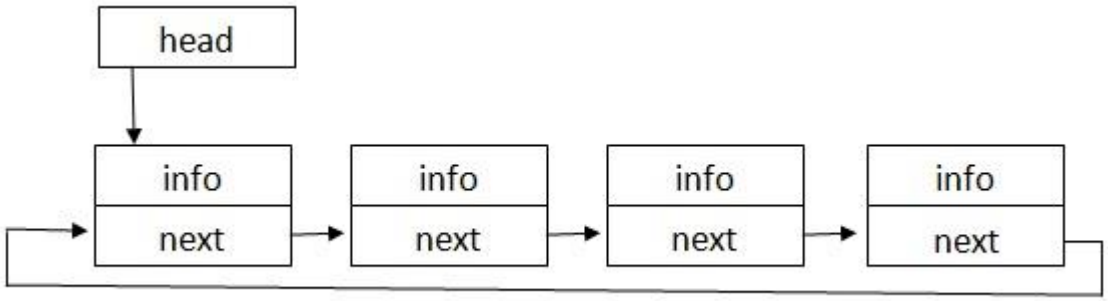

Рис. 2. Структура односвязного кольцевого списка

Опишем тип данных для хранения элементов односвязного списка:

```
struct zveno
{int info; //информационное поле
 zveno *next; //поле связки элементов, указатель на следующий элемент
 };
```

Классический **двунаправленный** или **двусвязный список** хранит в каждом элементе отдельно адреса предыдущего и следующего своего соседа, для хранения которых требуется два указателя:

```
struct zveno
{int info; //информационное поле
 zveno *next; //поле связки элементов, указатель на следующий элемент
 zveno *prev; //поле связки элементов, указатель на предыдущий элемент
};
```

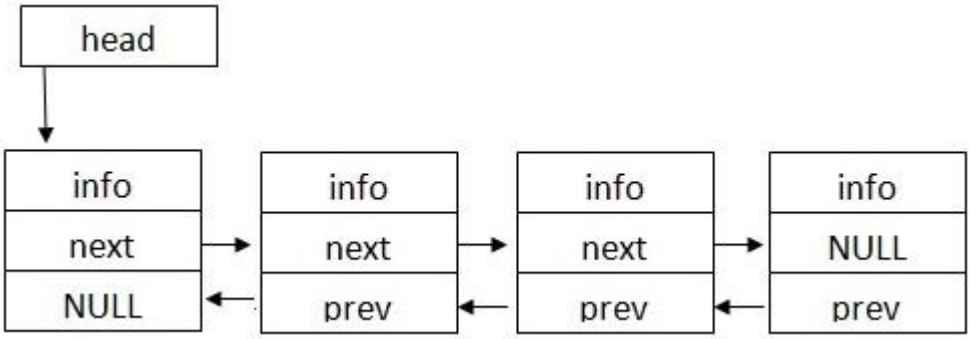

Рис. 3. Структура двусвязного линейного списка

XOR-связный список это структура данных, похожая на обычный двусвязный список, однако в каждом элементе хранящая только один адрес – результат выполнения операции XOR (исключающее или, сложение по модулю два) над адресами предыдущего и следующего элементов списка.

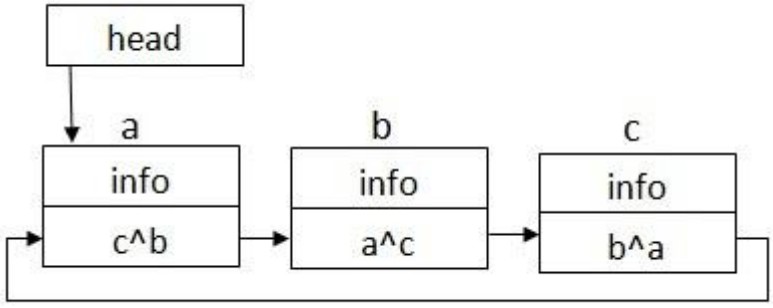

Рис. 4. Структура XOR списка

Таким образом, в нашем списке в поле next для элемента a хранится c^b, для b – a^c, для c – b^a.

Адрес следующего элемента вычисляется по двум предшествующим ему элементам. Чтобы узнать адрес элемента, следующего за элементом a, применим операцию XOR для поля next текущего элемента a и адреса предыдущего элемента c, получим (c^b)^c=b.

При инициализации XOR списка следует создать два пустых элемента (текущий и предыдущий), чтобы при добавлении первого элемента вычислить его адрес.

Тип данных для хранения элементов XOR списка может быть следующий:

```
struct zveno
 {int info; //информационное поле
  unsigned int address; //поле связки элементов для XOR-списка
 };
```

1.2. Операция добавления элемента в односвязный линейный список

Алгоритм добавления элемента в односвязный линейный список:

Шаг 1. Создать новый элемент списка (реализовано в функции `zveno* New()`).

Шаг 2. Определить место вставки.

Шаг 3. Включить элемент в список.

Рассмотрим один из вариантов реализации этого алгоритма.

Шаг 1. Создание элемента, не включенного в список и содержащего значение поля info равное 0.

```
zveno* New()
 {zveno* V;
  V= (zveno*)malloc(sizeof(zveno));
  V->info=0;
  V->next=NULL;
  return V;
 };
```

Шаг 2. Определение места вставки.

Самым простым вариантом является добавление элемента в начало списка.

```
zveno* AddFirst(zveno* head, int data)
 { zveno* v=New();//создали пустой элемент
   v->info=data; //присвоили значение
   v->next=head; //присоединили к списку
   return v; //возвращаем указатель на новую "голову" списка
}
```

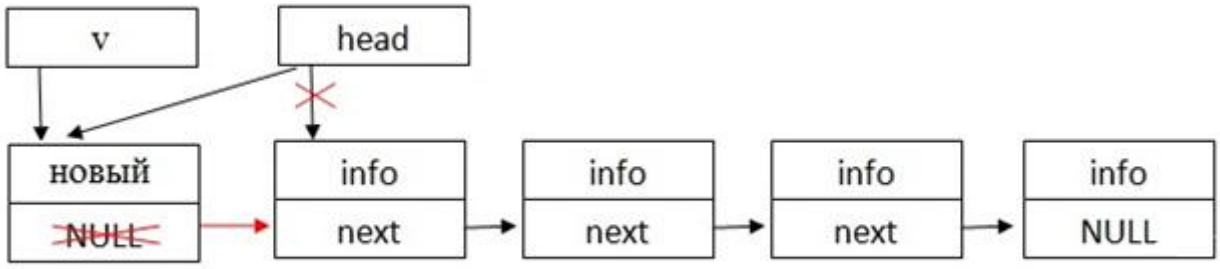

Рис. 5. Вставка элемента в начало списка

Использование в основной программе:

```
zveno* head = NULL;//объявили указатель на список
head=AddFirst(head,6); //добавили элемент со значением 6 в начало списка
```

Добавление элемента в конец списка.

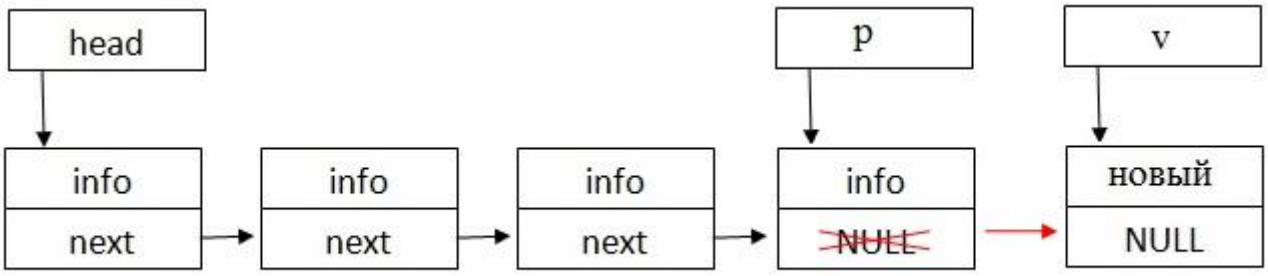

Рис. 6. Вставка элемента в конец списка

```
zveno* AddLast(zveno* head, int data)
 { zveno* v=New();
   zveno* p=head;//текущий указатель сначала указывает на первый элемент
   while(p->next!=NULL)// поиск последнего
     p=p->next;// перемещаем текущий указатель на следующий элемент
   p->next=v;//привязка нового элемента
   v->info=data;//занесение значения
   v->next=NULL;//элемент последний
      return head;
}
```

Использование в основной программе:

```
head=AddLast(head,7); //добавили элемент со значением 7 в конец списка
```

Вставка элемента в середину списка.

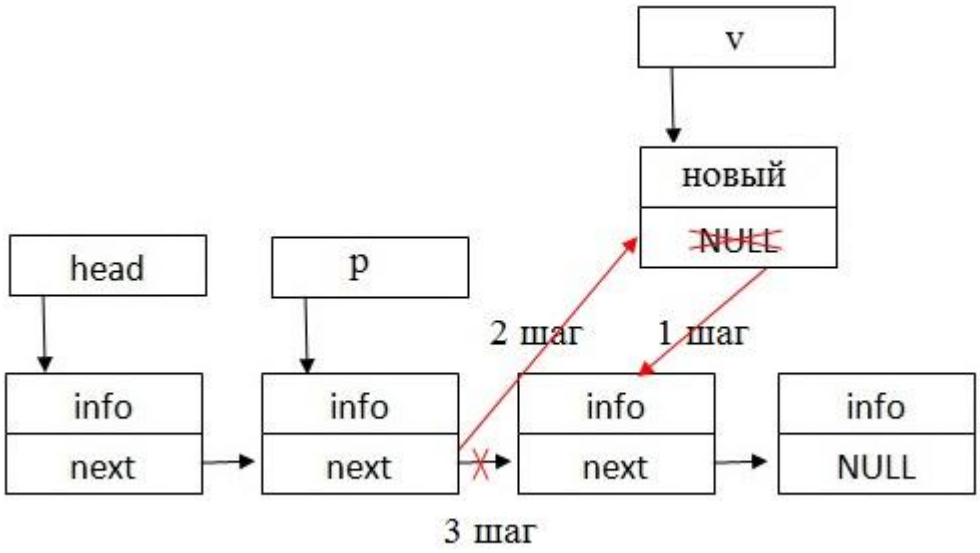

Рис. 7. Вставка элемента в середину списка

```
int AddNext(zveno* head, int value, int data)
{ zveno* p=head;//текущий указатель
  while (p->info<value && p->next!=0) //ищем место для вставки элемента
      p=p->next;// перемещаем текущий указатель)
  if(p->next==NULL) //если дошли до последнего элемента, значит условие для вставки
не выполнено
        return 0; //вставка неудачная
  zveno* v=New();
  v->info=data;
  v->next=p->next; //шаг 1
  p->next=v; //шаг 2
return 1; //вставка элемента прошла удачно
}
```

1.3. Операция удаления элемента из односвязного линейного списка

Алгоритм удаления элемента:

1. найти элемент, удовлетворяющий условию,
2. переставить указатель,
3. удалить элемент.

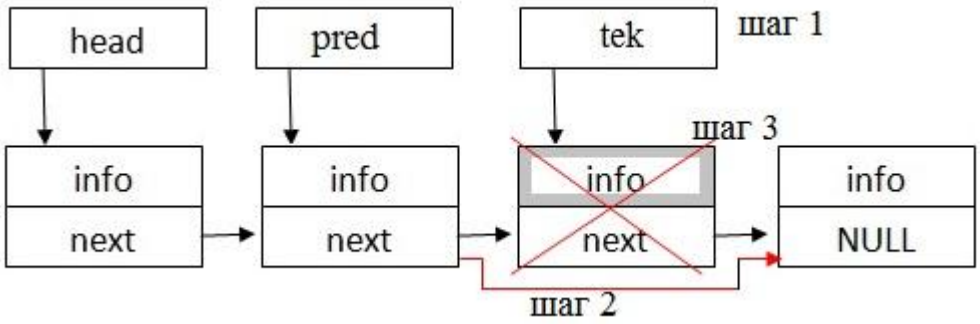

Рис. 8. Удаление элемента из середины списка

Отдельным случаем является удаление первого элемента, когда нужно переместить указатель головы списка.

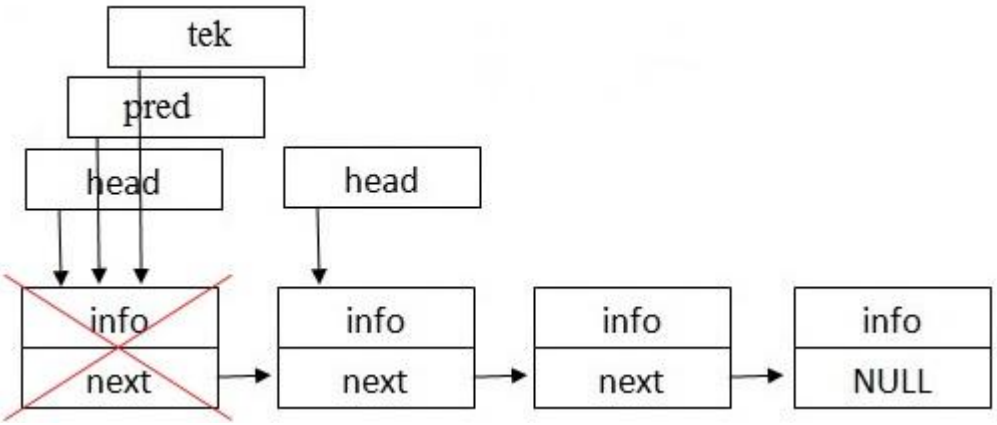

Рис. 9. Удаление первого элемента

```
zveno* Delete(zveno* head, int data)
 {
   zveno* tek=head, *pred=head ;//установили указатели на первый элемент
   if(head==NULL)//список пуст, удалять нечего
    return NULL;

   if(tek->info==data)// удаление первого
   {head=tek->next;
    delete tek;
    return head;
   }

   while(pred->next!=NULL)//идем по списку до последнего элемента
   {   if(tek->info==data)// ищем нужный элемент
       {pred->next = tek->next;//шаг 1
        delete tek;//шаг 2
         return head;
       }
    pred=tek;
    tek=tek->next;//перешли на следующий элемент
   }
// если дошли до этого места, то элемента, удовлетворяющего условию нет
   printf("\nno elem = %d\n", data);
       return head;
  };
```

1.4. Лабораторная работа № 1. Связные списки

Входные данные хранятся в текстовом файле и представляют собой набор чисел. Например, 3,7,2,9,11,4,6,33,88,34.

Задание 1. Хранение данных организовать в виде кольцевого однонаправленного списка.

Задание 2. Хранение данных организовать в виде двунаправленного списка.

Задание 3. Хранение данных организовать в виде XOR списка.

Для каждого вида списка реализовать следующие функции:

1. добавление элемента в начало, середину, конец списка;
2. удаления элемента, удовлетворяющиее заданному условию (например, по конкретному значению, удалить четные, удалить элементы, значение, которых >5 и т.п.);
3. подсчет числа элементов в списке;
4. печать списка в прямом и обратном порядке.

Демонстрацию работы программы провести следующим образом:

1. исходные данные считываются из файла;
2. сформированный список выводится в прямом и обратном порядке;
3. выводится число элементов списка;
4. к списку последовательно добавляются 3 элемента (в начало, середину, в конец), после каждого изменения список выводится на печать в прямом порядке;
5. из списка удаляются элементы, удовлетворяющие условию, список выводится на печать в прямом порядке;
6. из списка удаляются 3 элемента (из начала, из середины, последний элемент), после каждого изменения список выводится на печать в прямом порядке.

Примерный вид функции `main()`:

```
zveno* head = NULL;//указатель на список
head = ReadFromFile("data.txt");//вызов функции формирования списка
c = Count(head); // подсчет элементов
Print(head); // печать списка в прямом порядке
PrintRevers(head); // печать списка в обратном порядке
head = AddFirst(head,3); // добавить элемент 3 в начало списка
Print(head); // печать списка в прямом порядке
head = AddLast(head,3); // добавить элемент 3 в конец списка
Print(head); // печать списка в прямом порядке
head = Delete(head,8); // удалить элемент 8 из списка
Print(head); // печать списка в прямом порядке
```

2. Стек на основе связного списка

Стек – это такая структура данных, включение и исключение элементов из которой выполняются только с одной стороны, называемого вершиной стека. Основные операции над стеком – включение нового элемента (англ. push – заталкивать) и исключение элемента из стека (англ. pop – выскакивать). Полезными могут быть также вспомогательные операции: определение текущего числа элементов в стеке, очистка стека, чтение элемента с вершины стека [1].

Реализовать эту структуру данных можно на основе массива или линейного списка. Рассмотрим пример реализации основных операций со стеком на основе связного списка.

```
struct zveno
{int info;
 zveno *next;
 };
//------------------------------
int Print(zveno* v)
{
   if(v==NULL)
   printf("стек пуст\n");
   while(v!=NULL)
   { printf("%d ",v->info);
    v=v->next;
   }
   cout<<"\n";
   return 1;
 }
//------------------------------
zveno* Push(zveno* s, int a)
 {  zveno *v;
    v=new zveno;
    v->info=a;
    v->next=s;
    return v;
  };
 //------------------------------
 zveno* Pop(zveno* s)
 {zveno *v=s;
   s=s->next;
   delete v;
   return s;
  };
//------------------------------
  int Empty(zveno* s)
  {if(s==NULL)
    return 1;
    else return 0;
  }
```

Использование:

```
zveno *head=NULL;
if(Empty(head)) printf("empty\n");
else printf("not empty\n");
head =Push(head,1);
head =Push(head,2);
head =Push(head,3);
Print(head);
```

```
head =Pop(head);
Print(head);
if(Empty(head)) printf("empty\n");
else printf("not empty\n");
```

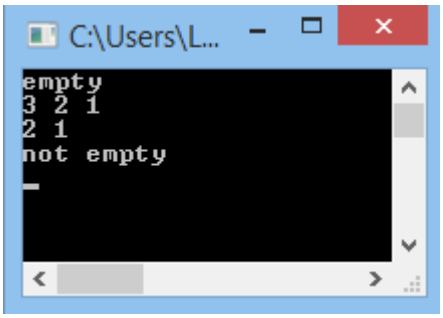

Рис. 10. Результат работы программы

Так как операции со стеком проводятся с определенным элементом, существует такое название этой структуры как LIFO (last in / first out последним пришел, первым ушел).

По противоположному принципу организуется работа с очередью элементов – первым пришел, первым ушел (FIFO: first in / first out). Для реализации работы очереди требуются минимальные изменения в программе.

```
zveno* Push(zveno* s, int a)
 {  zveno *v;
    v=New();
    v->info=a; v->next=NULL;
    zveno *q;
    while(q!=NULL)q=q->next;
    q->next=v;
   return s;
  };
```

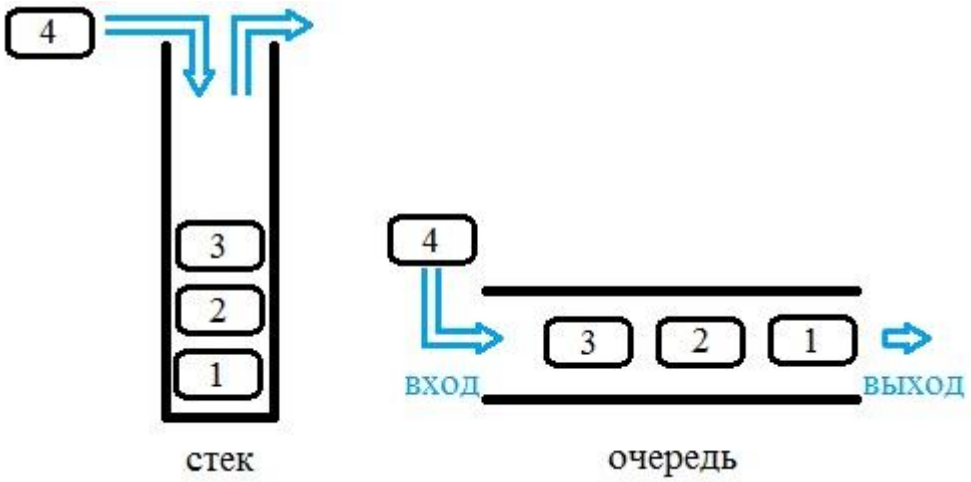

Рис. 11. Принцип работы стека и очереди

3. Обратная польская запись

Существуют 3 основных формы записи выражений: инфиксная, постфиксная, префиксная. Префиксы «пре», «пост» и «ин» относятся к относительной позиции оператора по отношению к обоим операндам.

Таблица 2

Формы записи выражений

инфиксная	a+b
постфиксная	ab+
префиксная	+ab

В инфиксной записи последовательность исполнения операций зависит не только от приоритетов операций, но и от расстановки скобок. Всем известен пример 2+2*2, результат которого 6, а не 8. Чтобы изменить порядок выполнения операций, надо воспользоваться скобками (2+2)*2.

3.1. Алгоритм перевода выражения в обратную польскую запись

Обратная польская нотация, предложенная польским математиком Я.Лукашевичем – форма записи математических выражений, в которой операнды расположены перед знаками операций.

Арифметическое выражение (1+2)*(3+4)-5 в обратной польской записи имеет вид 12+34+*5-.

Выражения в постфиксной записи не имеют скобок. Все операции выполняются в порядке их записи. Таким образом, вычисление выражения, записанного в обратной польской записи, может поводиться путем однократного просмотра выражения. Алгоритм перевода выражения из инфиксной записи в обратную польскую запись был предложен Дейкстрой.

Вводится понятие стекового приоритета операций.

Таблица 3

Приоритеты операций

Операция	Приоритет
(	0
)	1
+ -	2
* /	3
Возведение в степень	4

На вход алгоритма поступает строка выражения в инфиксной записи. Исходная строка символов просматривается слева направо. Очередной символ обрабатывается следующим образом:

1. числа переписываются в строку,
2. '(' помещается в стек,
3. ')' из стека извлекаются все символы до '(' и переписываются в строку,
4. знак операции op обрабатывается по следующему алгоритму:

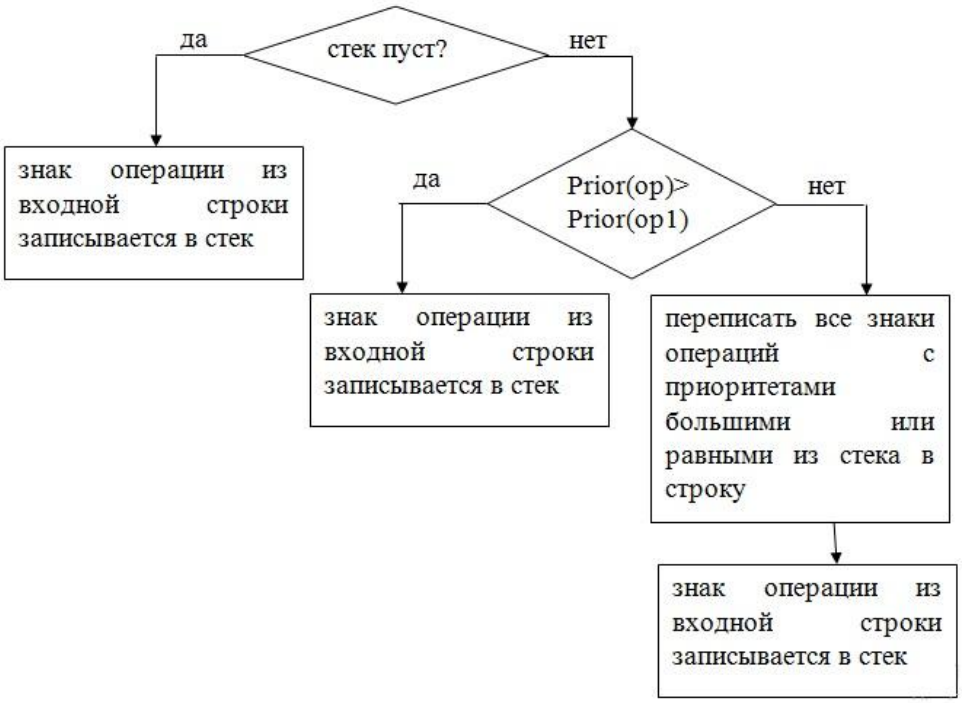

Рис. 12. Обработка знака операции

3.2. Вычисление результата выражения по обратной польской записи

Просматривая строку, анализируем очередной символ:

1. если число, то записываем его в стек;
2. если знак операции, то читаем 2 элемента из стека, выполняем математическую операцию, определяемую этим знаком, и заносим результат обратно в стек.

После просмотра всей строки в стеке должен оставаться один элемент, который является значением выражения.

3.3. Лабораторная работа № 2. Обратная польская запись

Входные данные хранятся в текстовом файле и представляют собой строку арифметического выражения со скобками.

Задание 1.

1. Выяснить корректность расстановки скобок. В случае некорректного выражения вывести сообщение об ошибке.
2. Перевести выражение в обратную польскую запись и записать в текстовый файл.
3. Вычислить результат выражения и записать в файл.

Задание 2. Написать генератор исходных данных.

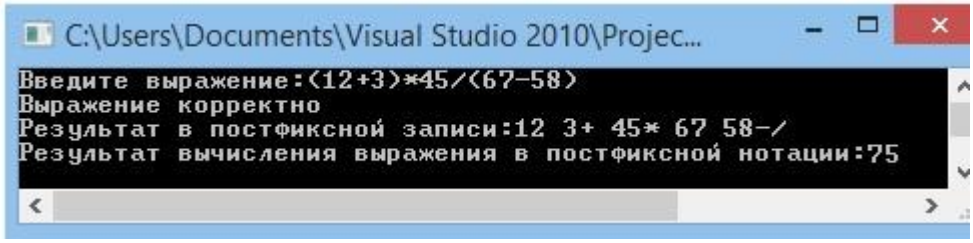

Рис. 13. Пример выполнения лабораторной работы

4. Создание и использование библиотеки

Для отработки навыков создания и использования библиотек динамической компоновки создадим два проекта. В проекте SumDll будет создаваться библиотека, состоящая из одной функции, суммирующей два числа int Sum(int a, int b). В проекте UseSum будем использовать нашу библиотечную функцию суммирования.

4.1. Проект для библиотечных функций

1. Создать проект приложения SumDll (консольное или Win 32) для описания библиотечных функций.

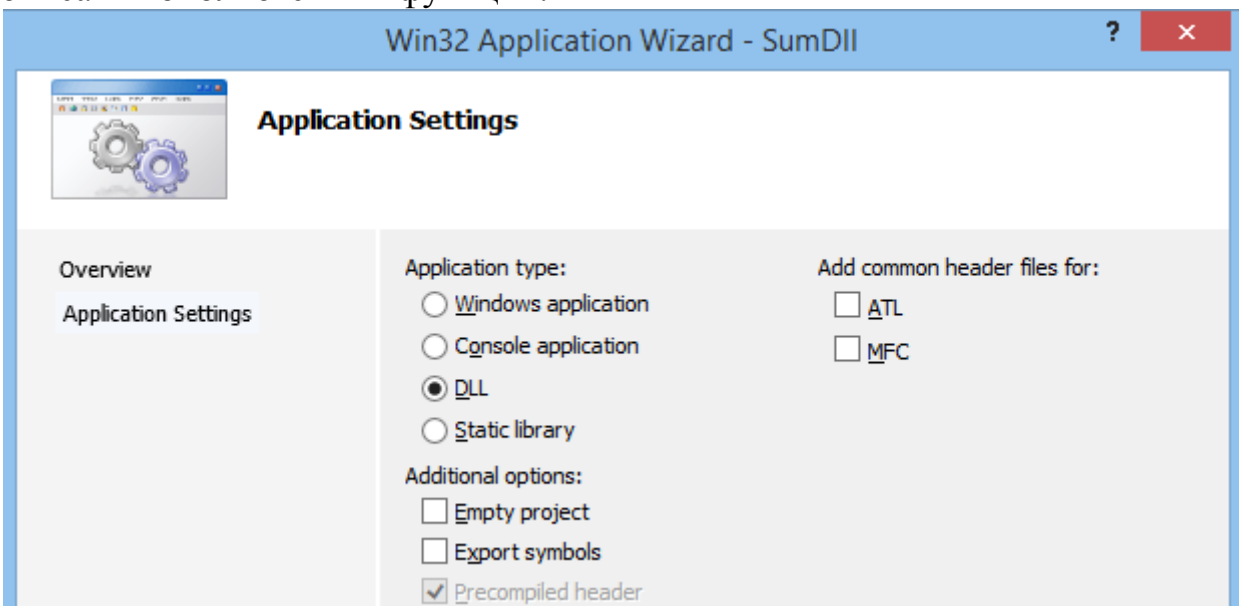

Рис. 14. Вид диалогового окна при создании приложения SumDll

2. Описать функции

Файл SumDll.cpp

```
#include "stdafx.h"
#include "Sum.h"
int Sum(int a, int b)
{return a+b;}
```

3. В h-файле для функций, предназначенных для «внешнего» использования, добавить к модификатор extern "C"_declspec(dllexport).

Файл SumDll.h

```
extern "C" _declspec(dllexport) int Sum(int a, int b);
```

4. Компилировать проект. В случае отсутствия ошибок Вы получите сообщение о невозможности запустить приложение, что вполне естественно, ведь мы создавали библиотеку, а не исполняемое приложение.

Из папки проекта SumDll нам потребуются 3 файла: h-файл, dll или lib, если мы перекомпилировали проект для получения библиотеки импорта.

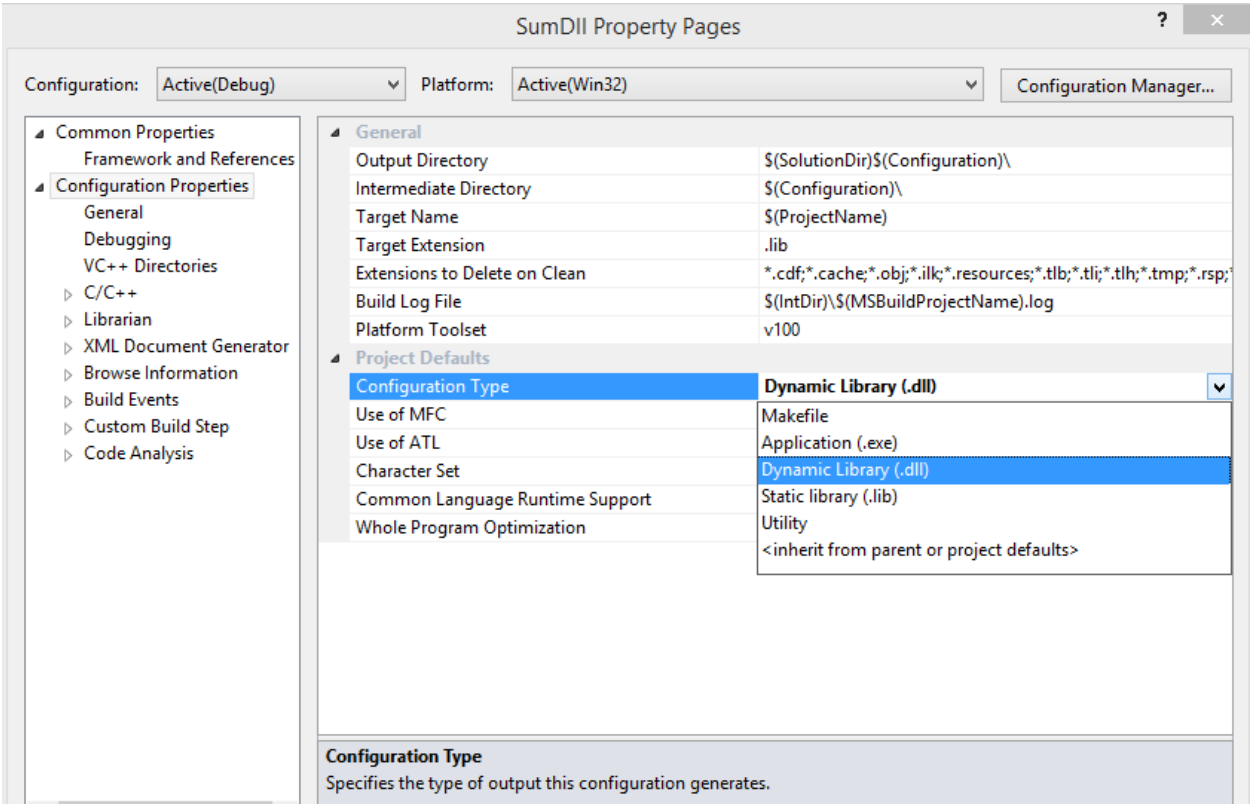

Рис. 15. Вид окна свойств проекта для создания DLL

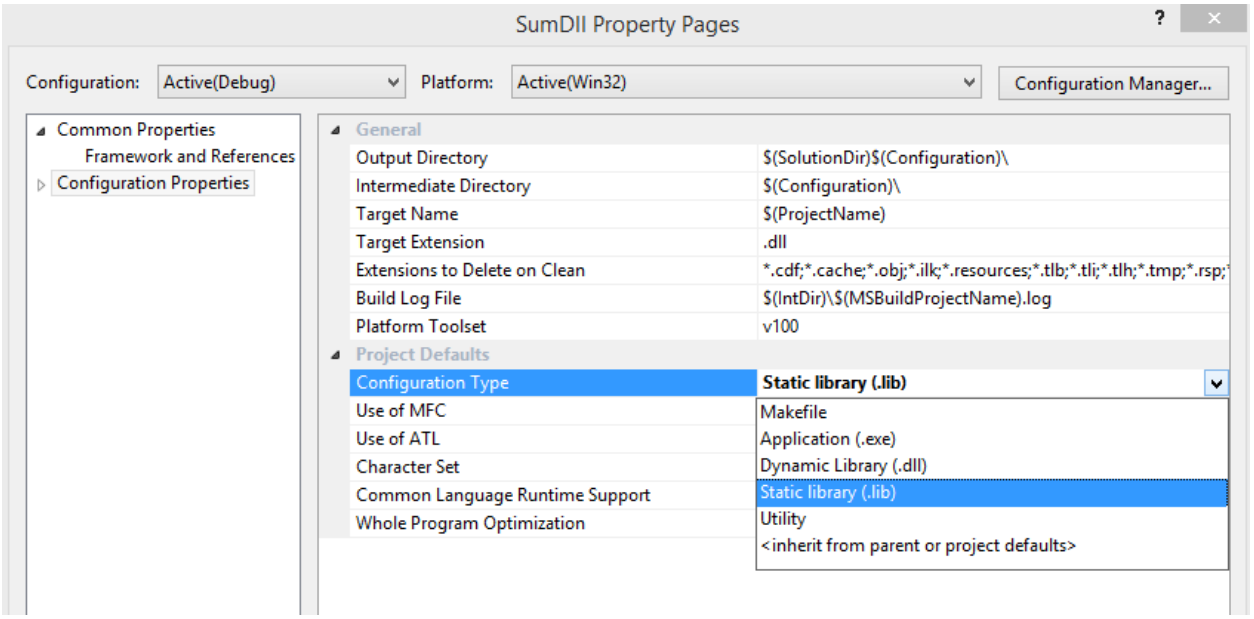

Рис. 16. Вид окна свойств проекта для создания библиотеки импорта

4.2. Использование библиотеки

Создать проект приложения UseSum (консольное или Win 32).

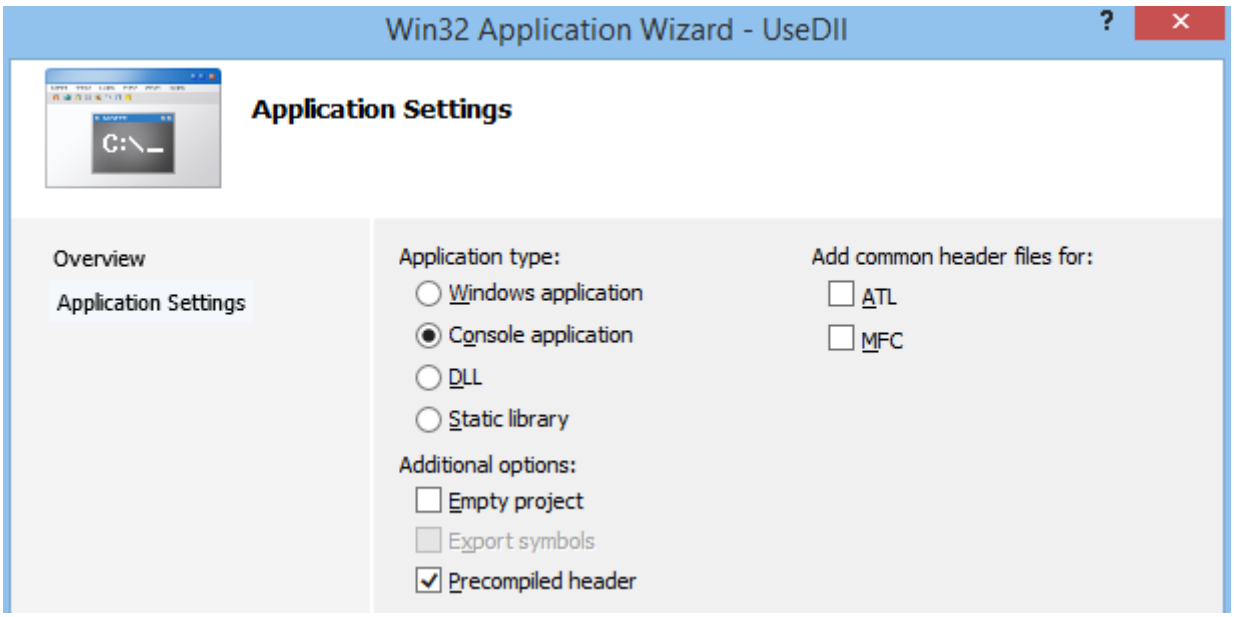

Рис. 17. Вид диалогового окна при создании приложения UseSum

В папку проекта перенести файлы библиотеки Sum.h, SumDll.dll и SumDll.lib.

Для того, чтобы воспользоваться функцией размещенной в библиотеке, надо сначала загрузить этот код в память, а потом сообщить программе, где этот код размещен. Делается это двумя способами: неявным (статическая компоновка) и явным образом (динамическая компоновка) [7].

Статическая компоновка

Первый способ – использовать директиву:

```
#pragma comment(lib,"SumDll.lib")
```

Второй способ – в пункт свойств проекта Linker/ Input/ Additional Dependencies добавить запись SumDll.lib

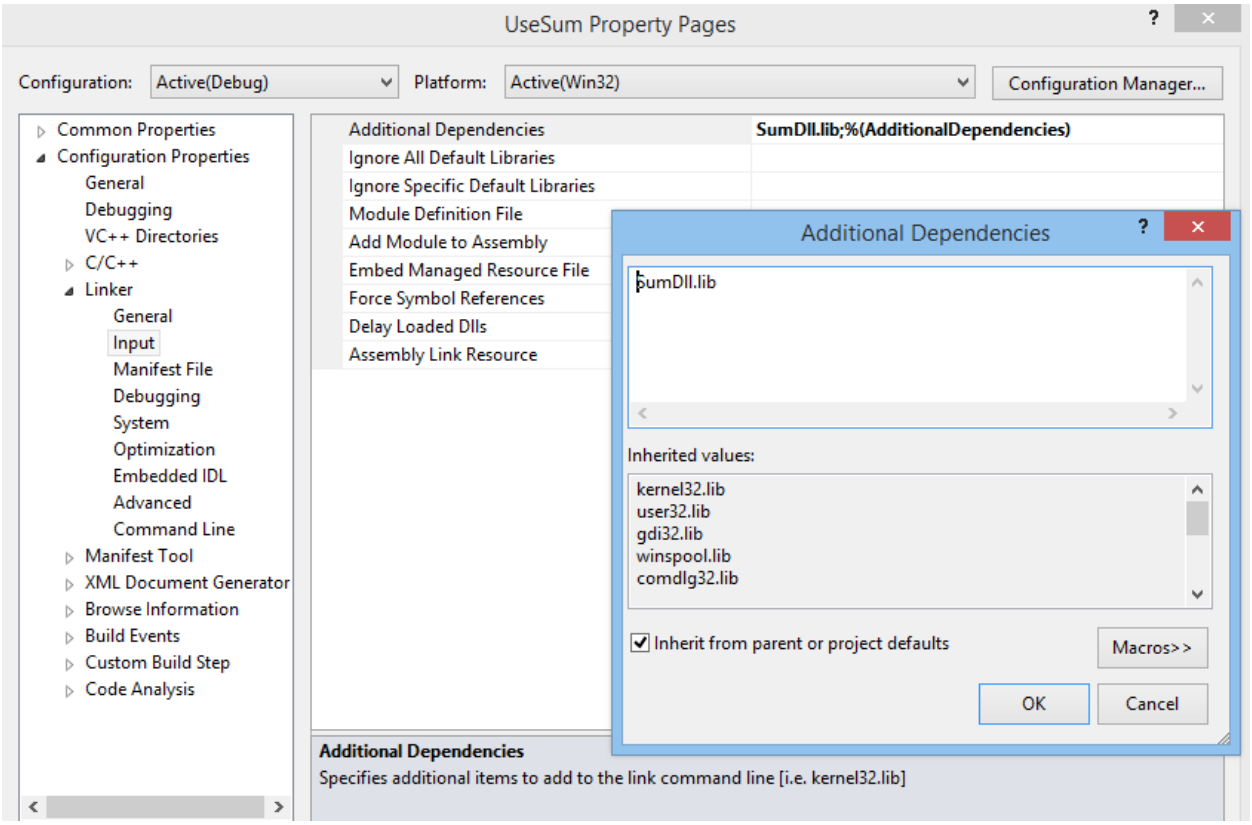

Рис. 18. Вид окна свойств проекта для подключения библиотеки

Добавить файл заголовков в файл UseSum.cpp

```
#include "Sum.h"
```

Использовать функцию

```
printf("%d + %d = %d",a,b,Sum(a,b));
```

Динамическая компоновка

1. Явно загрузить библиотеку, подготовленную нами на первом этапе.

```
#include <windows.h>

 HINSTANCE hMyDll; // переменная для сохранения дескриптора библиотеки
if((hMyDll=LoadLibrary("SumDll.dll"))==NULL) // пытаемся загрузить библиотеку
{cout<<"no dll\n";// библиотека не загружена
 getchar();
      return 1;
}
```

2. Для удобства определить тип – указатель на функцию нужного нам типа.

```
typedef int (*pMyFunc)(int, int);
```

3. Получим адрес нужной нам функции Sum

```
       pMyFunc pFunc=NULL; // указатель на функцию
         pFunc=(pMyFunc)GetProcAddress(hMyDll,"Sum");// получение адреса функции
```

4. Вызовем функцию для суммирования чисел 2 и 3

```
if(pFunc!=NULL)
        cout<<(*pFunc)(2,3);
        else cout<<"no func\n";
```

5. Отключить библиотеку

```
FreeLibrary(hMyDll);
```

4.3. Лабораторная работа № 3. Создание и использование библиотеки функций

Основой программы является л/р № 2.

Задание 1. Оформить функции, реализующие операции работы со стеком, в виде DLL.

Задание 2. Использовать статическую компоновку (через директиву или через установку опций проекта).

Задание 3. Использовать явную загрузку DLL.

5. Бинарное дерево

5.1. Основные определения

Дерево – это граф T = (V,E) без циклов, здесь V – множество вершин, E – множество ребер. Одна вершина $v \in V$ в дереве выделена, и называется **корневой вершиной**. **Лист** – вершина, не имеющая потомков. [3,8]

Поддерево. Поддеревом дерева T = (V,E) называется любое дерево T'=(V',E'), для которого выполнены условия:

1. $V' \subseteq V$, $V' \neq \varnothing$,
2. $E' \subseteq E$,
3. ни одна вершина из множества V\V' не является потомком вершины из множества V'.

Лес – множество непересекающихся деревьев.

Бинарное дерево – дерево, которое либо пусто, либо состоит из узла, называемого **корнем**, вместе с двумя двоичными деревьями, называемыми **левым поддеревом** и **правым поддеревом** корня. [4]

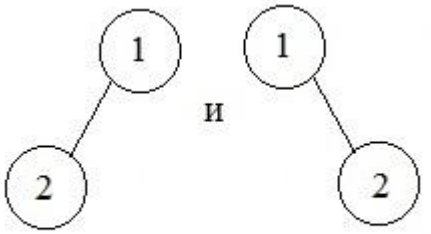

Рис. 19. Пример различных бинарных деревьев

Бинарное дерево **не является** частным случаем дерева. Пустое дерево – это бинарное дерево, а не дерево в обычном смысле. Также два дерева на рис. 19 одинаковы в обычном смысле, но являются различными бинарными деревьями, потому что у одного дерева нет правого поддерева, а у другого – левого.

Высотой дерева называется длина максимального пути от корня до вершины-листа.

Бинарное дерево называется **полным**, если присутствуют все листья одного уровня, и каждая внутренняя вершина имеет непустые правое и левое поддеревья.

Дерево называется **сбалансированным** тогда и только тогда, когда высоты двух поддеревьев каждой из его вершин отличаются не более чем на единицу.

Бинарное дерево является **сортирующим деревом** для элементов множества S, если значения в его узлах удовлетворяют правилу:

1. значение в узле u < значения в узле v для каждого узла u из левого поддерева узла v,

2. значение в узле w > значения в узле v для каждого узла w из правого поддерева узла v,
3. для любого элемента a∈S существует единственный узел v, такой, что значение в узле v равно a.

Обход дерева – это способ посещения узлов дерева, при котором каждый узел проходится только один раз.

Между лесом и бинарным деревом существует тесная связь: любой лес можно представить в виде бинарного дерева [1].

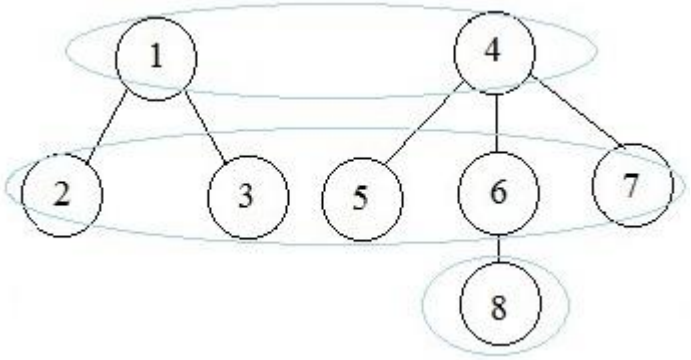

Рис. 20. Лес, состоящий из двух деревьев

Рассмотрим следующий лес из двух деревьев. Узлы, расположенные на одном уровне являются «братьями». На рис. 20 они обведены в овалы. Построим бинарное дерево, в котором соединим «братьев» и разорвем их связь с «родителем» для всех, кроме первого «ребенка».

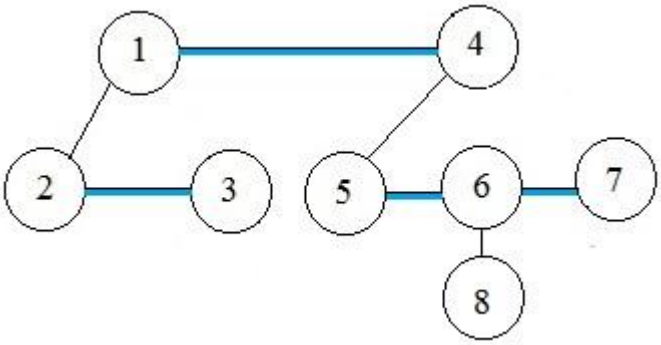

Рис. 21. Соответствующее бинарное дерево с корнем в вершине 1

5.2. Операции для работы с сортирующим бинарным деревом

Вершина дерева, как и узел любой динамической структуры, имеет две группы данных: полезную информацию и ссылки на узлы, связанные с ним. Для двоичного дерева таких ссылок будет две – ссылка на левое поддерево и ссылка на правое поддерево. В результате получаем структуру, описывающую вершину (предполагая, что полезными данными для каждой вершины является одно целое число):

```
struct Node
{int info;
 Node *left;
 Node *right;
 };
```

5.2.1. Алгоритм добавления вершины

Алгоритм добавления вершины в сортирующее дерево.
Шаг 0. Первый элемент становится коневым.
Шаг 1. Сравнить значение очередного элемента со значением в корне.
Шаг 2. Если значение меньше, то в качестве корня принять левого потомка корня, иначе правого потомка. Перейти на Шаг 1.
Шаг 3. Если потомка нет, то создать вершину и включить ее в дерево.
Пусть есть множество элементов {3,7,2,9,11,4,33,88,34,6}. Рассмотрим, как они добавляются сортирующее дерево.

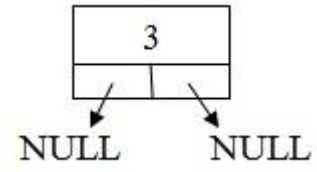

Рис. 22. Добавление первого узла

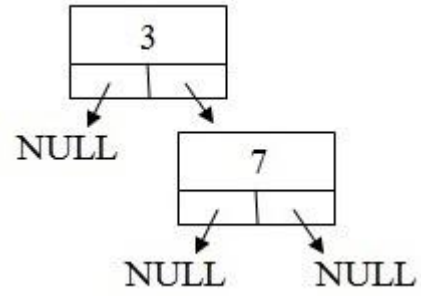

Рис. 23. Добавление второго узла

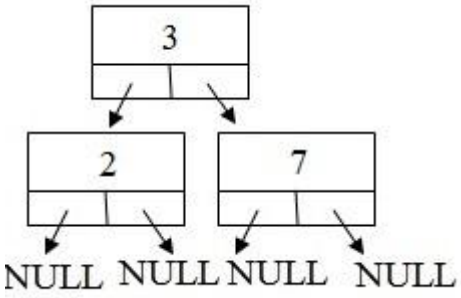

Рис. 24. Добавление третьего узла

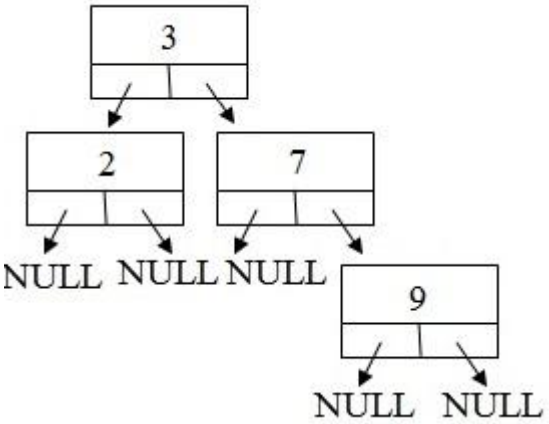

Рис. 25. Добавление четвертого узла

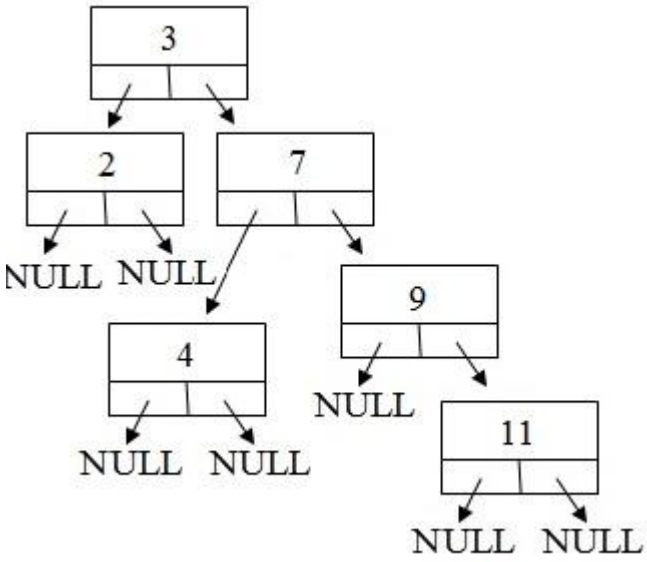

Рис. 26. Вид дерева после добавления шести узлов

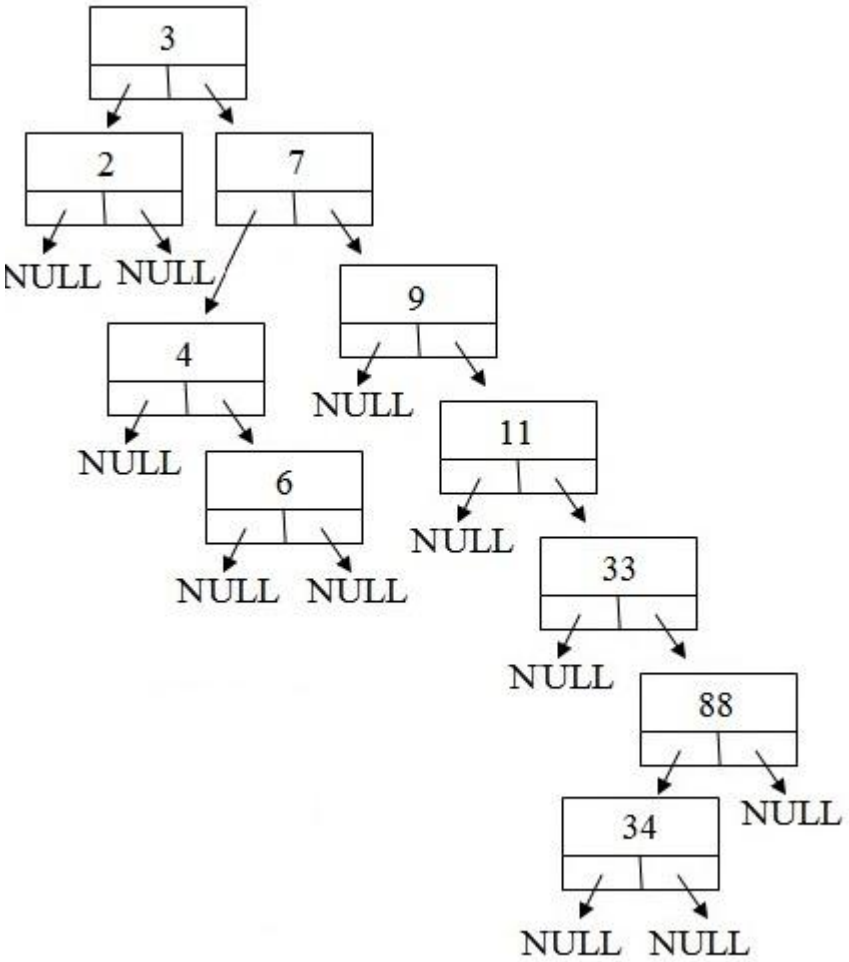

Рис. 27. Конечный вид дерева

5.2.2. Алгоритм удаления вершины

На подготовительном этапе необходимо найти узел дерева, содержащий заданное значение или убедиться, что такого нет. Этот процесс имеет в среднем $O(\log_2 n)$ операций сравнения.

Предполагаем, что вершина найдена. Рассмотрим варианты расположения вершины для удаления.

1. Лист.
2. Внутренняя вершина, имеющая одного поддерево.
3. Внутренняя вершина, имеющая оба поддерева. Сюда же входит случай удаления корня.

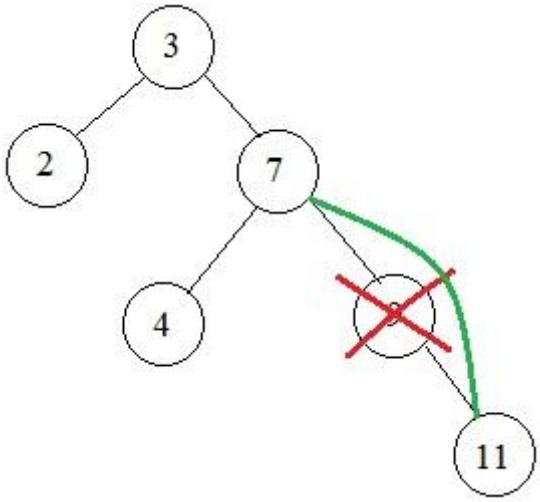

Рис. 28. Удаление вершины с одним поддеревом

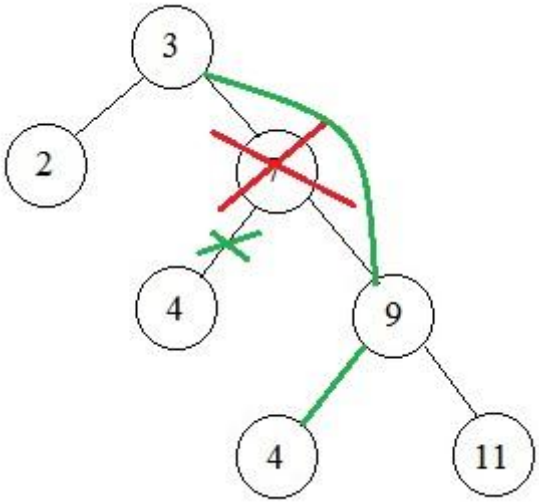

Рис. 29. Удаление вершины с двумя поддеревьями

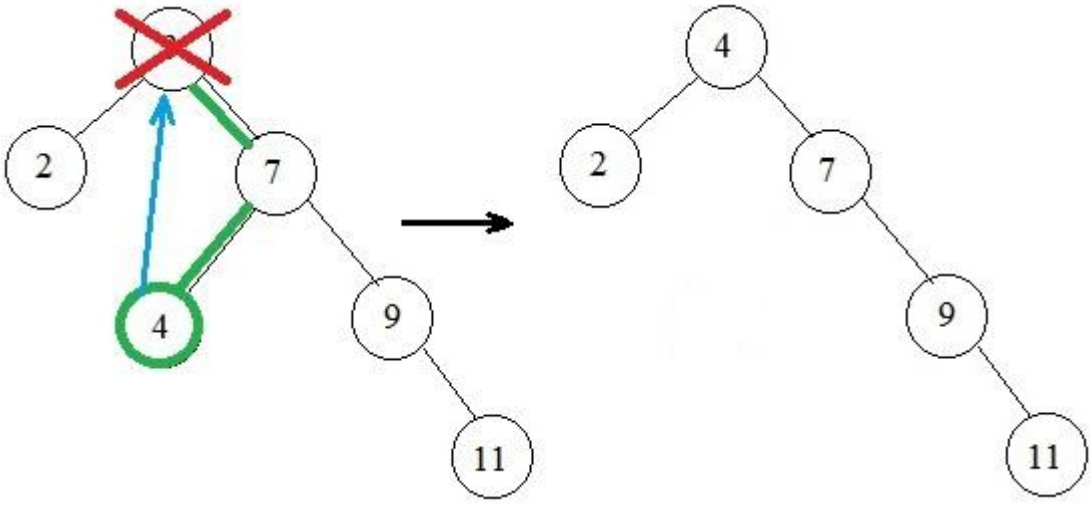

Рис. 30. Удаление корня

5.2.3. Реализация операций работы с бинарным деревом

Определим тип данных – вершина дерева:

```
struct Node
{int info;
 Node *left;
 Node *right;
 };
```

Процедура создания узла:

```
Node* New(int data)
 {Node* V;
  V=(Node*)malloc (sizeof Node);
  V->info=data;
  V->left=V->right=NULL;
  return V;
 };
```

Процедура добавления узла:

```
Node* Add(Node* v, int data)
 {
   if(v==NULL)// добавление нового
   return New(data);
   //поиск места для включения узла в дерево
   if(v->info>data)   v->right=Add(v->right, data);
   else v->left=Add(v->left, data);
   return v;
  };
```

Удаление узла:

1. Поиск вершины для удаления.

```
Node* Delete_node(Node* v, int key)
{
    Node* q;
// начинаем поиск узла по полю данных
    if (v==NULL) return v; // дерево пусто
    else if (key > v->info)
         v->left=Delete_node(v->left, key); // пошли по левой ветви
      else if (key < v->info)
            v->right=Delete_node(v->right, key); //пошли по правой ветви
         else
          {  q=v; //нашли и пометили
             if (q->right == NULL) // нет правого поддерева
                v=q->left;
             else
                if (q->left == NULL) // нет левого поддерева, см. рис.28
                   v=q->right;
                else
                   Del(&(q->left), &q); // есть оба поддерева, копируем данные
            delete q;    // Удаление
          }
       return v;
}
```

2. Вспомогательная процедура копирования данных при удалении вершины.

```
void Del(Node** r, Node** q)
{
    if ((*r)->right==NULL) // правого поддерева нет
    {
       (*q)->info=(*r)->info; // перенос данных
       (*q)->data=(*r)->data;
       *q=*r;
       (*r)=(*r)->left;
```

```
    }
    else
       Del(&((*r)->right), &(*q));
}
```

Подсчет узлов:

```
int Count(Node* v)
 {static  int n=0;

if ( !v ) return 0; // лист – окончание рекурсии
Count(v->left); // обход левого поддерева
n++;
Count(v->right); // обход правого поддерева
return n;
};
```

Печать дерева:

```
void Print(Node* v)
{ static  int n=0;//считаем уровень
if ( !v ) return;// лист – окончание рекурсии
n++;
Print(v->right); // обход правого поддерева
printf("(Level %d) %d, ",n , v->info); // вывод информации о вершине
Print(v->left); // обход левого поддерева
n--;
 }
```

Использование:

```
int _tmain(int argc, _TCHAR* argv[])
 {int i;
Node* V=NULL;
int mass[10]={3,7,2,9,11,4,33,88,34,6};
for(i=0;i<10;i++)
 V=Add(V,mass[i]);
Print(V);
printf("\ncount=%d , h=%d",Kol(V), Height(V));
getchar();
return 0;
}
```

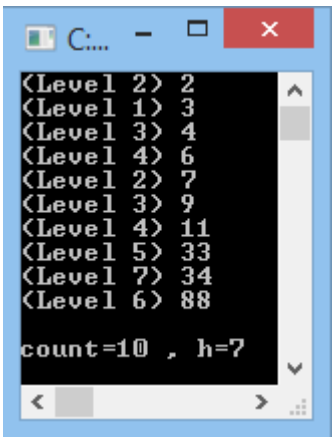

Рис. 31. Результат работы программы

Проблемы

Бинарное дерево не является сбалансированным деревом. Например, при добавлении элементов, которые увеличиваются на единицу, бинарное дерево вырождается в линейный список. Таким образом теряются все преимущества бинарного дерева – более быстрый поиск элементов.

Примерами сбалансированных деревьев являются АВЛ-деревья и черно-красные деревья [2].

5.3. Обходы бинарного дерева

Обход дерева – это способ последовательного посещения узлов дерева, при котором каждый узел посещается только один раз [5].

Таблица 5

Виды обходов дерева

Вид обхода	Основа нерекурсивной реализации	Разновидности
В ширину	очередь	
В глубину	стек	прямой
		поперечный
		обратный

5.3.1. Обходы в глубину

Существует три способа обхода дерева в глубину:

1. прямой (корень-лево-право), Pre_order
2. поперечный (лево-корень-право), In_order
3. обратный (лево-право-корень), Post_order.

Рекурсивный алгоритм прямого обхода.

Шаг 1. Посетить корень и вывести значение.

Шаг 2. Обойти левое поддерево.

Шаг 3. Обойти правое поддерево.

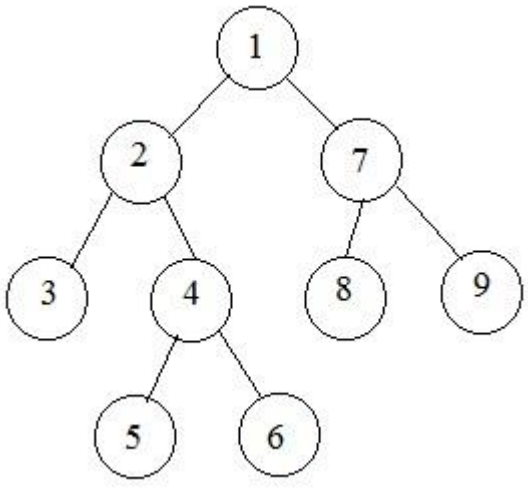

Рис. 32. Порядок обхода вершин при прямом обходе

Для дерева, соответствующего множеству {3,7,2,9,11,4,33,88,34,6}, результат прямого обхода 3,2,7,4,6,9,11,11,88,34.

Рекурсивный алгоритм поперечного обхода.

Шаг 1. Обойти левое поддерево.

Шаг 2. Посетить корень и вывести значение.

Шаг 3. Обойти правое поддерево.

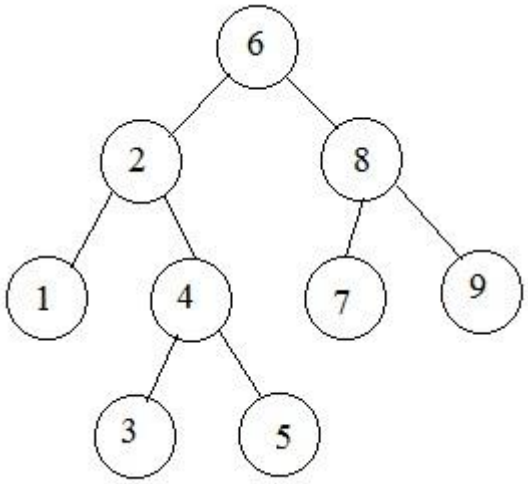

Рис. 33. Порядок обхода вершин при поперечном обходе

Для дерева, соответствующего множеству {3,7,2,9,11,4,33,88,34,6}, результат прямого обхода 2,3,4,6,7,9,11,33,34,88.

Рекурсивный алгоритм обратного обхода.

Шаг 1. Обойти левое поддерево.

Шаг 2. Обойти правое поддерево.

Шаг 3. Посетить корень и вывести значение.

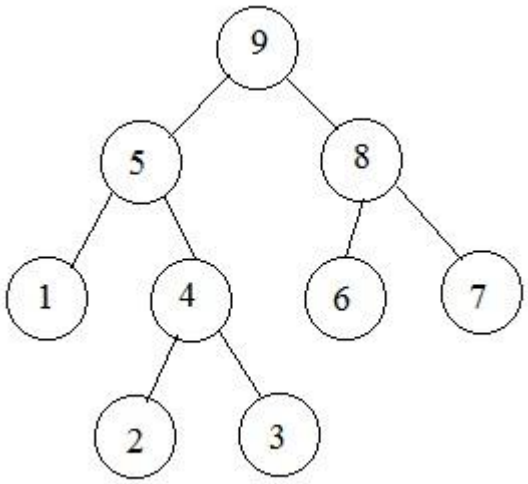

Рис. 34. Порядок обхода вершин при обратном обходе

Для дерева, соответствующего множеству {3,7,2,9,11,4,33,88,34,6}, результат обратного обхода 2,6,4,3,88,33,11,9,7,3.

5.3.2. Удаление бинарного дерева

Процедуру обратного обхода можно использовать для удаления дерева. Действительно, узлы посещаются слева направо и снизу вверх. Сначала посещаются листья, и их можно удалять не нарушая целостности дерева.

```
void DelTree(Node **v)
{if(*v)
      { DelTree(&((*v)->left));
        DelTree(&((*v)->right));
        free(*v);
      }
}
```

5.3.3. Обход в ширину

Алгоритм обхода дерева **в ширину** основан на структуре данных **очередь.**

Шаг 0. Поместить в очередь корень дерева.

Шаг 1. Изъять из очереди очередную вершину. Поместить в очередь ее дочерние вершины по порядку слева направо (справа налево).

Шаг 2. Если очередь пуста, то конец обхода, иначе перейти на Шаг 1.

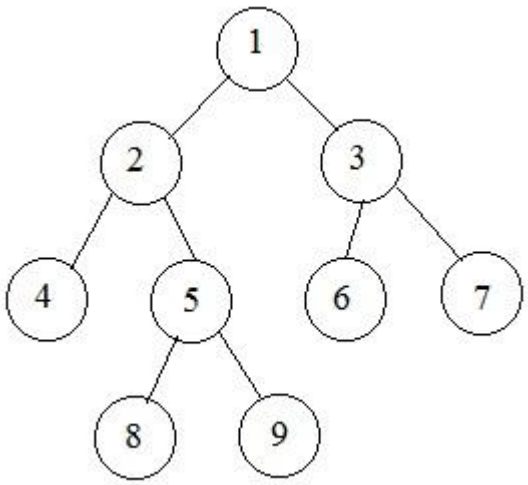

Рис. 35. Порядок обхода вершин при обходе в ширину

Для дерева, соответствующего множеству {3,7,2,9,11,4,33,88,34,6}, результат прямого обхода 3,2,7,4,9,6,11,33,88,34.

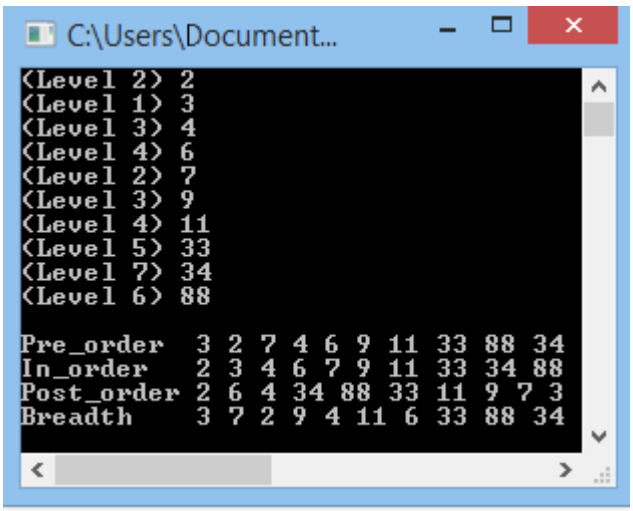

Рис. 36. Результат работы программы обходов дерева, соответствующего рис. 27

5.4. Лабораторная работа № 4. Сортирующее бинарное дерево

Входные данные хранятся в текстовом файле и представляют собой наборы букв, слов, чисел.

Задание 1. Написать функцию для построения сортирующего бинарного дерева по файлу данных.

Задание 2. Применить к полученному дереву процедуры обходов, результаты сохранить в файле.

Задание 3. Написать функцию, подсчитывающую высоту бинарного дерева.

6. Построение дерева для арифметического выражения

6.1. Алгоритм построения бинарного дерева простого арифметического выражения

Бинарное дерево является одним из вариантов хранения арифметического выражения. Листья такого дерева содержат операнды (числа или переменные), а внутренние вершины и корень – знаки операций.

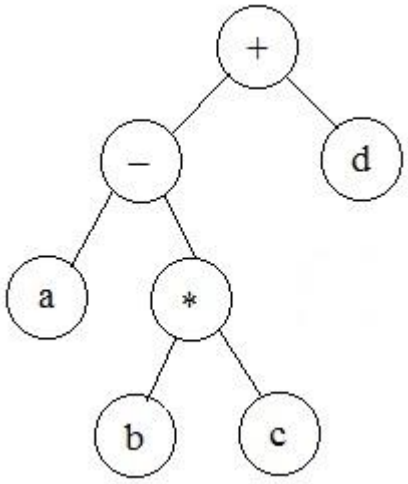

Рис. 37. Бинарное дерево для выражения a - b * c + d

Для начала рассмотрим самый простой вариант выражения, введя следующие ограничения:

1. в выражении могут присутствовать только однозначные целые числа,
2. рассматриваются операции: +, - ,* , /.
3. выражение не содержит скобок и унарных знаков (например, выражение 6+5, следует писать в виде 0-6+5).
4. предполагается, что выражение записано верно.

Помним, что порядок выполнения операций в выражении определяется **приоритетом операций** – первыми выполняются операции с более высоким приоритетом.

Предполагаем, что арифметическое выражение хранится в строке длины N.

Рекурсивный алгоритм построения дерева арифметического выражения следующий:

Шаг 1. В строке найти последнюю операцию с наименьшим приоритетом. Пусть ей соответствует позиция k.

Шаг 2: Создать узел, содержащий знак этой операции, и рекурсивно сформировать левого и правого потомка на Шаге 3.

Шаг 3: Левый потомок определяется подстрокой строки от k+1до N. Правый потомок определяется по подстроке от 1 до k-1.

Если длина подстроки равна 1, то надо создать новую вершину с числовым или буквенным значением.

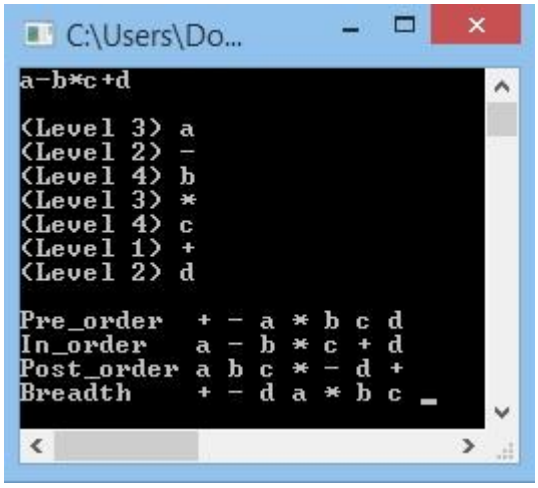

Рис. 38. Результат работы программы обходов дерева арифметического выражения

6.2. Алгоритм построения дерева для арифметического выражения со скобками

Скобки нужны для смены порядка выполнения операций при инфиксной записи. В дерево скобки не заносятся, поэтому восстановить инфиксную запись не удастся. Зато можно проверить корректность результата используя постфиксную запись и алгоритм из лабораторной работы №2.

Внесем изменения в алгоритм из п. 6.1.:

На Шаге 1 при поиске операции с наименьшим приоритетом надо пропускать выражения в скобках. Простейший способ реализации этого условия – ввести счетчик открытых скобок, который инициализирован нулем.

Если при поиске последней операции с наименьшим приоритетом встречается открывающая скобка, то счетчик увеличивается на 1, при встрече закрывающей скобки счетчик уменьшается на 1. Таким образом, операциям, расположенные вне скобок, соответствует нулевое значение счетчика.

Если ни одной операции не найдено, то мы нашли выражение, ограниченное скобками, и его анализ осуществляется рекурсивным вызовом. Не забываем отбросить скобки.

6.3. Вычисление результата выражения по дереву

Основой алгоритма является последовательное вычисление выражений, содержащих по 2 листа, и замена их родительской вершины на значение выражения. Недостатком этой реализации процесса вычисления значения выражения является то, что исходное дерево разрушается.

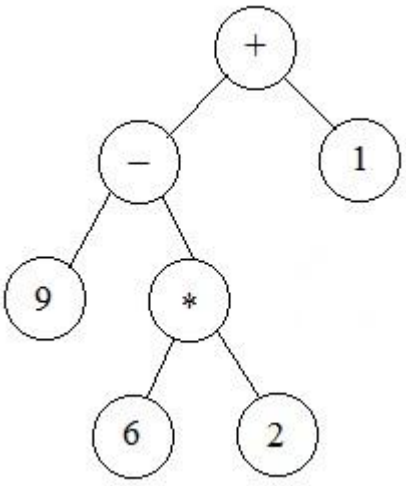

Рис. 39. Исходное дерево арифметического выражения 9-6*2+1

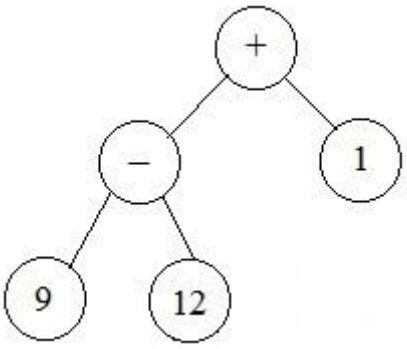

Рис. 40. Вид дерева после выполнения первой операции

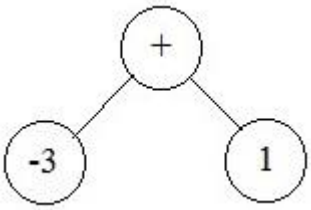

Рис. 41. Вид дерева после выполнения второй операции

Рис. 42. Окончательный вид дерева

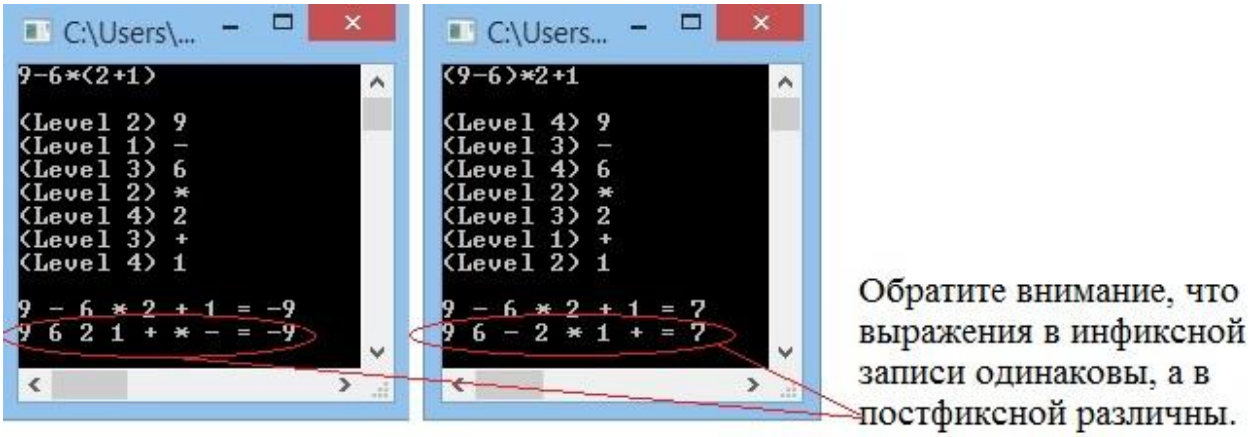

Рис. 43. Результат работы программы вычисления арифметического выражения со скобками

6.4. Лабораторная работа № 5. Бинарное дерево простого арифметического выражения

Входные данные хранятся в текстовом файле и представляют собой арифметическое выражение без скобок. Например, 1+2+3*4*5/6-7-8

Задание 1. Написать функцию для построения бинарного дерева арифметического выражения.
Задание 2. Применить к полученному дереву различные обходы. Результирующие выражения записать в файл.
Задание 3. Вычислить результат выражения по дереву.

6.5. Лабораторная работа № 6. Бинарное дерево арифметического выражения со скобками

Входные данные хранятся в текстовом файле и представляют собой арифметическое выражение со скобками. Например, (1+2)+3*4*5/6-(7-8).
Задание 1. Написать функцию для построения бинарного дерева арифметического выражения.
Задание 2. Применить к полученному дереву различные обходы. Результирующие выражения записать в файл.
Задание 3. Вычислить результат выражения по дереву.

Список литературы

1. Кнут Д. Искусство программирования. Т.1 – М.: Мир, 2004.
2. Кормен Т., Лейзерсон Ч., Ривест Р – Алгоритмы: построение и анализ. – М.: Издательский дом "Вильямс", 2005.
3. Кристофидес Н. Теория графов: алгоритмический подход. – М.: Мир, 1978.
4. Круз Р. Структуры данных и проектирование программ. –М.: БИНОМ. Лаборатория знаний, 2014.
5. Липский В. Комбинаторика для программистов. – М.: Мир, 1988.
6. Подбельский В.В., Фомин С.С. Программирование на языке Си: Учеб.пособие. – 2-е доп.изд. – М.: Финансы и статистика, 2004.
7. Румянцев П. В. Азбука программирования в Win32 API. – М.: Горячая Линия – Телеком, 2004.
8. Свами М. Н., Тхуласираман К. – Графы, сети и алгоритмы. – М.: Мир, 1984.

Елена Александровна **Кумагина**
Наталья Николаевна **Чернышова**

ВВЕДЕНИЕ В СТРУКТУРЫ ДАННЫХ

Учебно-методическое пособие

Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

603950, Нижний Новгород, пр. Гагарина, 23.