МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный электротехнический
университет «ЛЭТИ» им. В.И. Ульянова (Ленина)

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ, ПРАКТИЧЕСКИМ ЗАНЯТИЯМ И КУРСОВОЙ РАБОТЕ

Учебно-методическое пособие

Санкт-Петербург
2016

УДК 004.02

Методические указания к лабораторным работам, практическим занятиям и курсовой работе по дисциплине «Алгоритмы и структуры данных»»: учеб.-метод. пособие / сост.: С.А Ивановский , Т.Г. Фомичева , О.М. Шолохова.. СПб. 2017. 88 с.

Представлены материалы по дисциплине «Алгоритмы и структуры данных». Описываются такие структуры данных, как иерархический список, стек, очередь, дек, дерево, лес, бинарное дерево и алгоритмы их обработки. Рассматриваются вопросы программирования рекурсивных алгоритмов, рекурсивной обработки иерархических списков, задачи кодирования и поиска. Приводятся примеры типовых заданий и возможные варианты их выполнения. Каждый раздел содержит список предлагаемых студентам индивидуальных заданий.

Предназначено для студентов направлений «Программная инженерия» и «Прикладная математика и информатика».

Одобрено
Методической комиссией факультета компьютерных технологий и информатики
в качестве учебно-методического пособия

© СПбГЭТУ «ЛЭТИ», 2016

ВВЕДЕНИЕ

Дисциплина «Алгоритмы и структуры данных» входит в учебные планы подготовки бакалавров по направлениям 01.03.02 – Прикладная математика и информатика» и 09.03.04 – «Программная инженерия» по профилю «Разработка программно-информационных систем». Рабочая программа дисциплины предусматривает лекционные, лабораторные и практические занятия с преподавателем, а также большой объем самостоятельной работы студентов, в т.ч. выполнение курсовой работы.

Учебно-методическое пособие поддерживают всю практическую деятельность студента при изучении дисциплины, включая практические и лабораторные занятия, а также выполнение курсовой работы.

Цель практических занятий – помочь студентам при подготовке к выполнению курсовой и лабораторных работ. На практических занятиях преподаватель на примерах поясняет лекционный материал, рассматривает постановку задачи и общую схему ее решения, возможные варианты реализации алгоритмов на языке C++. Студент выбирает по каждому разделу индивидуальное задание из предложенного в пособии списка и реализует его в рамках часов, отведенных на лабораторные занятия и самостоятельную работу. Задания необходимо выполнять в соответствии с требованиями, указанными в каждом разделе пособия и с учетом рекомендаций преподавателя. Методические указания содержат краткие теоретические сведения по следующим пяти разделам дисциплины:

1. Программирование рекурсивных алгоритмов
2. Рекурсивная обработка иерархических списков
3. Линейные структуры данных: стек, очередь, дек.
4. Программирование алгоритмов с бинарными деревьями
5. Бинарные деревья поиска и алгоритмы сжатия

Обсуждаются возможные варианты выполнения типовых заданий. Приложение к каждому разделу содержит список предлагаемых студентам индивидуальных заданий.

Информационные технологии (операционные системы, программное обеспечение общего и специализированного назначения, информационные справочные системы) и материально-техническая база, используемые при осуществлении образовательного процесса по дисциплине, соответствуют требованиям федерального государственного образовательного стандарта высшего образования.

Для обеспечения образовательного процесса по дисциплине используются следующие информационные технологии:

1. Операционные системы: Microsoft Windows 10 Version 1607 Education x86, Ubuntu Desktop 16.04.1 x64.

2. Программное обеспечение общего и специализированного назначения: Microsoft Visual Studio Professional 2015, Microsoft Visual Studio Express 2010 Express,Qt 5.5.1, TDM64-GCC 5.1.0,Microsoft Office Профессиональный плюс 2007, LibreOffice 5.1.4.2 и LibreOffice 5.1 Help Pack (Russian),WPS Office (9.1.0.5240);

3. Информационные справочные системы: международная ассоциация сетей «Интернет», электронные библиотечные системы и ресурсы удаленного доступа библиотеки СПбГЭТУ «ЛЭТИ».

Образовательный процесс обеспечивается на следующей материально-технической базе (один из трех вариантов):

1) Персональный компьютер (системный блок RAMEC STORM, монитор LG L1953S, клавиатура, мышь, ИБП APC), персональный компьютер (системный блок RAMEC STORM, монитор LG L1953S, клавиатура, мышь, ИБП APC), сервер (RAMEC, монитор LG L1953S, клавиатура, мышь, ИПБ APC), проектор Mitsubishi XD430U, экран проекционный настенный;

2) Персональный компьютер (системный блок RAMEC STORM, монитор LG L1953S, клавиатура, мышь, ИБП APC), персональный компьютер (системный блок RAMEC STORM, монитор LG L222WS, клавиатура, мышь, ИБП APC), принтер HP Laser Jet, принтер HP Color Laser Jet, сервер SuperMicro, сканер планшетный HP LaserJet, проектор Mitsubishi XD430U, экран проекционный настенный подпружиненный ScreenMedia Goldview 213x213 MW;

3) Персональный компьютер (системный блок Hewlett-Packard, монитор Samsung SyncMaster 913N, клавиатура, мышь), персональный компьютер (системный блок Hewlett-Packard, монитор Samsung SyncMaster E2220, клавиатура, мышь), проектор Vivitek D555, компьютер-сервер (системный блок Universal KOMPUMIR, монитор Samsung SyncMaster 913N, клавиатура, мышь), кондиционер QuattroClima Industriale QA-RWD.

РАЗДЕЛ 1. ПРОГРАММИРОВАНИЕ РЕКУРСИВНЫХ АЛГОРИТМОВ

1.1. Цель и задачи

При освоении этого раздела студент должен познакомиться с основными понятиями и приемами рекурсивного программирования, получить навыки программирования рекурсивных процедур и функций на языке программирования C++.

Рекурсивным называется объект, содержащий сам себя или определенный с помощью самого себя.

Мощность рекурсии связана с тем, что она позволяет определить бесконечное множество объектов с помощью конечного высказывания. Точно так же бесконечные вычисления можно описать с помощью конечной рекурсивной программы. Рекурсивные алгоритмы лучше всего использовать, когда решаемая задача, вычисляемая функция или обрабатываемая структура данных определены с помощью рекурсии.

Если процедура (функция) P содержит явное обращение к самой себе, она называется прямо рекурсивной. Если P содержит обращение к процедуре (функции) Q, которая содержит (прямо или косвенно) обращение к P, то P называется косвенно рекурсивной.

Многие известные функции могут быть определены рекурсивно. Например факториал, который присутствует практически во всех учебниках по программированию, а также наибольший общий делитель, числа Фибоначчи, степенная функция и др.

Рассмотрим использование рекурсии при программировании алгоритмов вычисления степенной функции.

Вычисление степенной функции

Степенная функция $f(a,n)=a^n$ (основание степени a – например, вещественное число, а показатель степени n – целое неотрицательное число):

$$f(a,n) \equiv \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } f(a,n-1)* a\ , \qquad (1.1)$$

или другой вариант

$$f(a,n) \equiv \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } (f^2(a, n \textbf{ div } 2))*f(a, n \textbf{ mod } 2) \qquad (1.2)$$

Учитывая, что n **mod** 2 может принимать только значения 0 или 1, а $f(a,0) = 1$ и $f(a,1) = a$, получим еще один вариант.

$f(a,n) \equiv$ **if** $n = 0$ **then** 1 **else if** Even(n) **then** ($f^2(a, n$ **div** 2)) **else** ($f^2(a, n$ **div** 2)) a, *(1.3)*

где Even(n) — функция, возвращающая значение True, если n — четное.

Последнему определению (1.3) соответствует приведенная ниже программа, использующая рекурсивную функцию power.

```
//    Пример: рекурсивное вычисление степени a^n
//    вариант 1
#include <iostream>
#include <windows.h>
using namespace std ;
int main ( )
{     //для правильной кодировки русских букв:
      SetConsoleCP(1251);
      SetConsoleOutputCP(1251);

      float power (float, unsigned int);
      float a, y;
      unsigned int n;
      cout << "Введите вещественное a:" << endl ;
      cin >> a;
      cout << "Введено a:" << a << "\n" ;
      cout << "Введите показатель степени n >= 0 :" << endl ;
      cin >> n;
      cout << "Введено n >= 0 :" << n << "\n" ;
//{n >= 0}
      y = power(a, n);
//(y = a^n)
      cout << "Конец рекурсивного вычисления -\n" ;
      cout << "Вычислено " << a << "^" << n << " = " << y << " \n" ;
      return 0 ;
}
float power ( float a, unsigned int n)
{     float p;
      if (n == 0) return 1;
      else {
                  p = power (a, n/2);
                  p = p*p;
                  if ( n % 2) p = p*a;
                  return p;
      };
}
```

Возможны и другие варианты рекурсивной реализации функции power.

Вариант 2.

```
float power2 ( float a, unsigned int n)
{     float p;
      if (n == 0) return 1;
      else{
            p = power2 (a*a, n/2);
            if ( n % 2) p = p*a;
            return p;
      };
}
```

Вариант 3. С накопителем.

```
float power3 ( float a, unsigned int n, float b)
// accumulator b
{     float p;
      if (n == 0) return b;
      else{
            if ( n % 2) p = power3 (a, n-1, b*a);
            else p = power3 (a*a, n/2, b);
            return p;
      };
}
```

Параметр b хранит значение степени на предыдущем шаге.

Вариант 4. С использованием «хвостовой» рекурсии

```
float power4 ( float a, unsigned int n, float b)
// accumulator + tail recursion !
{
      if (n == 0) return b;
      else{
            if ( n % 2) {b = b*a; n = n-1;}
            else {a = a*a; n = n/2;}
            return power4 (a, n, b);
      };
}
```

Вызов power4 осуществляется на последнем шаге функции, при возвращении ее значения. Если в версии 3 использовались обращения к функции с разными значениями параметров в зависимости от четного или нечетного значения n на очередном шаге, то в версии 4 параметры вычисляются по-разному, но обращение к функции одинаковое. От этой версии уже 1 шаг до итеративной функции, приведенной ниже.

```
float power5 ( float a, unsigned int n) // not recursion
{     float b = 1;
      while(n!=0) {
            if (n % 2) {b = b*a; n = n-1;}
            else {a = a*a; n = n/2;}
      }
      return b;
```

}

Взаимно-рекурсивные функции и процедуры. Синтаксический анализатор понятия скобок

Рассмотрим пример взаимной рекурсии, когда P обращается к Q, а Q к P. Пусть требуется построить синтаксический анализатор понятия скобки:

скобки::=квадратные | *круглые*
квадратные::=[*круглые круглые*] | +
круглые::=(квадратные квадратные) | –

В этом рекурсивном определении последовательности символов, называемой скобки, присутствуют две взаимно-рекурсивные части: квадратные определяются через круглые, и наоборот, круглые – через квадратные. В простейшем случае квадратные есть символ «+», а круглые есть символ «–». Другие примеры последовательностей, порождаемых этим рекурсивным определением:

‘[– –]’, ‘(++)’, ‘[(++)([–(++)][– –])]’, ‘(+[(++)([–(++)][(+[– –])–])])’.

Синтаксическим анализатором будем называть программу, которая определяет, является ли заданная (входная) последовательность символов скобками или нет. В случае ответа «нет» сообщается место и причина ошибки.

Реализуем основную часть этой программы как булевскую функцию Bracket, которая вызывает две другие (парные) булевские функции Round и Square, определяющие, является ли текущая подпоследовательность частью круглые или квадратные соответственно. Каждая из функций Round и Square в свою очередь вызывает парную к себе (Square и Round соответственно). Входная последовательность читается из файла. Вспомогательные сообщения квалифицируют ошибки в записи последовательности скобки в том случае, когда результат функции Bracket есть False. Для формирования этих сообщений будет использована процедура Error.

Функции Round и Square реализованы так, что они читают очередной символ входной последовательности и далее действуют в прямом соответствии с рекурсивными определениями частей круглые и квадратные соответственно. При этом в функции Bracket приходится читать первый символ входной последовательности дважды. Можно было бы избежать этого, используя «заглядывание вперед», однако такая реализация менее прозрачна.

```
// Program SyntaxAnalysisOfBracket;
// вариант с синхронным выводом входной строки (до места ошибки включительно)
/*    Определения (синтаксис)
      Bracket = скобки, Round = кругл, Square = квадр
      скобки ::= квадр | кругл
      квадр ::= + | [кругл кругл]
      кругл ::= – | (квадр квадр)
*/
#include <iostream>
#include <fstream>
#include <iomanip>
#include <windows.h>
using namespace std ;

bool Bracket(ifstream &infile);
bool Round (ifstream &infile, char s);
bool Square (ifstream &infile, char s );
void Error (short k);

int main ( )
{
      setlocale (0,"Rus");                  // для MVC++ 2010

      ifstream infile ("in_seq5.txt");
      if (!infile) cout << "Входной файл не открыт!" << endl;

      cout << "Анализатор скобок:" << endl;
      bool b = Bracket (infile);

      cout << endl;
      if (b) cout << "ЭТО СКОБКИ!" << endl;
      else    cout <<"НЕТ, ЭТО НЕ СКОБКИ!" << endl;

      system("PAUSE");
      return 0;
}

bool Round (ifstream &infile, char s)
// кругл ::= – | (квадр квадр)
// s      —  текущий символ входной строки
{     bool k;
      if (s == '-') { return true;}
      else if ( s == '(' )
            {     //кругл ::= (квадр квадр)
                  if (infile >> s)
                  {     cout << s;
                        k = Square (infile,s);
                        if (k)
                        {     if (infile >> s)
```

```
                    {     cout << s;
                          k = Square (infile,s);}
                    else {Error (5); return false;} // квадр - пуст!
               }
               else return false; //первый квадр ошибочен

               if (k) // оба квадр правильны
                    if (infile >> s)
                    {     cout << s;
                          return (s == ')');
                    }
                    else {Error (6); return false;}
               else return false;
          }
          else { Error (5); return false;} // квадр      —  пуст!
     }
     else    { Error(7); return false;} // не – и не ( }
}
// end of Round

bool Square (ifstream &infile, char s)
// квадр ::= + | [кругл кругл]
// s - текущий символ входной строки
{    bool k;
     if (s == '+') return true;
     else if ( s == '[' )
     {    //квадр ::= [кругл кругл]
          if (infile >> s)
          {    cout << s;
               k = Round (infile,s);
               if (k)
               {    if (infile >> s)
                    {    cout << s;
                         k = Round (infile,s);
                    }
                    else {Error (8); return false;} // кругл      —  пуст!
               }
               else return false; //первый кругл ошибочен

               if (k) // оба кругл правильны
                    if (infile >> s)
                    {    cout << s;
                         return (s == ']');
                    }
                    else {Error (3); return false;}
               else return false;
          }
          else { Error (8); return false;} // кругл    —  пуст!
     }
     else    { Error(4); return false;} // не + и не [ }
```

```
}
// end of Square


bool Bracket(ifstream &infile)
{    char s;
     bool b;
     b = false;
     if (infile >> s)
     {    cout << s;
          if ((s == '+') || (s == '[')) b = Square (infile, s);
          else       if ((s == '-') || (s == '(')) b = Round (infile, s);
          else Error(2);      //недопустимый начальный символ
          infile >> s;
          if (b && !infile.eof())    Error(1);   // лишние символы
          b = (b && infile.eof());
     }
     else Error (0);           // пустая входная строка
return b;
}

void Error (short k)
{
     cout << endl << "err#" << k << endl;
  switch (k) {
     case 0: cout << "! - Пустая входная строка" << endl; break;                  //{Bracket}
     case 1: cout << "! - Лишние символы во входной строке" << endl; break;     //{Bracket}
     case 2: cout << "! - Недопустимый начальный символ" << endl; break;        //{Bracket}
     case 3: cout << "! - Отсутствует ']'." << endl;   break;                    //{Square}
     case 4: cout << "! - Отсутствует '+' или '['." << endl;     break;          //{Square}
     case 5: cout << "! - Очередной квадр  —  пуст." << endl; break;            //{Round}
     case 6: cout << "! - Отсутствует ')'." << endl;   break;                    //{Round}
     case 7: cout << "! - Отсутствует – или (." << endl; break;                  //{Round}
     case 8: cout << "! - Очередной кругл  —  пуст." << endl; break;            //{Square}
     default : cout << "! - ...";break;                                         // ?
  };
}
// end of Error
```

1.2. Требования и рекомендации к выполнению задания:

Прежде чем приступить к выполнению задания этого раздела, рекомендуется ознакомиться с разделом 2 учебного пособия [1].

Задания к разделу Рекурсия представлены в **Приложении 1**. При выполнении задания необходимо:

1. проанализировать полученное задание, выделив рекурсивно определяемые информационные объекты и (или) действия;

2. разработать программу, использующую рекурсию;

3. сопоставить рекурсивное решение с итеративным решением задачи;

4. сделать вывод о целесообразности и эффективности рекурсивного решения данной задачи.

РАЗДЕЛ 2. РЕКУРСИВНАЯ ОБРАБОТКА ИЕРАРХИЧЕСКИХ СПИСКОВ

2.1. Цель и задачи

Познакомиться с одной из часто используемых на практике нелинейных конструкций, способами её организации и рекурсивной обработки. получить навыки решения задач обработки иерархических списков, как с использованием базовых функций их рекурсивной обработки, так и без использования рекурсии.

Рекурсивное определение иерархического списка

В практических приложениях возникает необходимость работы с более сложными, чем линейные списки, нелинейными конструкциями. Рассмотрим одну из них, называемую иерархическим списком элементов базового типа El или S-выражением.

Определим соответствующий тип данных S_expr (El) рекурсивно, используя определение линейного списка (типа L_list):

< S_expr (El) > ::= < Atomic (El) > | < L_list (S_expr (El)) >,

< Atomic (E) > ::= < El >.

< *L_list*(*El*) > ::= < *Null_list* > | < *Non_null_list*(*El*) >

< *Null_lis t*> ::= *Nil*

< *Non_null_list*(*El*) > ::= < *Pair*(*El*) >

< *Pair*(*El*) > ::= (< *Head_l*(*El*) > . < *Tail_l*(*El*) >)

< *Head_l*(*El*) > ::= < *El* >

< *Tail_l*(*El*) > ::= < *L_list*(*El*) >

Представление иерархического списка

Традиционно иерархические списки представляют или графически или в виде скобочной записи. На рисунке 2.1 приведен пример графического

изображения иерархического списка. Соответствующая этому изображению сокращенная скобочная запись – это (a (b c) d e.

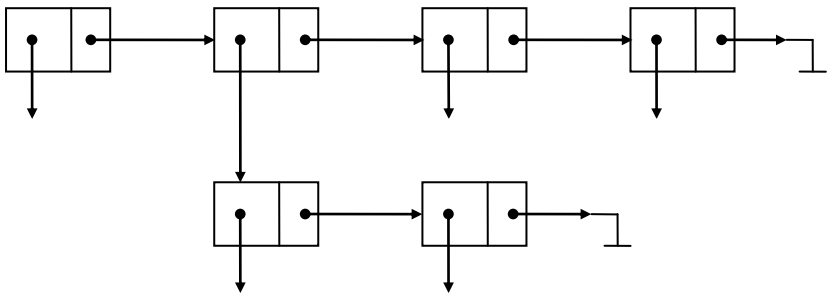


Рис. 2.1. Пример представления иерархического списка в виде двумерного рисунка

Переход от полной скобочной записи, соответствующей определению иерархического списка, к сокращенной производится путем отбрасывания конструкции . Nill и удаления необходимое число раз пары скобок вместе с предшествующей открывающей скобке точкой.

Полная запись	Сокращенная запись
a	***a***
Nil	( )
(***a*** **.** (***b*** **.** (***c*** **.** *Nil*)))	(***a b c***)
(***a*** **.** ((***b*** **.** (***c*** **.** *Nil*)) **.** (***d*** **.** (***e*** **.** *Nil*))))	(***a*** (***b c***) ***d e***)

Рис. 2.2. Примеры перехода от полной к сокращенной скобочной записи иерархических списков

Согласно приведенному определению иерархического списка, структура непустого иерархического списка – это элемент размеченного объединения множества атомов и множества пар «голова-хвост».

Рекурсивная структура иерархического списка на языке C++

```
typedef char base;      // базовый тип элементов (атомов)
    struct   s_expr;
    struct   two_ptr   {
        s_expr   *hd;
        s_expr   *tl;
    } ;   //end two_ptr;
    struct   s_expr {
```

```
        bool tag; // true: atom, false: pair
        union    {
              base    atom;
              two_ptr pair;
        } node;        //end union node
};              //end s_expr
typedef    s_expr    *lisp;
```

Поясняющие эту структуру иллюстрации представлены на рисунке 2.3.

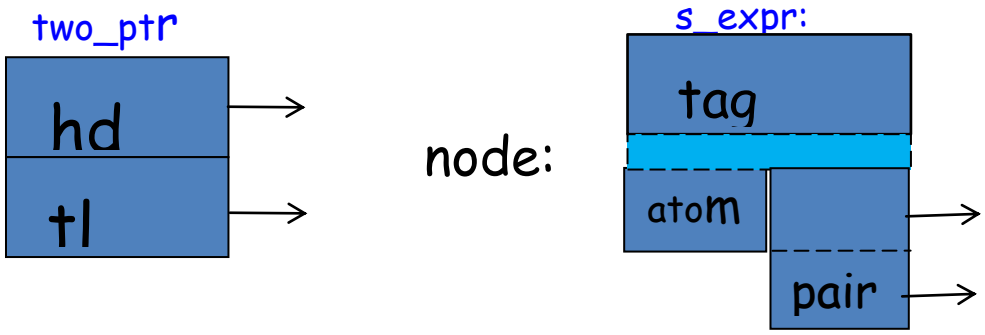


Рис. 2.3. Представление рекурсивной структуры списка

Функциональная спецификация и реализация иерархического списка

Функциональная спецификация иерархического списка включает:

- функции — селекторы Head и Tail , выделяющие соответственно «голову» и «хвост» списка
- функции — конструкторы: Cons, создающая точечную пару (новый список из «головы» и «хвоста»), и Make_Atom, создающая атомарное S-выражение.
- предикаты Is Null, проверяющий список на отсутствие в нем элементов, и Atom, проверяющий, является ли список атомом.

Необходимо включить в список базовых функций и функцию Destroy, позволяющую уничтожить созданный список, т.е. освободить псмять от ставших ненужными списочных структур.

При реализации иерархического списка, рассматриваемого нами, как абстрактный тип данных (АТД), на языке C++, поместим прототипы всех перечисленных выше базовых функций в отдельный файл с расширением .h. lisp head (const lisp s);

```
lisp tail (const lisp s);
    lisp cons (const lisp h, const lisp t);
    lisp make_atom (const base x);
    bool isAtom (const lisp s);
    bool isNull (const lisp s);
```

```
void destroy (lisp s);
```

Файл реализации этих функций имеет расширение .cpp.

```
lisp head (const lisp s)
    {// PreCondition: not null (s)
        if (s != NULL)
            if (!isAtom(s))    return s->node.pair.hd;
            else { cerr << "Error: Head(atom) \n";  exit(1); }
        else { cerr << "Error: Head(nil) \n";
            exit(1);
        }
    }
```

Если «голова» списка не атом, то функция **head** возвращает список, на который указывает голова пары, т.е. подсписок, находящийся на следующем уровне иерархии. Если же «голова» списка — атом, то выводится сообщение об ошибке и функция прекращает работу.

```
lisp tail (const lisp s)
{// PreCondition: not null (s)
        if (s != NULL)
            if (!isAtom(s))    return s->node.pair.tl;
            else { cerr << "Error: Tail(atom) \n"; exit(1); }
        else { cerr << "Error: Tail(nil) \n";
            exit(1);
        }
    }
bool isAtom (const lisp s)
    {    if(s == NULL) return false;
    else return (s -> tag);
    }
//.....................................
    bool isNull (const lisp s)
    { return s==NULL;
    }
```

Предикат **isAtom** возвращает значение tag, которое равно True, если элемент — атом, и значение False, если — «голова-хвост». В случае пустого списка значение предиката False.

```
lisp cons (const lisp h, const lisp t)
    // PreCondition: not isAtom (t)
    {lisp p;
```

```
if (isAtom(t)) { cerr << "Error: cons(*, atom) \n"; exit(1);}
else { p = new s_expr;
        if ( p == NULL)   {cerr << "Memory ...\n"; exit(1); }          else {
          p->tag = false;
          p->node.pair.hd = h;
          p->node.pair.tl = t;
          return p;
} } }
```

Функция **Cons** — конструктор. При создании нового S-выражения требуется выделение памяти. Если памяти нет, то p == NULL и это приводит к выводу соответствующего сообщения об ошибке. Если «хвост» — не атом, то для его присоединения к «голове» требуется создать новый узел (элемент), головная ссылка которого будет ссылкой на «голову» этого «хвоста», а хвостовая часть элемента (tag.hd.tl) — ссылкой на его «хвост»

```
lisp make_atom (const base x)
    {   lisp s;
        s = new s_expr;
        s -> tag = true;
        s->node.atom = x;
        return s;
    }
```

Создается узел типа (tag, x).

```
void destroy (lisp s)
    {
        if ( s != NULL) {
            if (!isAtom(s)) {
                destroy ( head (s));
                destroy ( tail(s));
            }
        delete s;
        };
    }
```

Функция **delete** удаляет текущий элемент списка.

Вся работа с иерархическими списками осуществляется только с помощью описанных выше базовых функций.

Над иерархическими списками могут выполняться, например, следующие операции:

- добавление нового элемента (списка или атома),
- удаление всех вхождений заданного элемента (атома),
- проверка иерархического списка на наличие в нем заданного элемента (атома),
- замена всех вхождений одного элемента на другой,
- подсчет числа атомов в списке или числа одинаковых атомов,
- проверка идентичности 2-х иерархических списков,
- обращение списка
- вычисление глубины (числа уровней вложения)

Следует помнить, что функция Cons не формирует копий исходных S-выражений, т.о. программист должен сам заботиться о копировании списочных структур, избегая побочного эффекта от включения одних списков в другие в качестве фрагментов. Т.о. в дополнение к базовым функциям в состав модуля реализации иерархического списка необходимо включить функцию копирования списка Copy_Lisp.

```
lisp copy_lisp (const lisp x)
{   if   (isNull(x)) return NULL;
    else if (isAtom(x))
                    return make_atom (x->node.atom);
    else return    cons (copy_lisp (head (x)),
                         copy_lisp (tail(x)));
} //end copy-lisp
```

Эта функция, однако, использует уже описанные выше базовые функции **Cons, Head, Tail, Make_Atom**, а также предикаты **IsNull** и **IsAtom**.

Кроме того, какую бы задачу мы ни решали, нам потребуются процедуры ввода/вывода иерархического списка. Эти процедуры должны быть написаны применительно к конкретной форме представления списков. Поскольку иерархические списки чаще всего представляются сокращенной скобочной записью, то можно предложить следующую процедуру вывода иерархического списка:

```
// функции вывода:
    void write_lisp (const lisp x);  // основная
    void write_seq (const lisp x);
void write_lisp (const lisp x)
    {//пустой список выводится как ()
```

```
    if (isNull(x)) cout << " ()";
    else if (isAtom(x)) cout << ' ' << x->node.atom;
         else   { //непустой список
              cout << " (" ;
              write_seq(x);
              cout << " )";
         }
    } // end write_lisp
void write_seq (const lisp x)
{/*выводит последовательность элементов списка без обрамляющих его скобок */
    if (!isNull(x)) {
         write_lisp(head (x));
         write_seq(tail (x));
    }
}
```

Здесь процедура вывода списка с обрамляющими его скобками — write_lisp, а без обрамляющих скобок — write_seq. Процедура write_lisp использует внутри себя вызов процедуры write_seq.

Для ввода иерархического списка, представленного сокращенной скобочной записью, при условии ввода этой записи с клавиатуры можно предложить процедуру read_lisp. Эта процедура использует внутри себя обращение к процедуре read_s_expr, а она, в свою очередь, обращение к read_seq.

```
// функции ввода:
    void read_lisp ( lisp& y);        // основная
    void read_s_expr (base prev, lisp& y);
    void read_seq ( lisp& y);

// ввод списка с консоли
    void read_lisp ( lisp& y)
    {     base x;
         do cin >> x; while (x==' ');
         read_s_expr ( x, y);
    } //end read_lisp

void read_s_expr (base prev, lisp& y)
{ //prev     — ранее прочитанный символ
    if ( prev == ')' ) {cerr << " ! List.Error 1 " << endl; exit(1); }
    else if ( prev != '(' ) y = make_atom (prev);
         else read_seq (y);
```

```
} //end read_s_expr

void read_seq ( lisp& y)
    {    base x;
         lisp p1, p2;
         if (!(cin >> x)) {cerr << " ! List.Error 2 " << endl; exit(1);}
         else {
              while   ( x==' ' ) cin >> x;
              if ( x == ')' ) y = NULL;
              else {     read_s_expr ( x, p1);
                   read_seq ( p2);
                   y = cons (p1, p2);
              }
         }
    } //end read_seq
```

Язык C++ поддерживает парадигмы процедурного, модульного и объектно-ориентированного программирования. Используя модульное программирование, получим проект, содержащий 3 файла:

- Заголовочный файл “l_intrfc.h”
- Файл реализации “l_impl.cpp”
- Файл с клиентской программой “l_mod1.cpp”, которая использует структуры данных и функции, объявленные в интерфейсе (“l_intrfc.h”)

2.2. Рекомендации по изучению раздела и выполнению индивидуального задания

При изучении данного раздела студент должен ознакомиться с разделом 1.7. учебного пособия [2}, выбрать индивидуальное задание из списка заданий, указанных в **Приложении 2**. Каждое задание предполагает самостоятельную разработку студентом одного или нескольких модулей на языке C++, реализующих согласованный с преподавателем набор операций над иерархическими списками, а также главной программы, непосредственно решающей поставленную задачу.

Предполагается выполнение задания в двух вариантах: с использованием базовых функций рекурсивной обработки иерархических списков и без использования рекурсии.

Во всех случаях, когда это не оговорено особо, предполагается, что исходные и результирующие списки размещаются в файлах подходящего типа. Для представления иерархических списков рекомендуется использовать сокращенную скобочную запись.

Рассмотрим для примера программу, реализующую следующие функции обработки иерархических списков:

● Сцепление двух иерархических списков, соединение их в один список (функция **Concat**). Например, y = (a (b c) d), z = (e f (g) h), Concat (y,z) = (a (b c) d e f (g) h)

● Обращение иерархического списка на всех уровнях вложения (функция **Reverse**). Например, (a (b c) d) → (d (c b) a)..

● Выравнивание иерархического списка, т.е. формирование из него линейного списка путем удаления из сокращенной скобочной записи иерархического списка всех внутренних скобок (функция **Flatten**) . Например, ((a b) c (d (e f (g)) h)) → (a b c d e f g h).

Функция **Concat** создает новый иерархический список из копий атомов, входящих в соединяемые списки.

Обращающая список функция **Reverse** в процессе работы создает рабочие копии атомов, входящих во временно существующие списки, которые уничтожаются в процессе работы. В приведенной ниже программ используется функция **Rev**, использующая накапливающие параметры. Функция Reverse запускает ее.

Для выравнивания иерархического списка в программе используется простой для понимания, но не слишком эффективный вариант функции. Более эффективный, но и более сложный для понимания вариант с накопителем t.

```
Выровнять (s, t) □
if Null (s) then t
else    {s    — непустой список}
   if Atom (s) then Cons (s, t )
        else
   Выровнять2 (Head (s), Выровнять (Tail (s), t))
```

```
// интерфейс АТД "Иерархический Список"
namespace h_list
{
```

```
    typedef char base;      // базовый тип элементов (атомов)

    struct s_expr;
    struct   two_ptr
        {
        s_expr *hd;
        s_expr *tl;
    } ;  //end two_ptr;

    struct s_expr {
        bool tag; // true: atom, false: pair
        union
            {
            base atom;
            two_ptr pair;
        } node;        //end union node
    };              //end s_expr

    typedef s_expr *lisp;

// функции
    void print_s_expr( lisp s );
    // базовые функции:
    lisp head (const lisp s);
    lisp tail (const lisp s);
    lisp cons (const lisp h, const lisp t);
    lisp make_atom (const base x);
    bool isAtom (const lisp s);
    bool isNull (const lisp s);
    void destroy (lisp s);

    base getAtom (const lisp s);

    // функции ввода:
    void read_lisp ( lisp& y);              // основная
    void read_s_expr (base prev, lisp& y);
    void read_seq ( lisp& y);

    // функции вывода:
    void write_lisp (const lisp x);        // основная
    void write_seq (const lisp x);

    lisp copy_lisp (const lisp x);

} // end of namespace h_list


// continue of namespace h_list
#include "l_intrfc.h"
#include <iostream>
```

```
#include <cstdlib>

using namespace std;
namespace h_list
{

//.....................................
	lisp head (const lisp s)
	{// PreCondition: not null (s)
		if (s != NULL) if (!isAtom(s)) return s->node.pair.hd;
			else { cerr << "Error: Head(atom) \n"; exit(1); }
		else { cerr << "Error: Head(nil) \n";
			exit(1);
		}
	}
//.....................................
	bool isAtom (const lisp s)
	{	if(s == NULL) return false;
	else return (s -> tag);
	}
//.....................................
	bool isNull (const lisp s)
	{ return s==NULL;
	}
//.....................................
	lisp tail (const lisp s)
{// PreCondition: not null (s)
		if (s != NULL) if (!isAtom(s)) return s->node.pair.tl;
			else { cerr << "Error: Tail(atom) \n"; exit(1); }
		else { cerr << "Error: Tail(nil) \n";
			exit(1);
		}
	}
//.....................................
	lisp cons (const lisp h, const lisp t)
	// PreCondition: not isAtom (t)
	{lisp p;
	if (isAtom(t)) { cerr << "Error: Cons(*, atom)\n"; exit(1);}
	else {
		p = new s_expr;
		if ( p == NULL)  {cerr << "Memory not enough\n"; exit(1); }
		else {
			p->tag = false;
			p->node.pair.hd = h;
			p->node.pair.tl = t;
			return p;
		}
	}
	}
//.....................
```

```
lisp make_atom (const base x)
{    lisp s;
     s = new s_expr;
     s -> tag = true;
     s->node.atom = x;
     return s;
}

//.....................
void destroy (lisp s)
{
if ( s != NULL) {
     if (!isAtom(s)) {
          destroy ( head (s));
          destroy ( tail(s));
     }
     delete s;
     // s = NULL;
};
}
//.....................
base getAtom (const lisp s)
{
     if (!isAtom(s)) { cerr << "Error: getAtom(s) for !isAtom(s) \n"; exit(1);}
     else return (s->node.atom);
}


//.....................
// ввод списка с консоли
void read_lisp ( lisp& y)
{    base x;
     do cin >> x; while (x==' ');
     read_s_expr ( x, y);
} //end read_lisp
//.....................
void read_s_expr (base prev, lisp& y)
{ //prev      —  ранее прочитанный символ}
     if ( prev == ')' ) {cerr << " ! List.Error 1 " << endl; exit(1); }
     else if ( prev != '(' ) y = make_atom (prev);
          else read_seq (y);
} //end read_s_expr
//.....................
void read_seq ( lisp& y)
{    base x;
     lisp p1, p2;

     if (!(cin >> x)) {cerr << " ! List.Error 2 " << endl; exit(1);}
     else {
          while    ( x==' ' ) cin >> x;
```

```
                if ( x == ')' ) y = NULL;
                else {
                      read_s_expr ( x, p1);
                      read_seq ( p2);
                      y = cons (p1, p2);
                }
          }
     } //end read_seq
//.........................
// Процедура вывода списка с обрамляющими его скобками   –  write_lisp,
//   а без обрамляющих скобок     –  write_seq
     void write_lisp (const lisp x)
     {//пустой список выводится как ()
     if (isNull(x)) cout << " ()";
     else if (isAtom(x)) cout << ' ' << x->node.atom;
          else   { //непустой список}
                cout << " (" ;
                write_seq(x);
                cout << " )";
          }
     } // end write_lisp
//.........................
void write_seq (const lisp x)
{//выводит последовательность элементов списка без обрамляющих его скобок
     if (!isNull(x)) {
          write_lisp(head (x));
          write_seq(tail (x));
     }
}
//.........................
lisp copy_lisp (const lisp x)
{    if   (isNull(x)) return NULL;
     else if (isAtom(x)) return make_atom (x->node.atom);
     else return cons (copy_lisp (head (x)), copy_lisp (tail(x)));
} //end copy-lisp

} // end of namespace h_list

/*   использование модуля с АТД "Иерархический Список" .
Интерфейс модуля в заголовочном файле "l_intrfc.h"
и его реализация (в отдельном файле l_impl.cpp) образуют
пространство имен namespace h_list
*/
#include <iostream>
#include <cstdlib>
#include "l_intrfc.h"
#include <windows.h>

using namespace std;
using namespace h_list;
```

```
lisp concat (const lisp y, const lisp z);

lisp reverse(const lisp s);
lisp rev(const lisp s,const lisp z);

lisp flatten(const lisp s);

      int main ( )
{     SetConsoleCP(1251);                  // для вывода кирилицы
      SetConsoleOutputCP(1251);   // для вывода кирилицы


      lisp s1, s2, s3;
      cout << boolalpha;
      cout << "введите list1:" << endl;
      read_lisp (s1);
      cout << "введен list1: " << endl;
      write_lisp (s1);
      cout << endl;

      cout << "flatten списка = " << endl;
      s3 = flatten (s1);
      write_lisp (s3);
      cout << endl;

      s2 = reverse(s1);
      cout << "для списка:" << endl;
      write_lisp (s1);    cout << endl;
      cout << "обращенный список есть:" << endl;
      write_lisp (s2);    cout << endl;

      cout << "destroy list3: " << endl;
      destroy ( s3);

      cout << "end! " << endl;
      return 0;
}
//.........................
      lisp concat (const lisp y, const lisp z)
      {
      if (isNull(y)) return copy_lisp(z);
      else return cons (copy_lisp(head (y)), concat (tail (y), z));
      } // end concat
// ----------------------
      lisp reverse(const lisp s)
      {     return(rev(s,NULL));
      }
//.....................
      lisp rev(const lisp s,const lisp z)
```

```
    {
        if(isNull(s)) return(z);
        else if(isAtom(head(s))) return(rev(tail(s), cons(head(s),z)));
        else return(rev(tail(s), cons(rev(head(s), NULL),z)));
    }
//.....................
    lisp flatten(const lisp s)
    {
        if (isNull(s)) return NULL;
        else if(isAtom(s)) return cons(make_atom(getAtom(s)),NULL);
        else //s   –  непустой список
        if (isAtom(head(s))) return cons( make_atom(getAtom(head(s))),flatten(tail(s)));
        else //Not Atom(Head(s))
            return concat(flatten(head(s)),flatten(tail(s)));
    } // end flatten
```

РАЗДЕЛ 3. ЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ: СТЕК, ОЧЕРЕДЬ, ДЕК.

3.1. Цель и задачи

Познакомиться с часто используемыми на практике линейными структурами данных, обеспечивающими доступ к элементам последовательности только через её начало и конец, и способами реализации этих структур, освоить на практике использование стека, очереди и дека для решения задач .

Спецификация стека и очереди

Стек, очередь и дек представляют собой разновидности линейного списка.

При задании спецификации линейных списков использовалась модель последовательности [2]. Доступ к каждому элементу последовательности можно получить, продвигаясь по списку от одного элемента к другому. В модели выделяется пройденная часть и текущий элемент, с которого начинается еще не пройденная часть. При этом ранее в курсе лекций и в учебном пособии [2] были определены функции над последовательностями:

- **First** возвращает первый элемент последовательности
- **Last** возвращает последний элемент последовательности
- **Rest** возвращает исходную последовательность, в которой присутствуют все элементы за исключением первого,

- **Lead** возвращает исходную последовательность, в которой присутствуют все элементы за исключением последнего,
- **Prefix** добавляет элемент в начало последовательности
- **Postfix** добавляет элемент в конец последовательности.

Доступ к элементам последовательности (чтение и запись) , осуществляется только через ее начало и конец. Функции First, Rest, Last, Lead – селекторы, а функции Prefix и Postfix – конструкторы.

Если ограничиться только функциями First, Rest, Prefix (или только функциями Last, Lead, Postfix), то получается структура данных, известная как стек (или магазин). Если использовать только набор функций First, Rest, Postfix (или только Last, Lead, Prefix) получим структуру данных, которая называется очередь (англ. queue). Если же используют весь набор функций, то соответствующую структуру данных обычно называют дек (от англ. deq – аббревиатуры сочетания double-ended-queue, т. е. «очередь с двумя концами»). Во все эти структуры данных необходимо добавить еще предикат-индикатор Null, идентифицирующий пустую последовательность, и либо константу, обозначающую пустую последовательность Δ, либо функцию Crtate, порождающую пустую последовательность.

Стек Stack	*First, Rest, Prefix*
	Last, Lead, Postfix
Очередь Queue	*First, Rest, Postfix*
	Last, Lead, Prefix
Дек Deq	*First, Rest, Last, Lead, Postfix, Prefix*

Рис. 3.1. Подструктуры последовательности

Исторически сложилось, что при работе со стеком для обозначения функций **First, Rest, Prefix** используют их синонимы **Top, Pop** и **Push** соответственно (Top – верхушка стека, Pop {up} – вытолкнуть (вверх), Push {down} – втолкнуть, вжать). Полезно определить процедуру **Pop2,** совмещающую результат действия функций Top и Pop.

```
procedure Pop2 (out p: α;  in-out s:  Stack (α));
begin
  p := Top (s); s := Pop (s)
end
```

Реализация стека и очереди

Ссылочная реализация стека и очереди в динамической памяти в основном аналогична ссылочной реализации линейных списков. Упрощение связано с отсутствием необходимости работать с текущим элементом списка. Идеи такой реализации ясны из рисунка 3.2.

Для ссылочной реализации дека естественно использовать двунаправленный список.

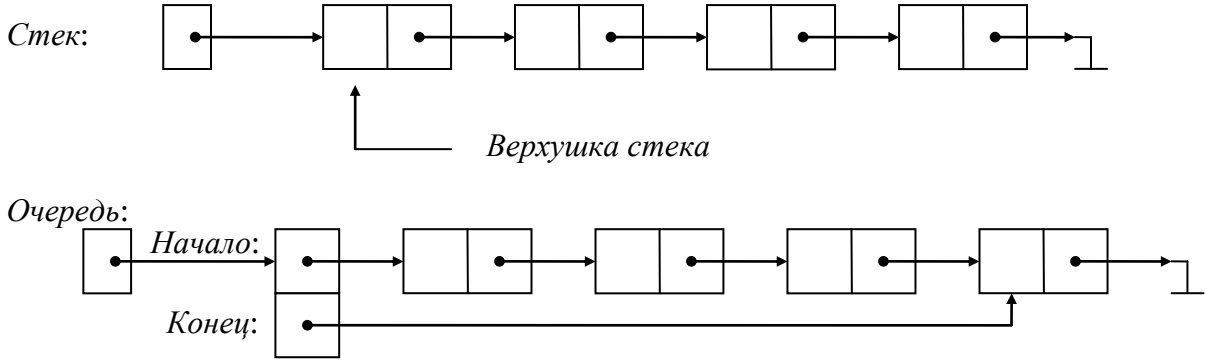


Рис. 3.2. Ссылочное представление стека и очереди

Поскольку для стека, очереди и дека доступ к элементам осуществляется только через начало и конец последовательности, то эти структуры данных допускают и эффективную непрерывную реализацию на базе вектора. При этом используется одномерный массив Mem: array [0..n] of α и переменная Верх : -1..n.

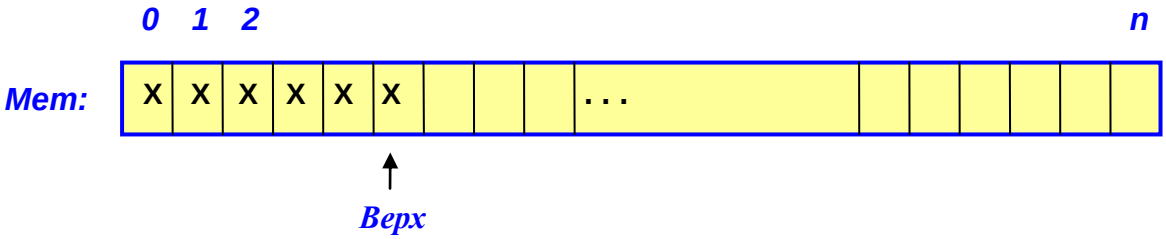


Рис. 3.3. Непрерывная реализация ограниченного стека на базе вектора

Для пустого стека Верх = = -1, для целиком заполненного стека Верх = = n. Вершина стека доступна как Mem [Верх], операция Pop реализуется как Верх:= Верх –1, а операция Push (p, s) как {Верх = Верх + 1; Mem [Верх] = p } при -1 ≤ Верх < n.

Непрерывная реализация ограниченной очереди на базе вектора требует, в отличие от стека, двух переменных Начало и Конец.

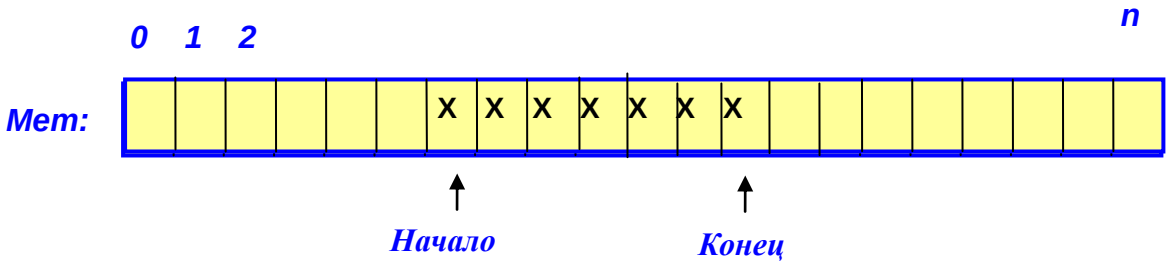


Рис. 34. Непрерывная реализация ограниченной очереди на базе вектора

Особенностью такого представления является наличие ситуации, когда последовательность элементов очереди по мере их добавления может выходить за границу вектора, продолжаясь с его начала (вектор имитирует здесь так называемый кольцевой буфер). Эта ситуация изображена на рис. 3.5.

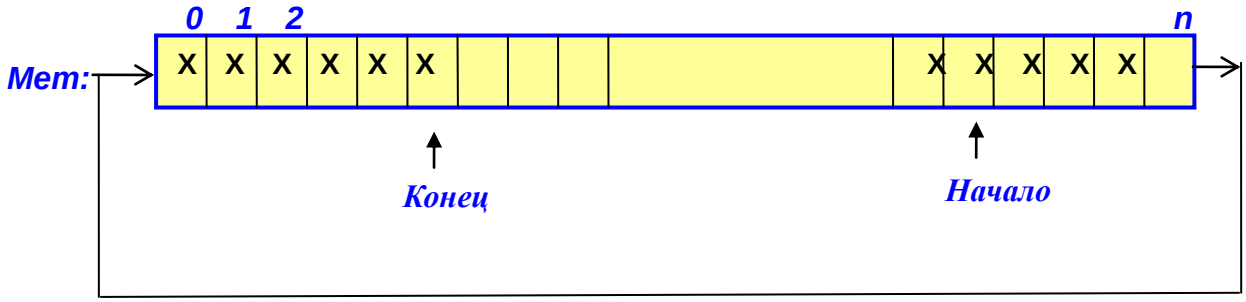


Рис. 3.5. Непрерывное представление очереди в кольцевом буфере

Переменная **Начало** может принимать значение **Конец+1** в случаях как пустой, так и полной очереди. Чтобы различать эти ситуации, надо ввести еще одну переменную **Длина**. Для пустой очереди Длина = 0, а для полной очереди Длина = n+1.

Так же может быть реализован и дек.

3.2. Рекомендации по изучению раздела и выполнению индивидуального задания

При изучении данного раздела студент должен ознакомиться с материалом раздела 2 пособия [2], выбрать индивидуальное задание из списка заданий, указанных в **Приложении 3**. Каждое задание предполагает самостоятельную разработку студентом одного или нескольких модулей на языке C++, реализующих тот или иной вариант стека, очереди или дека, а также главной программы, непосредственно решающей поставленную задачу.

В заданиях 1, 2, 3 следует использовать очередь и операции над ней. В заданиях 4 – 8 следует использовать стек и операции над ним. В заданиях с 9 по 11 следует использовать очередь и/или стек и операции над ними. При этом и стек и очередь могут быть реализован как на базе вектора, так и в связанной памяти (ссылочная реализация).

3.3. Пример реализации и использования стека для вычисления арифметического выражения в постфиксной форме

Напишем программу вычисления арифметического выражения в постфиксной форме. Суть работы программы состоит в следующем. В стек последовательно записываются символы входной строки, представляющей собой арифметическое выражение в постфиксной форме. Последовательно идущие друг за другом цифры преобразуются в числа. При появлении знака операции из стека последовательно выталкиваются 2 операнда, над ними выполняется операция, соответствующая этому знаку, результат помещается в стек и чтение входной строки продолжается.

Ссылочная реализация стека в динамической (связанной) памяти допускает несколько вариантов.

Вариант 1: Используется не шаблонный класс. Интерфейс и реализация стека разделены: st_interface.h — файл интерфейса стека, st_implementation.cpp — файл реализации стека. Файл интерфейса подключается к главной программе main1.cpp директивой #include “st_interf1.h”, а также к модулю реализации.

```
// Программа клиент вычисляет арифметическое выражение, заданное в постфиксной форме
// Ссылочная реализация в динамической (связанной) памяти
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <windows.h>
#include "st_interf1.h"
using namespace std;
using namespace st_modul1;

int main () {
	char a[100];
// это вставка для правильной кодировки русских букв
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
//
```

```
cout << "ввод строки с постфиксной записью выражения" << endl;
ifstream fin("postfix.txt");
fin >> noskipws; // включить манипулятор!

if (!fin ) {cout << "File not open for reading!\n"; return 1;}
int n1 = 100;
int n = 0;
while (n<n1 && fin >> a[n]) n++;
//вывод строки
cout << "длина строки = " << n << endl;
for (int i=0; i<n; i++) cout << a[i];
cout << endl;
cout << "вычислить!" << endl;

Stack s;
     for (int i = 0; i < n; i++)
     {     cout << "шаг: " << i+1 << " символ = " << a[i] << endl; // dem
          if (a[i] == '+')
            s.push(s.pop2() + s.pop2());
          if (a[i] == '*')
            s.push(s.pop2() * s.pop2());
          if ((a[i] >= '0') && (a[i] <= '9'))
            s.push(0);
          while ((a[i] >= '0') && (a[i] <= '9'))
                {s.push(10*s.pop2() + (a[i++]-'0'));
                cout << "шаг_: " << i+1 << " символ = " << a[i] << endl;// dem
                }
        }
     cout << "Результат =" << s.pop2() << endl;

     s.destroy();

return (0);
}

// интерфейс АТД "Стек" (ссылочная реализация в динамической памяти)
namespace st_modul1
{

//-----------------------------------
     typedef int base;

     class Stack {
     private:
          struct node;
/*   определение структуры будет дано в другом файле (продолжении namespace st_modul)
—в файле Implementation,
     а здесь достаточно объявления "struct node;"
*/
     node *topOfStack;
```

```
    public:
        Stack ()
            { topOfStack = NULL;
            };
        base top (void);
        void pop (void);
        base pop2(void);
        void push (const base &x);
        bool isNull(void);
        void destroy (void);
    };
}

// Implementation   —Реализация АТД "Стек"(ссылочная реализация в динамической памяти)
#include <iostream>
#include <cstdlib>
#include "st_interf1.h"
using namespace std ;

namespace st_modul1
{
    struct Stack::node {//
    base *hd;
    node *tl;
        // constructor
        node ()
            {hd = NULL; tl = NULL;
        }
    };// end node

//-------------------------------------
    base Stack::top (void)
    {// PreCondition: not null
        if (topOfStack == NULL) { cerr << "Error: top(null) \n"; exit(1); }
        else return *topOfStack->hd;
    }
//-------------------------------------
    void Stack::pop (void)
    {// PreCondition: not null
        if (topOfStack == NULL) { cerr << "Error: pop(null) \n"; exit(1); }
        else
        {   node *oldTop = topOfStack;
            topOfStack = topOfStack->tl;
            delete oldTop->hd;
            delete oldTop;
        }
    }
//-------------------------------------
```

```
    base Stack::pop2(void)
    {// PreCondition: not null
        if (topOfStack == NULL) { cerr << "Error: pop(null) \n"; exit(1); }
        else
        {   node *oldTop = topOfStack;
            base r = *topOfStack->hd;
            topOfStack = topOfStack->tl;
            delete oldTop->hd;
            delete oldTop;
            return r;
        }
    }
//-------------------------------------
    void Stack::push (const base &x)
    {   node *p;
        p = topOfStack;
        topOfStack = new node;
        if ( topOfStack != NULL){
            topOfStack->hd = new base;
            *topOfStack->hd = x;
            cout << "push -> " << x << endl;          // Demo
            topOfStack->tl = p;
        }
        else {cerr << "Memory not enough\n"; exit(1);}
    }
//-------------------------------------
    bool Stack::isNull(void)
    {   return (topOfStack == NULL) ;
    }
//-------------------------------------
    void Stack::destroy (void)
    {   while ( topOfStack != NULL) {
            pop();
        }
    }
} // end of namespace st_modul1
```

Вариант 2. Используется шаблонный класс, а интерфейс, и реализация располагаются в одном заголовочном файле st_interf2.h.

```
// Интерфейс АТД "Стек" (ссылочная реализация в динамической памяти)с шаблоном класса
// И интерфейс, и реализация в одном заголовочном файле
namespace st_modul2
{

//-------------------------------------
    template <class Elem>
    class Stack
```

```
{
private:
    struct node
{ //
    Elem *hd;
    node *tl;
    // constructor
    node ()
        {hd = NULL; tl = NULL;
    };
};// end node

node *topOfStack;

public:
    Stack ()
        { topOfStack = NULL;
        }//;

//-------------------------------------
    Elem Stack::top (void) //
    {// PreCondition: not null
        if (topOfStack == NULL) { cerr << "Error: top(null) \n"; exit(1); }
        else return *topOfStack->hd;
    }
//-------------------------------------
    void Stack::pop (void)//
    {// PreCondition: not null
        if (topOfStack == NULL) { cerr << "Error: pop(null) \n"; exit(1); }
        else
        {   node *oldTop = topOfStack;
            topOfStack = topOfStack->tl;
            delete oldTop->hd;
            delete oldTop;
        }
    }
//-------------------------------------
    Elem Stack::pop2(void)//
    {// PreCondition: not null
        if (topOfStack == NULL) { cerr << "Error: pop(null) \n"; exit(1); }
        else
        {   node *oldTop = topOfStack;
            Elem r = *topOfStack->hd;
            topOfStack = topOfStack->tl;
            delete oldTop->hd;
            delete oldTop;
            return r;
        }
    }
//-------------------------------------
```

```
    void Stack::push (const Elem &x)//
    {   node *p;
        p = topOfStack;
        topOfStack = new node;
        if ( topOfStack != NULL) {
            topOfStack->hd = new Elem;
            *topOfStack->hd = x;
            cout << "push -> " << x << endl; // Demo
            topOfStack->tl = p;
        }
        else {cerr << "Memory not enough\n"; exit(1);}
    }
//-------------------------------------
    bool Stack::isNull(void)//
    {   return (topOfStack == NULL) ;
    }
//-------------------------------------
    void Stack::destroy (void)//
    {   while ( topOfStack != NULL) {
            pop();
        }
    }
    };
}
```

Главная программа при этом не меняется по сравнению с первым вариантом, если не считать, что директива #include “st_interf1.h” изменится на #include “st_interf2.h”, a Stack s на Stack<int> s.

Вариант 3. Используется шаблонный класс. Интерфейс и реализация располагаются в разных h- файлах. Файл интерфейса подключается к главной программе, а файл реализации к заголовочному файлу (файлу интерфейса). Студенты могут предложить такой вариант реалиации самостоятельно.

Рассмотрим теперь непрерывную реализацию ограниченного стека на базе вектора.

Интерфейс и реализация будут располагаться в одном заголовочном файле «st_templ.h», используем шаблонный класс. Главную программу вычисления выражения в постфиксной форме назовем «postfix_calc.cpp».

```
// Программа клиент вычисляет арифметическое выражение, заданное в постфиксной
форме
// Непрерывная реализация на базе вектора
#include <iostream>
#include <fstream>
#include <cstdlib>
```

```
#include <windows.h>
#include "st_templ.h"
using namespace std;
int main ( )
{     char a[100]; //
      int maxN;
// это вставка для правильной кодировки русских букв
      SetConsoleCP(1251);
      SetConsoleOutputCP(1251);
//
cout << "введите максимальный размер стека =" << endl;
cin >> maxN;
cout << "введен максимальный размер стека =" << maxN << endl;

cout << "ввод строки с постфиксной записью выражения" << endl;
ifstream fin("postfix.txt");
fin >> noskipws; // включить манипулятор!

if (!fin ) {cout << "File not open for reading!\n"; return 1;}
int n1 = 100;
int n = 0;
while (n<n1 && fin >> a[n]) n++;
//вывод строки
cout << "длина строки = " << n << endl;
for (int i=0; i<n; i++) cout << a[i];
cout << endl;
cout << "вычислить!" << endl;

STACK<int> save(maxN);

for (int i = 0; i < n; i++)
        {   cout << "шаг " << i+1 << " символ = " << a[i] << endl;
          if (a[i] == '+')
            save.push(save.pop() + save.pop());
          if (a[i] == '*')
            save.push(save.pop() * save.pop());
          if ((a[i] >= '0') && (a[i] <= '9'))
            save.push(0);
          while ((a[i] >= '0') && (a[i] <= '9'))
                {save.push(10*save.pop() + (a[i++]-'0'));
                cout << "шаг_: " << i+1 << " символ = " << a[i] << endl;
                }
        }
     cout << save.pop() << endl;
   }

// интерфейс АТД "Стек"
// шаблонный класс. Без некоторых проверок...
// и интерфейс, и реализация в этом заголовочном файле
```

```
template <class Item>
class STACK
  {
    private:
      Item *s; int N; int N1; // my N1
    public:
      STACK(int maxN)
        { s = new Item[maxN]; N = 0; N1 = maxN;}   // my N1
      int empty() const
        { return N == 0; }
      void push(Item elem)
        { s[N++] = elem;
            if (N > N1) cout << "ошибка!!!" << endl; // my N1
         }
      Item pop()
        { return s[--N]; }
  };
///}
```

РАЗДЕЛ 4. ПРОГРАММИРОВАНИЕ АЛГОРИТМОВ С БИНАРНЫМИ ДЕРЕВЬЯМИ

4.1. Цель и задачи

Познакомиться с такой часто используемой на практике, особенно при решении задач кодирования и поиска, нелинейной структурой данных, как бинарное дерево, способами её представления и реализации, получить навыки решения задач обработки бинарных деревьев.

Более подробно материал этого раздела представлен в [2].

Определения и представления дерева, леса, бинарного дерева

Деревом называется [5] конечное множество T, состоящее из одного или более узлов, таких, что

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

б) остальные узлы (исключая корень) содержатся в $m \geq 0$ попарно не пересекающихся множествах T1, T2, .., Tm, каждое из которых, в свою очередь,является деревом (поддеревом дерева T).

Это рекурсивное определение дерева.

Лесом называется множество (обычно упорядоченное), состоящее из некоторого (быть может, равного нулю) числа непересекающихся деревьев.

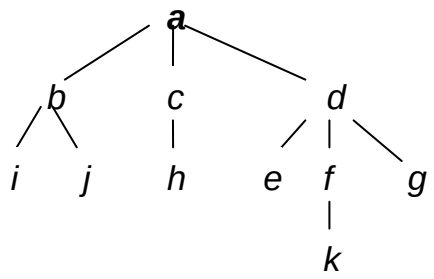
Тогда дернво можнорассматривать как структуру, состоящую из корня и леса поддеревьев.

Дерево T = (корень, лес поддеревьев)

Скобочное представление дерева и леса:

< лес > ::= пусто | < дерево > < лес >,

< дерево > ::= (< корень > < лес >).



(a (b (i) (j)) (c (h)) (d (e) (f (k)) (g)))

Рис. 4.1 Пример дерева и его скобочное представление

Бинарное дерево — это конечное множество узлов, которое либо пусто, либо состоит из корня и двух непересекающихся бинарных деревьев, называемых правым поддеревом и левым поддеревом.

Примером может служить структура на рисунке 4.2.

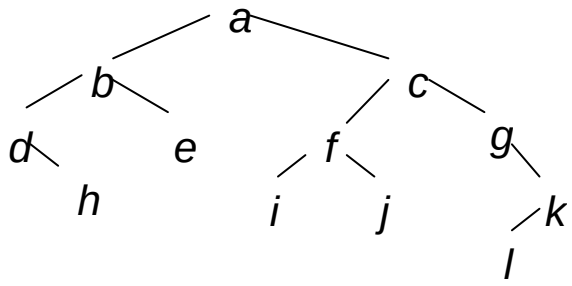


Рис. 4.2 Пример бинарного дерева

Скобочное представление бинарного дерева (БД):

< БД > ::= < пусто > | < непустое БД >,

< пусто > ::= Λ,

< непустое БД > ::= (< корень > < БД > < БД >).

Скобочное представление дерева на рисунке 4.2 выглядит следующим образом:

(a (b (d Λ(h Λ Λ))(e Λ Λ))(c (f (i Λ Λ)(j Λ Λ))(g Λ(k (l Λ Λ) Λ))))

Применив правила упрощения скобочной записи:

1.(<корень> <непустое БД> Λ) ≡ (<корень> <непустое БД>),

2. (< корень > Λ Λ) ≡ (< корень >),

получим (a (b (d Λ (h)) (e)) (c (f (i) (j)) (g Λ (k (l)))))

Естественное соответствие бинарного дерева и леса

Между лесом и бинарным деревом существует естественное сответствие, что позволяет реализовать многие операции над лесом (деревом) при помощи операций над соответствующим ему бинарным деревом.

Пусть лес F типа Forest задан как список деревьев Ti типа Tree (для ∀i ∈ 1..m):

F = (T1 T2 ... Tm).

Head (F) = T1 Tail (F) = (T2 ... Tm).

При этом дерево T1,

а следовтельно и Head (F) = (Root (Head (F)), Listing (Head (F)))

Т.о. Лес F — это совокупность трех частей:

1) корня первого дерева Root (Head (F)),

2) леса поддеревьев первого дерева Listing (Head (F)),

3) леса остальных деревьев Tail (F).

Из этих трех частей рекурсивно порождается бинарное дерево B (F), представляющее лес F:

B (*F*) ≡ **if** *Null* (*F*) **then**

else

ConsBT (*Root* (*Head* (*F*)),

B (*Listing* (*Head* (*F*))),

B (*Tail* (*F*))).

Графически пример такого преобразования изображен на рисунке 4.3.

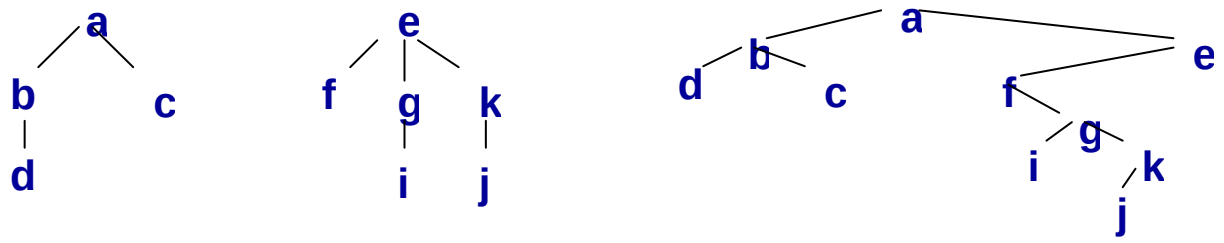


Рис. 4.3. Пример преобразования леса из двух деревьев в бинарное дерево.

Возможно и обратное преобразование бинарного дерева в лес.

F (B) ≡ **if** Null (B) **then** Nil
else Cons (ConsTree (RootBT (B), F (Left (B))),
F (Right (B))).

Обходы бинарных деревьев и леса

Во многих алгоритмах работы с деревьями используется понятие обхода (посещения узлов) бинарного дерева, дерева и леса. В основном используются 3 порядка обхода: прямой (сверху вниз), обратный (слева направо) и концевой (снизу вверх).

Рекурсивные процедуры обхода бинарного дерева в каждом из этих порядков приведены ниже.

```
void обходКЛП   (BTree   b)
{     // прямой порядок обхода
      if ( ! Null (b) )
      {
           посетить (RootBT (b));
           обходКЛП (Left (b));
           обходКЛП (Right (b));
      }
 } // обходКЛП
```

Буква К общзначает корень, Л — левое поддерево, П — правое поддерево.

```
procedure обходЛКП (b: BTree);
{обратный}
 begin
    if not Null (b) then
    begin
       обходЛКП (Left (b));
       посетить (Root (b));
       обходЛКП (Right (b));
    end
 end{обходЛКП};
```

```
procedure обходЛПК (b: BTree);
{концевой}
begin
   if not Null (b) then
   begin
      обходЛПК (Left (b));
      обходЛПК (Right (b));
      посетить ( Root (b));
   end
end{обходЛПК}
```

Повысить эффективность реализации операции обхода бинарных деревьев позволяет использование нерекурсивных процедур.

Общий нерекурсивный алгоритм для всех трех порядков обхода использует стек для хранения упорядоченных пар (<узел бинарного дерева>, <номер операции>). Операции «посетить корень», «пройти левое поддерево», «пройти правое поддерево» нумеруются числами 1, 2, 3 в зависимости от порядка их выполнения в данном варианте обхода. Операция выполняется над узлом при выборе пары из стека. Реализация этих операций рассмотрена в [2].

Нередко кроме перечисленных способов обхода бинарного дерева используется его обход в горизонтальном порядке (в ширину). При таком способе узлы бинарного дерева проходятся слева направо по уровням от корня вниз. Нерекурсивная процедура обхода в ширину, аналогичная процедуре КЛП-обхода (в глубину), но использующая вместо стека очередь приведена ниже.

```
procedure обход_горизонтальный (b: BTree);
   var Q: queue of BTree;
       p: BTree;
begin
     Q := Create;
     Q ← b;
     while not Null (Q) do
     begin
        p ← Q;
        посетить (p);
        if not Null (Left (p)) then Q ← Left (p);
```

```
        if not Null (Right (p)) then Q ← Right (p)
    end{while}
end{обход_горизонтальный}
```

При необходимости обхода леса (дерева) можно сначала построить соответствующее ему бинарное дерево, а затем использовать процедуры обхода бинарного дерева, хотя, конечно, можно написать и непосредственно процедуры обхода леса (дерева)

Реализация бинарных деревьев

Возможна ссылочная реализация абстрактного типа данных (АТД) «бинарное дерево» как в связанной памяти, так и на базе вектора. Каждый узел бинарного дерево (BT) рассматривается как корень поддерева, которому сопоставляется запись из трех полей: одно из них содержит значение узла, а два других являются указателями на левое и правое поддеревья.

Базовые операции бингарного дерева задаются набором функций:
NullBT (t: BinT): Boolean;
RootBT (t: BinT): Elem;
LeftBT (t: BinT): BinT;
RightBT (t: BinT): BinT;
ConsBT (e: Elem; LS, RS: BinT): BinT;

К ним надо добавить функцию **CreateBT**: BinT и процедуру **DestroyBT** (var b: BinT) для начала и завершения работы с экземпляром динамической структуры, а также процедуру **Otkaz**(n: Byte), которая вызывается при попытке некорректного использования основных функций.

Приведенная ниже ссылочная реализация ограниченного бинарного дерева на базе вектора, описана в [2] и во многом напоминает ссылочную реализацию линейного списка на базе вектора.

```
Type Adr = 0 .. MaxAdr;          {диапазон «адресов» в векторе Mem}
    BinT = Adr;                  {представление бинарного дерева}
    Node = record                {узел:}
        Info: Elem;              {  содержимое}
        LSub: BinT;              {  левое поддерево}
        RSub: BinT               {  правое поддерево}
    end {Node};
    Mem = array [Adr] of Node    {вектор для хранения дерева}
```

4.2. Рекомендации по изучению раздела и выполнению индивидуального задания

Прежде чем приступать к выполнению задания из **Приложения 4**, рекомендуется изучить раздел 3 пособия [2].

При выполнении заданий следует использовать варианты реализации бинарного дерева , указанные преподавателем.

Для представления деревьев во входных данных рекомендуется использовать скобочную запись, кроме случаев, специально оговоренных в условии задачи. В выходных данных рекомендуется представлять дерево (лес) в горизонтальном виде (т. е. с поворотом на 90°), а бинарные деревья — в виде уступчатого списка. Например, уступчатый список, представляющий дерево, изображенное на рисунке 4.1, показан на рисунке 4.4.

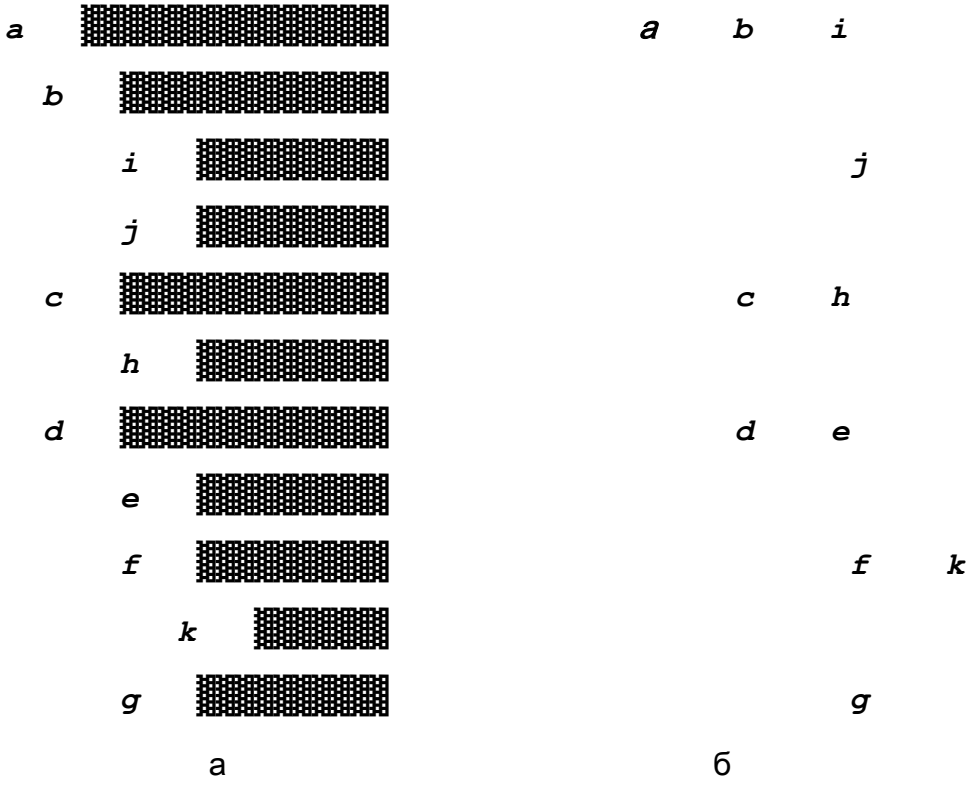


Рис.4.6. Представление дерева: а – в виде уступчатого списка; б – в виде «упрощенного» уступчатого списка

В заданиях 1-3 предлагается реализовать как рекурсивные, так и нерекурсивные процедуры (функции); в последнем случае следует использовать стек и операции над ним.

4.3. Примеры функций работы с бинарными деревьями

1. Подсчитать число узлов бинарного дерева

Число узлов Nv бинарного дерева T равно 0, если T = Λ, и Nv(T) = Nv(TL) + Nv(TR) +1, еслиT ≠ Λ.

```
Nat0 Nv (BinT t)
{
     if (Null(t)) return 0;
     else return (Nv (LeftBT(t)) + Nv (RightBT(t)) +1);
}
```

2. Подсчитать число листьев бинарного дерева

NLeaf(T) = 0, при T = Λ □

NLeaf(T) = 1, при (TL = Λ) & (TR = Λ)

NLeaf(T) = NLeaf(TL) + NLeaf(TR), иначе.

```
Nat0 NLeaf (BinT t):
{
  if (Null(t)) return 0;
  else if (Null(LeftBT(t)) && Null(RightBT(t))) return 1;
     else return (NLeaf (LeftBT(t)) + NLeaf (RightBT(t)));
}
```

3. Определить высоту бинарного дерева

Высота H бинарного дерева T равна 0 при T = Λ и H(T) = max (H(TL), H(TR)) +1 при при T ≠ Λ.

```
Nat0 H ( BinT t)
{
     if (Null(t)) return 0;
     else return (max (H(LeftBT(t)), H(RightBT(t))) +1);
}
```

4. Проверить свойство бинарного дерева: для каждого узла H(TL) – H(TR) = 1

```
Bool   HF (BinT   t)
{    if (Null(t))   return true;
     else
     {
        HL = H(LeftBT(t));
        HR = H(RightBT(t));
        return (HF(LeftBT(t)) && HF(RightBT(t))
                    && ( (HL –   HR) == 1));
     }
```

}

Это неэффективный вариант реализации, поскольку осуществляется многократный обход поддеревьев на одном уровне рекурсивных вызовов. Кроме того, лучше сразу проверить, не равна-ли разность высот поддеревьев единице. Если не равна, вглубь идти не надо, можно сразу вернуть значение false и завершить работу. Если же равна 1, то это не означает, что так будет на всех расположенных ниже уровнях, поэтому надо рекурсивно вызвать процедуру HF для левого и правого поддерева. Попробуйте самостоятельно написать «хороший» вариант реализации функции HF .

4.4. Пример выполнения задания с АТД "Бинарное дерево”

Пусть требуется:

a. осуществить ввод бинарного дерева в КЛП — представлении из файла;

b. вывести КЛП-представление построенного бинарного дерева на экран ;

c. вывести на экран бинарное дерево, повернутое на 90°, т.е. его уступчатое представление;

d. подсчитать высоту бинарного дерева;

e. подсчитать количество узлов бинарного дерева;

f. вывести на экран бинарное дерево в КЛП-порядке;

g. вывести на экран бинарное дерево в ЛКП-порядке;

h. вывести на экран бинарное дерево в ЛПК-порядке.

Для реализации АТД «Бинарное дерево» будем использовать процедурно-модульную парадигму. Выберем ссылочную реализацию бинарного дерева в связанной памяти.

Программа, включающая перечисленные выше операции, содержится в файле work_bt.cpp. Все процедуры и функции, реализующие эти операции, используют исключительно функции для работы с АТД "Бинарное дерево”. Интерфейс АТД "Бинарное дерево” представлен в файле Btree.h, а реализация основных функций для работы с АТД "Бинарное дерево” — в файле bt_implementanion.cpp.

```
// Пример работы с АТД "Бинарное дерево" (в процедурно-модульной парадигме)
#include <iostream>
#include <fstream>
#include <fstream>
```

```
#include <cstdlib>
#include "Btree.h"
#include <windows.h>
using namespace std ;
using namespace binTree_modul;

typedef unsigned int unInt;

binTree enterBT ();
void outBT(binTree b);
void displayBT (binTree b, int n);
unInt hBT (binTree b);
unInt sizeBT (binTree b);
void printKLP (binTree b);
void printLKP (binTree b);
void printLPK (binTree b);

ifstream infile ("KLP.txt");
int main ()
{
    binTree b;

    SetConsoleCP(1251);            // для вывода кирилицы
    SetConsoleOutputCP(1251);   // для вывода кирилицы

    b = enterBT();
    cout << "Бинарное дерево в КЛП-представлении" << endl;
    outBT(b);

    cout << "Бинарное дерево (повернутое): " << endl;
    displayBT (b,1);
    cout << "Высота дерева = " << hBT(b) << endl;
    cout << "Размер (число узлов) дерева = " << sizeBT(b) << endl;

    cout << "Бинарное дерево в КЛП-порядке: " << endl;
    printKLP(b);
    cout << endl;

    cout << "Бинарное дерево в ЛКП-порядке: " << endl;
    printLKP(b);
    cout << endl;

    cout << "Бинарное дерево в ЛПК-порядке: " << endl;
    printLPK(b);
    cout << endl;

    destroy (b);
    outBT(b);
    cout << endl;
return (0);
```

```
}

//-------------------------------------
    binTree enterBT ()
    {   char ch;
        binTree p, q;
        infile >> ch;
        if (ch=='/') return NULL;
        else {p = enterBT(); q = enterBT(); return ConsBT(ch, p, q);}
    }
//-------------------------------------
    void outBT(binTree b)
    {
        if (b!=NULL) {
            cout << RootBT(b);
            outBT(Left(b));
            outBT(Right(b));
        }
        else cout << '/';
    }

//-------------------------------------
    void displayBT (binTree b, int n)
    {   // n   —  уровень узла
        if (b!=NULL) {
            cout << ' ' << RootBT(b);
            if(!isNull(Right(b))) {displayBT (Right(b),n+1);}
            else cout << endl; // вниз
            if(!isNull(Left(b))) {
                for (int i=1;i<=n;i++) cout << "    "; // вправо
                displayBT (Left(b),n+1);}
        }
        else {};
    }
//-------------------------------------
    unInt hBT (binTree b)
    {
        if (isNull(b)) return 0;
        else return max (hBT (Left(b)), hBT(Right(b))) + 1;
    }

//-------------------------------------
    unInt sizeBT (binTree b)
    {
        if (isNull(b)) return 0;
        else return sizeBT (Left(b))+ sizeBT(Right(b)) + 1;
    }
//-------------------------------------
    void printKLP (binTree b)
    {   if (!isNull(b)) {
```

```
            cout << RootBT(b);
            printKLP (Left(b));
            printKLP (Right(b));
        }
    }
//--------------------------------------
    void printLKP (binTree b)
    {   if (!isNull(b)) {
            printLKP (Left(b));
            cout << RootBT(b);
            printLKP (Right(b));
        }
    }
//--------------------------------------
    void printLPK (binTree b)
    {   if (!isNull(b)) {
            printLPK (Left(b));
            printLPK (Right(b));
            cout << RootBT(b);
        }
    }


// интерфейс АТД "Бинарное дерево"(в процедурно-модульной парадигме)
namespace binTree_modul
{

//-------------------------------------
    typedef char base;

    struct node {
        base info;
        node *lt;
        node *rt;
            // constructor
            node () {lt = NULL; rt = NULL;}
    };

    typedef node *binTree; // "представитель" бинарного дерева

    binTree Create(void);
    bool isNull(binTree);
    base RootBT (binTree); // для непустого бин.дерева
    binTree Left (binTree);// для непустого бин.дерева
    binTree Right (binTree);// для непустого бин.дерева
    binTree ConsBT(const base &x, binTree &lst,   binTree &rst);
    void destroy (binTree&);

} // end of namespace binTree_modul
```

```
// Implementation   —  Реализация АТД "Бинарное дерево" (в процедурно-модульной парадигме)
#include <iostream>
#include <cstdlib>
#include "Btree.h"
using namespace std ;

namespace binTree_modul
{

//-------------------------------------
    binTree Create()
    {   return NULL;
    }
//-------------------------------------
    bool isNull(binTree b)
    {   return (b == NULL);
    }
//-------------------------------------
    base RootBT (binTree b)             // для непустого бин.дерева
    {   if (b == NULL) { cerr << "Error: RootBT(null) \n"; exit(1); }
        else return b->info;
    }
//-------------------------------------
    binTree Left (binTree b)        // для непустого бин.дерева
    {   if (b == NULL) { cerr << "Error: Left(null) \n"; exit(1); }
        else return b ->lt;
    }
//-------------------------------------
    binTree Right (binTree b)       // для непустого бин.дерева
    {   if (b == NULL) { cerr << "Error: Right(null) \n"; exit(1); }
        else return b->rt;
    }
//-------------------------------------
    binTree ConsBT(const base &x, binTree &lst,   binTree &rst)
    {   binTree p;
        p = new node;
        if ( p != NULL)  {
            p ->info = x;
            p ->lt = lst;
            p ->rt = rst;
            return p;
        }
        else {cerr << "Memory not enough\n"; exit(1);}
    }
//-------------------------------------
    void destroy (binTree &b)
    {   if (b != NULL)   {
            destroy (b->lt);
            destroy (b->rt);
```

```
            delete b;
            b = NULL;
        }
    }

} // end of namespace h_list
```

РАЗДЕЛ 5. БИНАРНЫЕ ДЕРЕВЬЯ ПОИСКА И АЛГОРИТМЫ СЖАТИЯ

5.1. Цель и задачи

Научиться применять бинарные деревья для решения задач кодирования (сжатия) и поиска

Бинарные деревья поиска

Задача поиска. Пусть задано множество ключей, для ключей определена операция проверки на равенство. Требуется определить входит ли заданный ключ в множество.

Множество может быть представлено различными структурами данных. Определим основные операции АТД «Множество» («Set»):

bool contains(key) — проверка, входит ли в множество ключ со значением *key*
void add(key) — добавление в множество ключа со значением *key*
void remove(key) — удаление из множества ключа со значением *key*

Если с каждым ключом сопоставляются некоторые данные, то можно определить АТД «Словарь» («Map») со следующими основными операциями:

bool contains(key) — проверка входит ли в словарь элемент с ключом key;
void add(key, data) — добавление в словарь элемента со значением data и ключом key;
data value(key) — получение значения элемента с ключом key;
void remove(key) — удаление из словаря элемента с ключом key

Реализация АТД «Множество» на основе неупорядоченного массива представлена в табл. 5.1.

Таблица 5.1

Реализация операций АТД «Множество» на основе неупорядоченного массива

Операция	Реализация	Сложность
bool contains(key)	*Линейный поиск.* Перебираем поочередно все	O(n)

	элементы массива	
void add(key)	Проверка наличия элемента в множестве за O(n) и добавление элемента в конец массива за O(1)	O(n)
void remove(key)	Поиск удаляемого элемента за O(n) и сдвиг элементов, расположенных в массиве после удаляемого элемента за O(n)	O(n)

Если для ключей определена операция сравнения, то можно реализовать множество на основе упорядоченного массива (табл. 5.2).

Таблица 5.2

Реализация операций АТД «Множество» на основе упорядоченного массива

Операция	Реализация	Сложность
bool contains(key)	Бинарный поиск	O(log n)
void add(key)	Проверка наличия элемента в множестве за O(log n) и вставка в упорядоченный массив за O(n)	O(n)
void remove(key)	Поиск удаляемого элемента за O(log n) и сдвиг элементов, расположенных в массиве после удаляемого элемента за O(n)	O(n)

Рассмотрим реализацию множества на основе линейного списка (табл. 5.3).

Таблица 5.3

Реализация операций АТД «Множество» на основе линейного списка

Операция	Реализация	Сложность
bool contains(key)	*Линейный поиск*. Перебираем поочередно все элементы списка	O(n)
void add(key)	Проверка наличия элемента в множестве за O(log n) и вставка в линейный список за O(1)	O(n)
void remove(key)	Поиск удаляемого элемента за O(n) и удаление из линейного списка за O(1)	O(n)

На практике такая сложность основных операций может оказаться неподходящей. Далее мы рассмотрим структуру данных «Бинарное дерево поиска», на основе которой все основные операции АДТ «Множество» можно выполнить за O(log n).

Бинарное дерево называется **бинарным деревом поиска** (БДП), если для каждого узла x выполняется следующее условие: для каждого узла l в левом поддереве справедливо key(l) < key(x), а для каждого узла r в правом поддереве – key(r) > key(x).

Можно дать другое определение БДП: бинарное дерево является БДП, если ключи узлов дерева упорядочены при ЛКП-обходе. Например (рисунок 5.1).

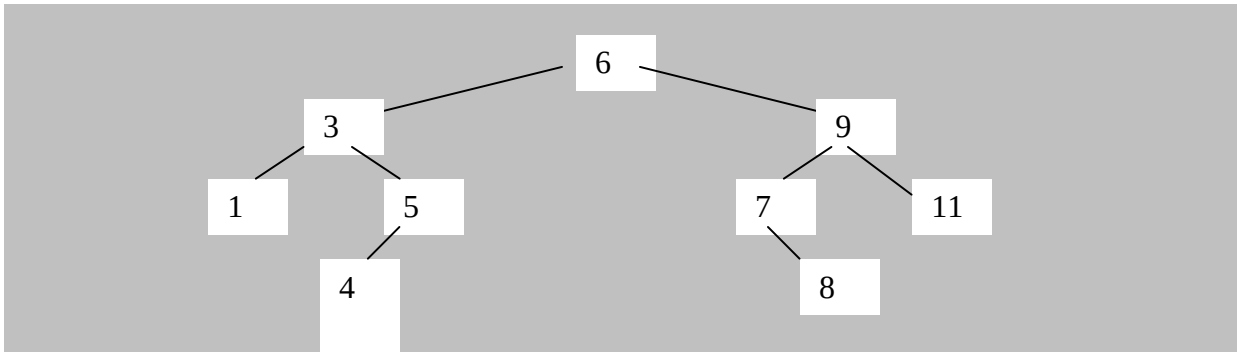


Рис. 5.1. Пример дерева поиска

Во всех бинарных деревьях поиска процедура поиска элемента выполняется по следующему алгоритму. Если корень дерева содержит искомый ключ, то поиск завершен. Иначе, если значение в корне дерева меньше искомого, то выполняем поиск в правом поддереве, иначе – в левом поддереве. Если мы переходим в нулевое поддерево, это означает, что искомого элемента в дереве нет. Эта процедура схожа с бинарным поиском.

Ясно, что время поиска пропорционально высоте дерева – O(h), где h – высота дерева. Поэтому поиск наиболее эффективен в БДП, высота которых минимальна при заданном количестве узлов.

Введем понятие **идеально сбалансированных деревьев.** Идеально сбалансированным назовем такое бинарное дерево, для каждого узла x которого справедливо соотношение □|size(left(x)) - size(right(x))|□≤ 1, где size(left(x)) – количество узлов в левом поддереве узла x, а size(right(x)) – количество узлов в правом поддереве узла x. Примеры некоторых идеально сбалансированных деревьев даны на рисунке 5.2.

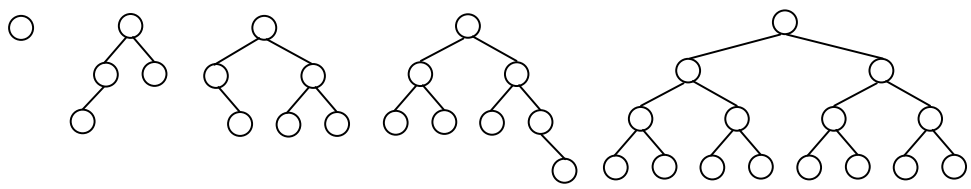


Рис. 5.2. Идеально сбалансированные деревья

В идеально сбалансированном дереве число узлов n и высота дерева h связаны соотношением $2^{h-1}-1<n\le 2^{h}-1$, или в другой форме  .

Если после каждой вставки или удаления восстанавливать свойство идеально сбалансированных деревьев, то операция поиска будет наиболее эффективна. Однако на практике восстанавливать свойство идеальной сбалансированности может быть очень дорого (в некоторых случаях может потребоваться полное изменение структуры дерева), поэтому существуют различные виды самобалансирующихся деревьев, высота которых соответствует оценке *O(log n)*.

Алгоритмы сжатия данных

Пусть задан алфавит A = {α1, α2, …, αn}, где αi – символы алфавита. Пусть имеется сообщение (a1, a2, …, am), ai∈A. В этом сообщении каждый символ алфавита αi встречается wi раз (wi – вес символа αi). Требуется закодировать входное сообщение, т. е. для каждого символа алфавита αi получить кодовое слово ci, и построить выходное сообщение, заменив каждый символ во входном сообщении его кодом. При этом требуется получить выходное сообщение минимальной длины так, чтобы его можно было однозначно декодировать (получить исходное сообщение). Будем рассматривать только двоичные коды (состоящие из нулей и единиц).

Существуют равномерные коды (каждому символу сопоставляется кодовое слово фиксированной длины, при этом декодирование выполняется однозначно) и неравномерные коды (символам могут сопоставляться кодовые слова различной длины, при этом для однозначного декодирования требуется выполнение свойства префиксности кода). Код называется префиксным, если никакое кодовое слово не является префиксом (началом) другого кодового слова.

Рассмотрим пример: Алфавит A = {А, Б, В, Г} и сообщение «АББАГАВ» (соответственно набор весов W = {3, 2, 1, 1}). При использовании равномерного кодирования C = {00, 01, 10, 11} получаем дину сообщения 14 бит. При использовании неравномерного префиксного кода C = {0, 10, 110, 111} получаем длину закодированного сообщения равную 13 бит.

Поставим в соответствие двоичному коду бинарное дерево. Пусть задан код, например, C = {0, 11, 100, 101}. Тогда последовательность нулей и единиц в коде задает путь от корня двоичного дерева к узлу, который содержит соответствующий символ. При этом ноль соответствует переходу к левому ребенку, а единица – к правому. Для префиксного кода символы сопоставляются листьям в кодовом дереве.

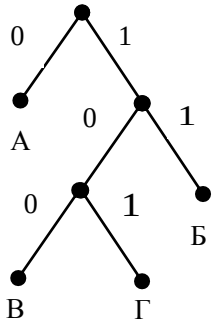


Рис. 5.3. Пример кодового дерева

Пусть построено кодовое дерево. При кодировании кодовое слово для символа αi получают, проходя по дереву от листа, соответствующего αi, к корню. При декодировании расшифровка кодового слова получается движением по кодовому дереву от корня до листа в соответствии с кодовым словом.

Более подробную информацию о деревьях кодирования и поиска вы найдете в работах [3, 6, 8].

5.2. Указания к выполнению задания раздела и примеры его выполнения

При изучении данного раздела студент должен выбрать индивидуальное задание из списка заданий, указанных в **Приложении 5.** Список содержит задания двух видов:

- кодирование/декодирование сообщений
- создание бинарного дерева поиска (БДП) заданного типа с последующим выполнением указанных действий с ним.

В вариантах заданий первого вида (кодирование и декодирование) на вход подается файл с закодированным или незакодированным содержимым.

Требуется раскодировать или закодировать содержимое файла алгоритмом заданного типа.

В вариантах заданий второго вида (БДП) требуется:

- По заданному файлу F (типа file of Elem), все элементы которого различны, построить БДП определенного типа;
- Выполнить одно из следующих действий:

-для построенного БДП проверить, входит ли в него элемент e типа Elem, и если не входит, то добавить элемент e в дерево поиска.

-для построенного БДП проверить, входит ли в него элемент e типа Elem, и если входит, то удалить элемент e из дерева поиска.

-записать в файл элементы построенного БДП в порядке их возрастания; вывести построенное БДП на экран.

-другое действие.

Выбор индивидуального задания к разделу 5 предопределяет и **выбор задания на курсовую работу**. Предлагаются задания на курсовую работу трех типов:

"Демонстрация" — визуализация структур данных, алгоритмов, действий. Демонстрация должна быть подробной и понятной, сопровождаться пояснениями, чтобы программу можно было использовать в обучении для объяснения используемой структуры данных и выполняемых с ней действий.

"Текущий контроль" — создание программы для генерации заданий с ответами к ним для проведения текущего контроля (ТК) среди студентов. Задания и ответы должны выводиться в файл в удобной форме: тексты заданий должны быть готовы для передачи студентам, проходящим ТК; ответы должны обеспечивать удобную проверку правильности выполнения заданий.

"Исследование" — генерация входных данных, использование их для измерения количественных характеристик структур данных, алгоритмов, действий, сравнение экспериментальных результатов с теоретическими.

Реализация структур данных и алгоритмов должна быть выполнена в соответствии с изученными ранее технологическими приемами и стилем программирования.

Задания на курсовую работу представлены в **Приложении 5**.

Пример 1. Случайные бинарные деревья поиска

В случайных БДП структура дерева определяется последовательностью вставок и удалений элементов. При этом высота дерева никак не регулируется, поэтому на некоторых последовательностях операций (например, вставка упорядоченной последовательности элементов), дерево будет вырождаться в линейный список с максимальной высотой, равной n.

Вставка элемента со значением key в дерево tree. Если дерево пусто, создаем новый узел со значением key, иначе, если key(tree)>key то выполняем рекурсивную вставку в левое поддерево tree, иначе — в правое поддерево.

Удаление элемента со значением key из дерева tree. Если удаляемый узел b — лист, то просто удаляем его. Если у узла b один ребенок x, то удаляем узел b, вместо него помещаем узел x. Если у узла b два ребенка, то находим узел x, который непосредственно следует за узлом b при ЛКП обходе, т. е. узел с минимальным значением в правом поддереве. Обмениваем значения узлов b и x и удаляем узел x , имеющий не более одного ребенка (рисунок 5.4).

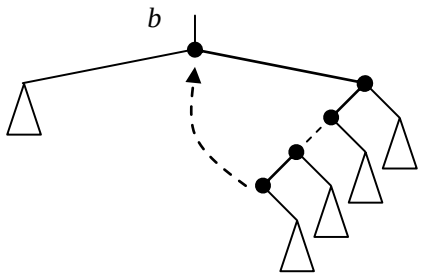


Рис. 5.4. Удаление из дерева узла с двумя детьми

Пример реализации случайных БДП на языке C++:

```
#ifndef BSTREE_H
#define BSTREE_H

template <typename T>
struct Node {
	T key;
	Node* left;
	Node* right;
};
```

```
template <typename T>
using BSTree = Node<T>*;

template <typename T>
BSTree<T> create_bst()
{
	return nullptr;
}

template <typename T>
BSTree<T> create_bst(const T& key, BSTree<T> left,    BSTree<T> right)
{
	BSTree<T> result = new Node<T>;
	result->key = key;
	result->left = left;
	result->right = right;
	return result;
}

template <typename T>
bool empty(const BSTree<T> tree)
{
	return tree == nullptr;
}

template <typename T>
void destroy(BSTree<T>& tree)
{
	if(tree == nullptr) return;

	destroy(tree->left);
	destroy(tree->right);

	delete tree;
	tree = nullptr;
}

template <typename T>
bool contains(const BSTree<T> tree, const T& key)
{
	if(tree == nullptr)
		return false;

	if(key == tree->key)
		return true;

	if(key < tree->key)
		return contains(tree->left, key);
	else
		return contains(tree->right, key);
```

```
}

template <typename T>
bool insert(BSTree<T>& tree, const T& key)
{
	if(tree == nullptr) {
		tree = create_bst<T>(key, nullptr, nullptr);
		return true;
	}

	if(key == tree->key)
		return false;

	if(key < tree->key)
		return insert(tree->left, key);
	else
		return insert(tree->right, key);
}
#endif // BSTREE_H
```

Далее для балансировки деревьев нам потребуется операция вращения, которая сохраняет свойство БДП (упорядоченность ключей дерева при ЛКП-обходе), но изменяет расположение узлов по уровням и, возможно, изменяет высоту дерева.

С помощью операции вращения осуществляется балансировка деревьев после операций вставки и удаления.

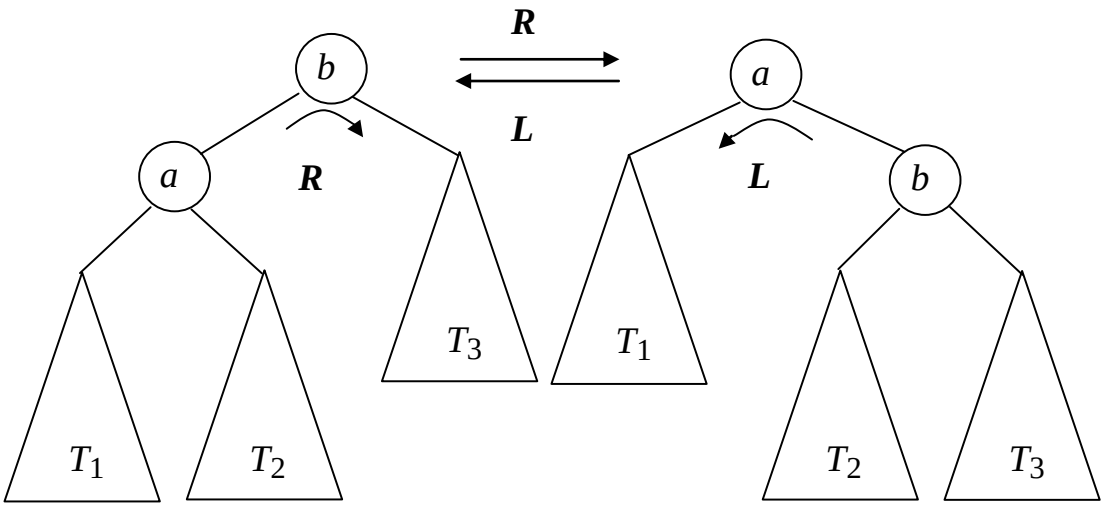


Рис. 5.5. Операция вращения вправо вокруг вершины b и операция вращения влево вокруг вершины a.

Пример 2. Случайные бинарные деревья поиска с рандомизацией

Ключевая идея случайных БДП с рандомизацией состоит в чередовании обычной вставки в дерево поиска и вставки в корень. Чередование происходит случайным (рандомизированным) образом с использованием компьютерного генератора псевдослучайных чисел. Цель такого чередования – сохранить хорошие свойства случайного БДП в среднем и исключить (сделать маловероятным) появление «худшего случая» (поддеревьев большой высоты).

Вставка элемента со значением key в дерево tree. Рассмотрим операцию вставки в корень. Если дерево пусто, создаем новый узел со значением key, иначе, если key(tree)>key то выполняем вставку в корень в левом поддереве tree и выполняем правое вращение, иначе — вставку в корень в правом поддереве и левое вращение. Таким образом узел со значением key становится корнем дерева.

Опишем теперь рандомизированную вставку значением key в дерево tree. Пусть в дереве имеется n узлов. Тогда будем считать, что после добавления еще одного узла любой узел с равной вероятностью может быть корнем дерева. Тогда, с вероятностью 1/(n +1) осуществим вставку в корень, иначе рекурсивно используем рандомизированную вставку в левое или правое поддерево в зависимости от значения ключа key.

Удаление элемента со значением key из дерева tree аналогично случайным БДП.

Пример реализации случайных БДП с рандомизацией на языке C++:

```
#ifndef BSTREE_H
#define BSTREE_H

#include <cassert>
#include <cstdlib>

template <typename T>
struct Node {
	T key;
	Node* left;
	Node* right;
	size_t size;
};

template <typename T>
using BSTree = Node<T>*;
```

```
template <typename T>
BSTree<T> create_bst()
{
	return nullptr;
}

template <typename T>
BSTree<T> create_bst(const T& key, BSTree<T> left,    BSTree<T> right)
{
	BSTree<T> result = new Node<T>;
	result->key = key;
	result->left = left;
	result->right = right;
	result->size = (left ? left->size : 0) + (right ? right->size : 0) + 1;
	return result;
}

template <typename T>
bool empty(const BSTree<T> tree)
{
	return tree == nullptr;
}

template <typename T>
void destroy(BSTree<T>& tree)
{
	if(tree == nullptr) return;

	destroy(tree->left);
	destroy(tree->right);

	delete tree;
	tree = nullptr;
}

template <typename T>
bool contains(const BSTree<T> tree, const T& key)
{
	if(tree == nullptr)
		return false;

	if(key == tree->key)
		return true;

	if(key < tree->key)
		return contains(tree->left, key);
	else
		return contains(tree->right, key);
}
```

```
template <typename T>
bool insert(BSTree<T>& tree, const T& key)
{
	if(tree == nullptr) {
		tree = create_bst<T>(key, nullptr, nullptr);
		return true;
	}

	if(key == tree->key)
		return false;

	if(rand() < RAND_MAX / (tree->size + 1))
		return insert_in_root(tree, key);

	bool inserted = false;
	if(key < tree->key)
		inserted = insert(tree->left, key);
	else
		inserted = insert(tree->right, key);

	if(inserted) tree->size++;
	return inserted;
}

template <typename T>
void left_rotate(BSTree<T>& tree)
{
	if(tree == nullptr)
		return;

	assert(tree->right);

	Node<T>* tmp = tree->right;
	tree->size -= tmp->size;
	tree->size += tmp->left ? tmp->left->size : 0;
	tree->right = tmp->left;
	tmp->size -= tmp->left ? tmp->left->size : 0;
	tmp->size += tree->size;
	tmp->left = tree;
	tree = tmp;
}

template <typename T>
void right_rotate(BSTree<T>& tree)
{
	if(tree == nullptr)
		return;

	assert(tree->left);
```

```
    Node<T>* tmp = tree->left;
    tree->size -= tmp->size;
    tree->size += tmp->right ? tmp->right->size : 0;
    tree->left = tmp->right;
    tmp->size -= tmp->right ? tmp->right->size : 0;
    tmp->size += tree->size;
    tmp->right = tree;
    tree = tmp;
}

template <typename T>
bool insert_in_root(BSTree<T>& tree, const T& key)
{
    if(tree == nullptr) {
        tree = create_bst<T>(key, nullptr, nullptr);
        return true;
    }

    if(key == tree->key)
        return false;

    bool inserted = false;
    if(key < tree->key) {
        inserted = insert_in_root(tree->left, key);
        if(inserted) tree->size++;
        right_rotate(tree);
    } else {
        inserted = insert_in_root(tree->right, key);
        if(inserted) tree->size++;
        left_rotate(tree);
    }
    return inserted;
}
#endif // BSTREE_H
```

Пример 3. Реализация кодового дерева и операций кодирования и декодирования

Файл «code_tree.h»

```
#ifndef CODE_TREE_H
#define CODE_TREE_H

#define MAX_CODE_LEN 1000

struct Symbol {
    char c;
    int weight;
};
```

```
bool symbol_less(const Symbol& l, const Symbol& r);
bool symbol_greater(const Symbol& l, const Symbol& r);

struct CodeTree {
	Symbol s;

	CodeTree* parent;
	CodeTree* left;
	CodeTree* right;
};

CodeTree* make_leaf(const Symbol& s);
CodeTree* make_node(int weight, CodeTree* left, CodeTree* right);

bool is_leaf(const CodeTree* node);
bool is_root(const CodeTree* node);

char* encode(const CodeTree* tree, const char* message);
char* decode(const CodeTree* tree, const char* code);

void destroy(CodeTree* tree);

#endif // CODE_TREE_H
```

Файл «code_tree.cpp»

```
#include "code_tree.h"

#include <climits>
#include <cstring>

bool symbol_less(const Symbol& l, const Symbol& r)
{
	return l.weight < r.weight;
}

bool symbol_greater(const Symbol& l, const Symbol& r)
{
	return l.weight > r.weight;
}

CodeTree* make_leaf(const Symbol& s)
{
	return new CodeTree{ s, nullptr, nullptr, nullptr };
}

CodeTree* make_node(int weight, CodeTree* left, CodeTree* right)
{
	Symbol s{ 0, weight };
	return new CodeTree{ s, nullptr, left, right };
```

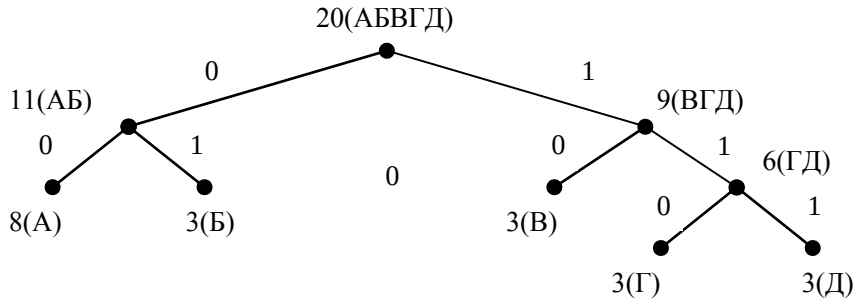
```
}

bool is_leaf(const CodeTree* node)
{
	return node->left == nullptr && node->right == nullptr;
}

bool is_root(const CodeTree* node)
{
	return node->parent == nullptr;
}

static void fill_symbols_map(const CodeTree* node, const CodeTree** symbols_map);

char* encode(const CodeTree* tree, const char* message)
{
	char* code = new char[MAX_CODE_LEN];

	const CodeTree** symbols_map = new const CodeTree*[UCHAR_MAX];
	for(int i = 0; i < UCHAR_MAX; ++i) {
		symbols_map[i] = nullptr;
	}

	fill_symbols_map(tree, symbols_map);

	int len = strlen(message);
	int index = 0;

	char path[UCHAR_MAX];
	for(int i = 0; i < len; ++i) {
		const CodeTree* node = symbols_map[message[i] - CHAR_MIN];
		int j = 0;
		while(!is_root(node)) {
			if(node->parent->left == node)
				path[j++] = '0';
			else
				path[j++] = '1';
			node = node->parent;
		}
		while(j > 0) code[index++] = path[--j];
	}
	code[index] = 0;

	delete [] symbols_map;
	return code;
}

char* decode(const CodeTree* tree, const char* code)
{
	char* message = new char[MAX_CODE_LEN];
```

```
    int index = 0;

    int len = strlen(code);
    const CodeTree* v = tree;

    for(int i = 0; i < len; ++i) {
        if(code[i] == '0')
            v = v->left;
        else
            v = v->right;

        if(is_leaf(v)) {
            message[index++] = v->s.c;
            v = tree;
        }
    }
    return message;
}

void destroy(CodeTree* tree)
{
    if(tree == nullptr) return;

    destroy(tree->left);
    destroy(tree->right);

    delete tree;
    tree = nullptr;
}

void fill_symbols_map(const CodeTree* node, const CodeTree** symbols_map)
{
    if(is_leaf(node))
        symbols_map[node->s.c - CHAR_MIN] = node;
    else {
        fill_symbols_map(node->left, symbols_map);
        fill_symbols_map(node->right, symbols_map);
    }
}
```

Пример 4. Код Фано-Шеннона

Пусть набор весов W упорядочен, а именно: $w_1 \geq w_2 \geq \ldots \geq w_n$. В качестве корня дерева выбирается такой узел (и соответственно набор $W_{1\ldots n}$ разбивается на два поднабора $W_{1\ldots k}$ и $W_{k+1\ldots n}$ так), что веса поддеревьев различаются минимально, т.е. $k = \arg\min_{1\leq q\leq n-1}(|\sum_{i=1\ldots q} w_i - \sum_{i=q+1\ldots n} w_i|)$. Эта процедура повторяется для поддеревьев до тех пор, пока не будет получен лист в качестве текущего поддерева.

Пример. Пусть A = {А, Б, В, Г, Д} и W = {8, 3, 3, 3, 3}). Тогда кодовое дерево Фано-Шеннона есть



Полная длина кода есть 2(8 + 3 + 3) + 3(3 + 3) = 46 бит. Равномерный код (по 3 бита) дал бы суммарную длину 3×20 = 60 бит.

Однако код Фано-Шеннона не является оптимальным. Так, для данного примера существует код C = {0, 100, 101, 110, 111}, который дает длину сообщения 8×1+ 3(3 + 3 + 3 + 3) = 44 бита.

Пример построения кодового дерева с помощью алгоритма Фано-Шеннона на языке C++:

Файл «fh.h»

```
#ifndef FH_H
#define FH_H

#include "code_tree.h"

CodeTree* fanno_shannon(const char* message);
CodeTree* fanno_shannon(const Symbol* symbols, int len);

#endif // FH_H
```

Файл «fh.cpp»

```
#include "fh.h"

#include <algorithm>
#include <climits>
#include <cstring>

static int middle(const Symbol* symbols, int l, int sum, int& lsum, int& rsum);

CodeTree* fanno_shannon(const Symbol* symbols, int l, int r, int sum)
{
    if(l >= r) return nullptr;
```

```
	if(r - l == 1) return make_leaf(symbols[l]);

	int lsum, rsum;
	int m = middle(symbols, l, sum, lsum, rsum);
	CodeTree* ltree = fanno_shannon(symbols, l, m+1, lsum);
	CodeTree* rtree = fanno_shannon(symbols, m+1, r, rsum);

	CodeTree* node = make_node(sum, ltree, rtree);
	ltree->parent = node;
	rtree->parent = node;

	return node;
}

CodeTree* fanno_shannon(const Symbol* symbols, int len)
{
	int sum = 0;
	for(int i = 0; i < len; ++i)
		sum += symbols[i].weight;

	return fanno_shannon(symbols, 0, len, sum);
}

CodeTree* fanno_shannon(const char* message)
{
	Symbol symbols[UCHAR_MAX];
	for(int i = 0; i < UCHAR_MAX; ++i) {
		symbols[i].c = i + CHAR_MIN;
		symbols[i].weight = 0;
	}

	int size = strlen(message);
	for(int i = 0; i < size; ++i)
		symbols[message[i] - CHAR_MIN].weight++;

	std::sort(symbols, symbols + UCHAR_MAX, symbol_greater);

	int len = 0;
	while(symbols[len].weight > 0 && len < UCHAR_MAX) len++;

	return fanno_shannon(symbols, len);
}

int middle(const Symbol* symbols, int l, int sum, int& lsum, int& rsum)
{
	int m = l;
	lsum = symbols[m].weight;
	rsum = sum - lsum;
	int delta = lsum - rsum;
```

```
    while(delta + symbols[m+1].weight < 0) {
        m++;
        lsum += symbols[m].weight;
        rsum -= symbols[m].weight;
        delta = lsum - rsum;
    }
    return m;
}
```

Пример 5. Метод Хаффмана

Пусть набор весов W упорядочен, а именно: $w_1 \geq w_2 \geq \ldots \geq w_n$. Если n = 2, то завершаем процесс кодирования, приписав сообщению с весом w_1 код 1, а сообщению с весом w_2 – код 0. Иначе выполняем следующие действия:

1. Из минимальных весов w_{n-1} и w_n образуем $w'_{n-1} = w_{n-1} + w_n$.
2. Из набора W исключаем элементы w_{n-1} и w_n и добавляем в него новый элемент w'_{n-1}. Полученный набор обозначим W'_{n-1}.
3. Решаем таким же способом задачу с набором весов W'_{n-1}, а затем в полученном решении заменяем узел (лист) w'_{n-1} на поддерево из двух листьев w_{n-1} и w_n, приписав им коды 1 и 0 соответственно.

Пример. Пусть A = {А, Б, В, Г, Д} и W = {8, 3, 3, 3, 3}. Алгоритм Хаффмана по шагам дает следующее:

8(А), 3(Б), 3(В), <u>3(Г), 3(Д)</u>

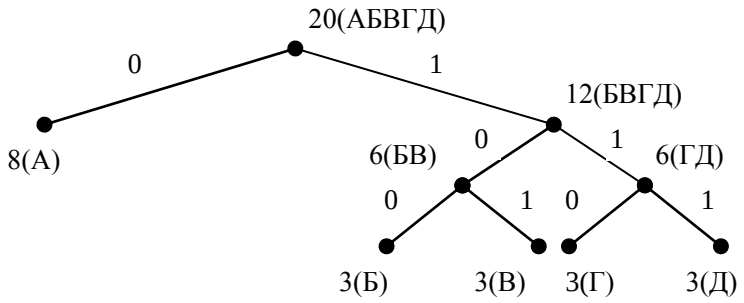
8(А), 6(ГД), <u>3(Б), 3(В)</u>

8(А), <u>6(ГД), 6(БВ)</u>

<u>12(БВГД), 8(А)</u>

20(АБВГД)

Получаем дерево Хаффмана



Длина полученного сообщения равна 44 бита, и это оптимальный код.

Пример построения кодового дерева с помощью алгоритма Хаффмана на языке C++:

Файл «haffman.h»

```
#ifndef HAFFMAN_H
#define HAFFMAN_H

#include "code_tree.h"

CodeTree* haffman(const char* message);
CodeTree* haffman(const char* keys, const int* w, int len);

#endif // HAFFMAN_H
```

Файл «haffman.cpp»

```
#include "haffman.h"

#include "priority_queue.h"

#include <functional>
#include <algorithm>
#include <climits>
#include <cstring>
#include <iostream>

CodeTree* haffman(const Symbol* symbols, int len)
{
	PriorityQueue<CodeTree*>* queue = create_pq<CodeTree*>(len);

	for(int i = 0; i < len; ++i)
		push(queue, symbols[i].weight, make_leaf(symbols[i]));

	while(size(queue) > 1) {
		CodeTree* ltree = pop(queue);
		CodeTree* rtree = pop(queue);
		int weight = ltree->s.weight + rtree->s.weight;
		CodeTree* node = make_node(weight, ltree, rtree);
		ltree->parent = node;
		rtree->parent = node;
		push(queue, weight, node);
	}

	CodeTree* result = pop(queue);
	destroy_pq(queue);
	return result;
}

CodeTree* haffman(const char* message) {

	Symbol symbols[UCHAR_MAX];
	for(int i = 0; i < UCHAR_MAX; ++i) {
```

```
            symbols[i].c = i + CHAR_MIN;
            symbols[i].weight = 0;
        }

        int size = strlen(message);
        for(int i = 0; i < size; ++i)
            symbols[message[i] - CHAR_MIN].weight++;

        std::sort(symbols, symbols + UCHAR_MAX, symbol_greater);

        int len = 0;
        while(symbols[len].weight > 0 && len < UCHAR_MAX) len++;
        return haffman(symbols, len);
}
```

Очередь с приоритетами может быть реализована с помощью структуры данных «куча», например, следующим образом:

```
#ifndef PRIORITY_QUEUE_H
#define PRIORITY_QUEUE_H

#include <utility>

template <typename T>
struct PriorityQueueItem {
        int key;
        T data;
};

template <typename T>
struct PriorityQueue {
        int size_;
        int capacity_;
        PriorityQueueItem<T>* heap_;
};

template <typename T>
PriorityQueue<T>* create_pq(int capacity)
{
        PriorityQueue<T>* pq = new PriorityQueue<T>;
        pq->heap_ = new PriorityQueueItem<T>[capacity];
        pq->capacity_ = capacity;
        pq->size_ = 0;
        return pq;
}

template <typename T>
int size(PriorityQueue<T>* pq)
{
        return pq->size_;
```

```
}

template <typename T>
void sift_up(PriorityQueue<T>* pq, int index)
{
	int parent = (index - 1) / 2;
	while(parent >= 0 && pq->heap_[index].key < pq->heap_[parent].key) {
		std::swap(pq->heap_[index], pq->heap_[parent]);
		index = parent;
		parent = (index - 1) / 2;
	}
}

template <typename T>
bool push(PriorityQueue<T>* pq, int key, const T& data)
{
	if(pq->size_ >= pq->capacity_) return false;

	pq->heap_[pq->size_].key = key;
	pq->heap_[pq->size_].data = data;
	pq->size_++;

	sift_up(pq, pq->size_ - 1);
	return true;
}

template <typename T>
void sift_down(PriorityQueue<T>* pq, int index)
{
	int l = 2 * index + 1;
	int r = 2 * index + 2;
	int min = index;
	if(l < pq->size_ && pq->heap_[l].key < pq->heap_[min].key)
	   min = l;
	if(r < pq->size_ && pq->heap_[r].key < pq->heap_[min].key)
	   min = r;

	if(min != index) {
		std::swap(pq->heap_[index], pq->heap_[min]);
		sift_down(pq, min);
	}
}

template <typename T>
T pop(PriorityQueue<T>* pq)
{
	std::swap(pq->heap_[0], pq->heap_[pq->size_ - 1]);
	pq->size_--;

	sift_down(pq, 0);
```

```
    return pq->heap_[pq->size_].data;
}

template <typename T>
void destroy_pq(PriorityQueue<T>* pq)
{
    delete [] pq->heap_;
    delete pq;
}

#endif // PRIORITY_QUEUE_H
```

ПРИЛОЖЕНИЕ 1. ЗАДАНИЯ К РАЗДЕЛУ РЕКУРСИЯ

1. Для заданных неотрицательных целых n и m вычислить (рекурсивно) биномиальные коэффициенты, пользуясь их определением:

$$C_n^m = \begin{cases} 1, \text{если } m = 0, n > 0 \text{ или } m = n \geq 0, \\ 0, \text{если } m > n \geq 0, \\ C_{n-1}^{m-1} + C_{n-1}^m \text{ в остальных случаях} \end{cases}$$

2. Задано конечное множество имен жителей некоторого города, причем для каждого из жителей перечислены имена его детей. Жители X и Y называются *родственниками*, если (а) либо X – ребенок Y, (б) либо Y – ребенок X, (в) либо существует некоторый Z, такой, что X является родственником Z, а Z является родственником Y. Перечислить все пары жителей города, которые являются родственниками.

3. Имеется n городов, пронумерованных от 1 до n. Некоторые пары городов соединены дорогами. Определить, можно ли попасть по этим дорогам из одного заданного города в другой заданный город. Входная информация о дорогах задается в виде последовательности пар чисел i и j ($i < j$ и $i, j \in 1..n$), указывающих, что i-й и j-й города соединены дорогами.

4. Напечатать все перестановки заданных n различных натуральных чисел (или символов).

5. Функция $f(n)$ определена для целых положительных чисел:

$$f(n) = \begin{cases} 1, & \text{если } n = 1, \\ \sum_{i=2}^{n} f(n \operatorname{div} i), & \text{если } n \geq 2. \end{cases}$$

Вычислить $f(k)$ для k= 15, 16,…, 30.

6. Построить синтаксический анализатор для понятия *простое выражение.*

простое_выражение::=простой_идентификатор |

(*простое_выражение знак_операции простое_выражение*)

простой_идентификатор::= буква

знак_операции:: = – | + | *

7. Построить синтаксический анализатор для понятия *вещественное число*.

вещественное_число::= целое_число . целое_без_знака |

*целое_число.целое_без_знака*E*целое число* |

*целое_число*E*целое_число*

целое_без_знака::=цифра | *цифра целое_без_знака*

целое_число::=целое_без_знака | + *целое_без_знака* | –*целое_без_знака*

8. Построить синтаксический анализатор для понятия *простое_логическое*.

простое_логическое::= TRUE | FALSE |*простой_идентификатор* |
NOT *простое_логическое* |
(*простое_логическое знак_операции простое_логическое*)

простой-идентификатор::=буква

знак-операции::= AND | OR

9. Разработать программу, которая по заданному *простому_логическому* выражению (определение понятия см. в предыдущей задаче), не содержащему вхождений простых идентификаторов, вычисляет значение этого выражения.

10. Построить синтаксический анализатор для определяемого далее понятия *константное_выражение.*

константное_выражение::=ряд_цифр|
константное_выражение знак_операции константное_выражение

знак_операции::=+ | – | *

ряд_цифр::=цифра | *цифра ряд_цифр*

11. Написать программу, которая по заданному (см. предыдущее задание) *константному_выражению* вычисляет его значение либо сообщает о переполнении (превышении заданного значения) в процессе вычислений.

12. Построить синтаксический анализатор для понятия *скобки*.

скобки::=квадратные | *круглые* | *фигурные*

квадратные::=[*круглые фигурные*] | +

круглые::=(*фигурные квадратные*) | –

фигурные::={*квадратные круглые*} | 0

13. Построить синтаксический анализатор для понятия *скобки.*

*скобки::=*A | *скобка скобки*

скобка::= (B *скобки*)

14. Построить синтаксический анализатор для понятия *скобки.*

*скобки::=*A | (B *скобки скобки*)

15. Построить синтаксический анализатор для понятия *скобки.*

*скобки::=*A | A (*ряд_скобок*)

ряд_скобок::= скобки | *скобки* ; *ряд_скобок*

16. Построить синтаксический анализатор для понятия *скобки*.

*скобки::=*A | B | (*скобки скобки*)

17. Функция Ф преобразования текста определяется следующим образом (аргумент функции – это текст, т. е. последовательность символов):

$$\Phi(\alpha)=\begin{cases}\Phi(\gamma)\beta, \text{ если } \alpha=\beta/\gamma \text{ и текст } \beta \text{ не содержит вхождений символа «/»},\\ \alpha, \text{ если в } \alpha \text{ нет вхождений символа «/».}\end{cases}$$

Например: Ф(«ла/ска»)=«скала», Ф(«б/ру/с»)=«сруб», Ф(«ца/ри/ца»)= «царица», Ф(«ум/ри/ва/к/а»)= «аквариум». Реализовать функцию Ф рекурсивно.

18. Пусть определена функция Ф преобразования целочисленного вектора α:

$$\Phi(\alpha)=\begin{cases}\alpha, \text{ если} \|\alpha\|=1,\\ ab, \text{ если} \|\alpha\|=2, \alpha=ab \text{ и } a\le b,\\ ba, \text{ если} \|\alpha\|=2, \alpha=ab \text{ и } b<a,\\ \Phi(\beta)\Phi(\gamma) \text{ , если } \|\alpha\|>2, \alpha=\beta\gamma, \text{где } \|\beta\|=\|\gamma\| \text{ или } \|\beta\|=\|\gamma\|+1.\end{cases}$$

Например: Ф(1,2,3,4,5) = 1,2,3,4,5; Ф(4,3,2,1) = 3,4,1,2; Ф(4,3,2) = 3,4,2. Отметим, что функция Ф преобразует вектор, не меняя его длину. Реализовать функцию Ф рекурсивно.

19. Функция Ф преобразования целочисленного вектора α определена следующим образом:

$$\Phi(\alpha)=\begin{cases}\alpha, \text{если} \|\alpha\|\le 2,\\ \Phi(\beta)\Phi(\gamma), \text{если} \alpha=\beta\gamma, \|\beta\|=\|\gamma\|, \|\alpha\|>2,\\ \Phi(\beta\, a)\,\Phi(a\,\gamma), \text{ если} \alpha=\beta a\gamma, \|\beta\|=\|\gamma\|, \|\alpha\|>2, \|a\|=1.\end{cases}$$

Например: Ф(1,2,3,4,5) = 1,2,2,3,3,4,4,5; Ф(1,2,3,4,5,6) = 1,2,2,3,4,5,5,6; Ф(1,2,3,4,5,6,7) = 1,2,3,4,4,5,6,7; Ф(1,2,3,4,5,6,7,8) = 1,2,3,4,5,6,7,8. Отметим, что функция Ф может удлинять вектор. Реализовать функцию Ф рекурсивно.

20. Построить синтаксический анализатор понятия *список_параметров.*

список_параметров::= параметр | *параметр , список_параметров*

параметр::= имя=цифра цифра | *имя*=(*список_параметров*)

имя::= буква буква буква

21. Построить синтаксический анализатор для понятия *скобки*.

скобки::=квадратные | *круглые*

квадратные:: = [[*квадратные*] (*круглые*)] | B

круглые::=((*круглые*) [*квадратные*]) | A

22. Построить синтаксический анализатор для определенного далее понятия *логическое_выражение*.

логическое_выражение ::= TRUE | FALSE |*идентификатор* |

NOT (*операнд*) | *операция* (*операнды*)

идентификатор::=буква

операция::= AND | OR

операнды::= операнд | *операнд, операнды*

операнд::= логическое_выражение

23. Разработать программу, которая, имея на входе заданное (см. предыдущее задание) *логическое_выражение*, не содержащее вхождений идентификаторов, вычисляет значение этого выражения и печатает само выражение и его значение.

24. Построить синтаксический анализатор для понятия *текст_со_скобками.*

текст_со_скобками::=элемент | *элемент текст_со_скобками*

*элемент::=*A | B | (*текст_со_скобками*) | [*текст_со_скобками*] |

{ *текст_со_скобками* }

25. Построить синтаксический анализатор для параметризованного понятия *скобки(T)*, где *T* – заданное конечное множество, а круглые скобки «(» и «)» не являются терминальными символами, а отражают зависимость определяемого понятия от параметра *T*.

скобки(T)::=элемент(T) | *список(скобки(T))*

*список(E)::=*N | [*ряд(E)*]

ряд(E)::=элемент(E) | *элемент(E) ряд(E)*

ПРИЛОЖЕНИЕ 2. ЗАДАНИЯ К РАЗДЕЛУ РЕКУРСИВНАЯ ОБРАБОТКА ИЕРАРХИЧЕСКИХ СПИСКОВ

Решить следующие задачи с использованием базовых функций рекурсивной обработки списков:

1. проверить иерархический список на наличие в нем заданного элемента (атома) *x*;

2. удалить из иерархического списка все вхождения заданного элемента (атома) *x*;

3. заменить в иерархическом списке все вхождения заданного элемента (атома) *x* на заданный элемент (атом) *y*;

4. подсчитать число атомов в иерархическом списке; сформировать линейный список атомов, соответствующий порядку подсчета;

5. подсчитать число различных атомов в иерархическом списке; сформировать из них линейный список;

6. сформировать линейный список атомов исходного иерархического списка путем устранения всех внутренних скобок в его сокращенной скобочной записи;

7. вычислить глубину (число уровней вложения) иерархического списка как максимальное число одновременно открытых левых скобок в сокращенной скобочной записи списка; принять, что глубина пустого списка и глубина атомарного *S*-выражения равны нулю; например, глубина списка (***a*** (***b*** ( ) ***c***) ***d***) равна двум;

8. обратить иерархический список на всех уровнях вложения; например, для исходного списка (***a*** (***b c***) ***d***) результатом обращения будет список (***d*** (***c b***) ***a***);

9. проверить структурную идентичность двух иерархических списков (списки структурно идентичны, если их устройство (скобочная структура и количество элементов в соответствующих подсписках) одинаково, при этом атомы могут отличаться).

10. Пусть выражение (логическое, арифметическое) представлено иерархическим списком. В выражение входят константы и переменные, которые являются атомами списка. Операции представляются в префиксной форме, т.е. **(<операция> <аргументы>)**. Аргументов может быть 1, 2 и более. Например, (+ a (* b (- c))) или (OR a (AND b (NOT c))).

11. Проверить синтаксическую корректность арифметического выражения, в которое могут входить любые 2 из операций +, -, *, /.

12. Упростить (преобразовать) арифметическое выражение. Например, (+ 0 (* 1 (+ a b))) преобразуется в (+ a b).

13. Вычислить арифметическое выражение. На вход подается список значений переменных ((x1 c1) (x2 c2) … (xk ck)), где xi – переменная, а ci – ее значение (константа) целого типа.

14. Проверить синтаксическую корректность логического выражения.

15. Вычислить логическое выражение.На вход подается список значений переменных ((x1 c1) (x2 c2) … (xk ck)), где xi – переменная, а ci – ее значение (константа) логического типа.

Примечание. Во всех заданиях для ввода и вывода списков, можно использовать функции ввода-вывода, рассмотренные в этом пособии. Можно усовершенствовать их, например, для функции ввода, добавив синтаксический анализ.

ПРИЛОЖЕНИЕ 3. ЗАДАНИЯ К РАЗДЕЛУ ЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ: СТЕК, ОЧЕРЕДЬ, ДЕК.

1. За один просмотр заданного файла *F* (типа **file of** *Real*) и без использования дополнительных файлов вывести элементы файла *F* в следующем порядке: сначала - все числа, меньшие *a*, затем - все числа на отрезке [*a, b*] и наконец - все остальные числа, сохраняя исходный взаимный порядок в каждой из этих групп чисел (*a* и *b* задаются пользователем, $a < b$).

2. Содержимое заданного текстового файла *F*, разделенного на строки, переписать в текстовый файл *G*, перенося при этом в конец каждой строки все входящие в нее цифры (с сохранением исходного взаимного порядка как среди цифр, так и среди остальных литер строки).

3. Рассматриваются следующие типы данных:

type *имя* = (*Анна, ..., Яков*);

дети = **array** [*имя, имя*] **of** *Boolean*;

потомки = **file of** *имя*.

Задан массив *Д* типа *дети* (*Д* [*x, y*] = *true*, если человек по имени *y* является ребенком человека по имени *x*). Для введенного пользователем имени *И* записать в файл *П* типа *потомки* имена всех потомков человека с именем *И* в следующем порядке: сначала - имена всех его детей, затем - всех его внуков, затем - всех правнуков и т. д.

4. Содержимое заданного текстового файла F, разделенного на строки, переписать в текстовый файл G, выписывая литеры каждой строки в обратном порядке.

5. Правильная скобочная конструкция с тремя видами скобок определяется как

< текст > ::= < пусто > | < элемент > < текст >

< элемент > ::= < символ > | (< текст >) | [< текст >] | { < текст > }

где < символ > - любой символ, кроме (,) , [,] , { , }. Проверить, является ли текст, содержащийся в заданном файле F, правильной скобочной конструкцией; если нет, то указать номер ошибочной позиции.

6. Проверить, является ли содержимое заданного текстового файла F правильной записью формулы следующего вида:

< формула > ::= < терм > | < терм > + < формула > | < терм > - < формула >

< терм > ::= < имя > | (< формула >) | [< формула >] | { < формула > }

< имя > ::= x | y | z

Если не является, то указать номер ошибочной позиции.

7. В заданном текстовом файле F записана формула вида

< формула > ::= < цифра > | M (< формула > , < формула >) |

m (< формула > , < формула >)

< цифра > ::= 0 | 1 | ... | 9

где M обозначает функцию max, а m – функцию min. Вычислить (как целое число) значение данной формулы. Например, M (5, m(6, 8)) = 6.

8. В заданном текстовом файле F записано логическое выражение (ЛВ) в следующей форме:

<ЛВ>::= true | false | (Ø< ЛВ >) | (< ЛВ > Ù < ЛВ >) | (< ЛВ > Ú < ЛВ >)

где знаки Ø, Ù и Ú обозначают соответственно отрицание, конъюнкцию и дизъюнкцию. Вычислить (как Boolean) значение этого выражения.

9. В заданном текстовом файле F записан текст, сбалансированный по круглым скобкам:

< текст > ::= < пусто > | < элемент > < текст >

< элемент > ::= < символ > | (< текст >)

где < символ > – любой символ, кроме (,). Для каждой пары соответствующих открывающей и закрывающей скобок вывести номера их позиций в тексте, упорядочив пары в порядке возрастания номеров позиций:

а) закрывающих скобок; б) открывающих скобок.

Например, для текста A + (45 - F (X) * (B - C)) надо напечатать:

а) 8 10; 12 16; 3 17; б) 3 17; 8 10; 12 16.

10. Определить, имеет ли заданная в файле F символьная строка следующую структуру:

a D b D c D d ...,

где каждая строка a, b, c, d, ..., в свою очередь, имеет вид x1 C x2. Здесь x1 есть строка, состоящая из символов A и B, а x2 - строка, обратная строке x1 (т. е. если x1 = ABABB, то x2 = BBABA). Таким образом, исходная строка состоит только из символов A, B, C и D. Исходная строка может читаться только последовательно (посимвольно) слева направо.

11. Рассматривается выражение следующего вида:

< выражение >::= < терм > | < терм > + < выражение > |

< терм > - < выражение >

< терм > ::= < множитель > | < множитель > * < терм >

< множитель > ::= < число > | < переменная > | (< выражение >) |

< множитель > ^ < число >

< число > ::= < цифра >

< переменная > ::= < буква >

Такая форма записи выражения называется инфиксной.

Постфиксной (префиксной) формой записи выражения aDb называется запись, в которой знак операции размещен за (перед) операндами: abD (Dab).

Примеры

Инфиксная	Постфиксная	Префиксная
a–b	ab–	–ab
a*b+c	ab*c+	+*abc
a*(b+c)	abc+*	*a+bc
a+b^c^d*e	abc^d^e*+	+a*^b^cde.

Постфиксная и префиксная формы записи выражений не содержат скобок.

Требуется:

а) вычислить как целое число значение выражения (без переменных), записанного в постфиксной форме в заданном текстовом файле postfix;

б) то же для выражения в префиксной форме (задан текстовый файл prefix);

в) перевести выражение, записанное в обычной (инфиксной) форме в заданном текстовом файле infix, в постфиксную форму и в таком виде записать его в текстовый файл postfix;

г) то же, но в префиксную форму (записать в файл prefix);

д) вывести в обычной (инфиксной) форме выражение, записанное в постфиксной форме в заданном текстовом файле postfix (рекурсивные процедуры не использовать и лишние скобки не выводить);

е) то же, но при заданной префиксной форме (в файле prefix).

ПРИЛОЖЕНИЕ 4. ЗАДАНИЯ К РАЗДЕЛУ ПРОГРАММИРОВАНИЕ АЛГОРИТМОВ С БИНАРНЫМИ ДЕРЕВЬЯМИ

1. Задано бинарное дерево *b* типа *BT* с типом элементов *Elem*. Для введенной пользователем величины *E* (**var** *E*: *Elem*):

а) определить, входит ли элемент *E* в дерево *b*;

б) определить число вхождений элемента *E* в дерево *b*;

в) найти в дереве *b* длину пути (число ветвей) от корня до ближайшего узла с элементом *E* (если *E* не входит в *b*, за ответ принять -1).

2. Для заданного бинарного дерева *b* типа *BT* с произвольным типом элементов:

а) определить максимальную глубину дерева *b*, т. е. число ветвей в самом длинном из путей от корня дерева до листьев;

б) вычислить длину внутреннего пути дерева *b*, т. е. сумму по всем узлам длин путей от корня до узла;

в) напечатать элементы из всех листьев дерева *b*;

г) подсчитать число узлов на заданном уровне *n* дерева *b* (корень считать узлом 1-го уровня);

д) определить, есть ли в дереве *b* хотя бы два одинаковых элемента.

3. Заданы два бинарных дерева *b*1 и *b*2 типа *BT* с произвольным типом элементов. Проверить:

а) подобны ли они (два бинарных дерева ***подобны***, если они оба пусты либо они оба непусты и их левые поддеревья подобны и правые поддеревья подобны);

б) равны ли они (два бинарных дерева ***равны***, если они подобны и их соответствующие элементы равны);

в) зеркально подобны ли они (два бинарных дерева ***зеркально подобны***, если они оба пусты, либо они оба непусты и для каждого из них левое поддерево одного подобно правому поддереву другого);

г) симметричны ли они (два бинарных дерева симметричны, если они зеркально подобны и их соответствующие элементы равны).

4. Задано бинарное дерево *b* типа *BT* с произвольным типом элементов. Используя очередь и операции над ней, напечатать все элементы дерева *b* по уровням: сначала - из корня дерева, затем (слева направо) - из узлов, сыновних

по отношению к корню, затем (также слева направо) - из узлов, сыновних по отношению к этим узлам, и т. д.

5**.** Для заданного леса с произвольным типом элементов:

а) получить естественное представление леса бинарным деревом;

б) вывести изображение леса и бинарного дерева;

в) перечислить элементы леса в горизонтальном порядке (в ширину).

6. (Обратная задача.) Для заданного бинарного дерева с произвольным типом элементов:

а) получить лес, естественно представленный этим бинарным деревом;

б) вывести изображение бинарного дерева и леса;

в) см. п. 5, в.

7. Рассматриваются бинарные деревья с элементами типа *char*. Заданы перечисления узлов некоторого дерева *b* в порядке КЛП и ЛКП. Требуется:

а) восстановить дерево *b* и вывести его изображение;

б) перечислить узлы дерева *b* в порядке ЛПК.

8**.** Рассматриваются бинарные деревья с элементами типа *char*. Заданы перечисления узлов некоторого дерева *b* в порядке ЛКП и ЛПК. Требуется:

а) восстановить дерево *b* и вывести его изображение;

б) перечислить узлы дерева *b* в порядке КЛП.

9. Формулу вида

< формула > ::= < терминал > | (< формула > < знак > < формула >)

< знак > ::= + | - | *

< терминал > ::= 0 | 1 | ... | 9 | *a* | *b* | ... | *z*

можно представить в виде бинарного дерева («***дерева-формулы***») с элементами типа *char* согласно следующим правилам:

- формула из одного терминала представляется деревом из одной вершины с этим терминалом;

- формула вида (f_1 *s* f_2) представляется деревом, в котором корень - это знак *s*, а левое и правое поддеревья - соответствующие представления формул f_1 и f_2. Например, формула (5 * (*a* + 3)) представляется деревом-формулой, показанной на рис. П4.1.

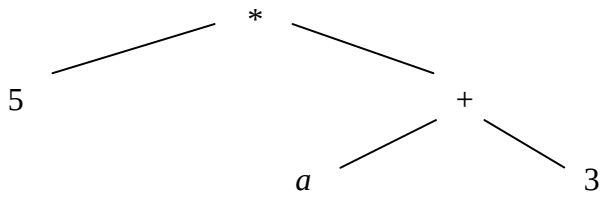


Рис. П4.1. Дерево-формула

Требуется:

а) для заданной формулы *f* построить дерево-формулу *t*;

б) для заданного дерева-формулы *t* напечатать соответствующую формулу *f*;

в) с помощью построения дерева-формулы *t* преобразовать заданную формулу *f* из инфиксной формы в префиксную (перечисление узлов *t* в порядке КЛП) или в постфиксную (перечисление в порядке ЛПК);

г) выполнить с формулой *f* преобразования, обратные преобразованиям из п. в);

д) если в дереве-формуле *t* терминалами являются только цифры, то вычислить (как целое число) значение дерева-формулы *t*;

е) упростить дерево-формулу *t*, заменяя в нем все поддеревья, соответствующие формулам (*f* + 0), (0 + *f*), (*f* - 0), (*f* * 1), (1 * *f*), на поддеревья, соответствующие формуле *f*, а поддеревья, соответствующие формулам (*f* * 0) и (0 * *f*), - на узел с 0;

ж) преобразовать дерево-формулу *t*, заменяя в нем все подде-ревья, соответствующие формулам (f_1 * (f_2 + f_3)) и ((f_1 + f_2) * f_3), на поддеревья, соответствующие формулам ((f_1 * f_2) + (f_1 * f_3)) и ((f_1 * f_3) + (f_2 * f_3));

з) выполнить в дереве-формуле *t* преобразования, обратные преобразованиям из п. ж);

и) построить дерево-формулу *t*1 - производную дерева-формулы *t* по заданной переменной.

ПРИЛОЖЕНИЕ 5. ЗАДАНИЯ К РАЗДЕЛУ БИНАРНЫЕ ДЕРЕВЬЯ ПОИСКА И К КУРСОВОЙ РАБОТЕ

№ вар.	Лабораторная. работа	Курсовая работа
1	Кодирование: Фано-Шеннона	Статическое кодирование и декодирование текстового файла методами Хаффмана и Фано-Шеннона – **демонстрация**
3	Кодирование: статическое Хаффмана	Динамическое кодирование и декодирование по Хаффману – сравнительное **исследование** со “статическим” методом.
4	Декодирование: статическое Хаффмана	Кодирование и декодирование методами Хаффмана и Фано-Шеннона – **исследование**
5	Кодирование: динамическое Хаффмана	Динамическое кодирование и декодирование по Хаффману – **демонстрация**
6	Декодирование: динамическое Хаффмана	Динамическое кодирование и декодирование по Хаффману – **текущий контроль**
7	БДП: случайное* БДП; действие: 1+2а	Случайные БДП – вставка и исключение. **Демонстрация**
8	БДП: случайное* БДП; действие: 1+2б	Случайные БДП – вставка и исключение. **Исследование** (в среднем, в худшем случае)
9	БДП: случайное* БДП; действие: 1+2в	Случайные БДП – вставка и исключение. **Текущий контроль**
10	БДП: случайное* БДП с рандомизацией; действие: 1+2а	Случайные БДП с рандомизацией. **Демонстрация**
11	БДП: случайное* БДП с рандомизацией; действие: 1+2б	Случайные БДП с рандомизацией. **Исследование** (в среднем, в худшем случае)
12	БДП: случайное* БДП с рандомизацией; действие: 1+2в	Случайные БДП с рандомизацией. **Текущий контроль**
13	БДП: Рандомизированная пирамида поиска (treap); действие: 1+2а	Рандомизированные пирамиды поиска (Treaps) – вставка и исключение. **Демонстрация**
14	БДП: Рандомизированная пирамида поиска (treap); действие: 1+2б	Рандомизированные пирамиды поиска (Treaps) – вставка и исключение. **Исследование** (в среднем, в худшем случае)
15	БДП: Рандомизированная пирамида поиска (treap); действие: 1+2в	Рандомизированные пирамиды поиска (Treaps) – вставка и исключение. **Текущий контроль**

16	БДП: АВЛ-дерево; действие: 1+2а	АВЛ-деревья ─ вставка и исключение. **Демонстрация**
17	БДП: АВЛ-дерево; действие: 1+2б	АВЛ-деревья ─ вставка и исключение. **Исследование** (в среднем, в худшем случае)
18	БДП: АВЛ-дерево; действие: 1+2в	АВЛ-деревья ─ вставка и исключение. **Текущий контроль**
19	БДП: Рандомизированная пирамида поиска (treap); действие: 1+2г: сцепление двух пирамид	Сцепляемые очереди (упорядоченные линейные списки) на основе рандомизированных пирамид поиска ─ операции сцепления и расщепления. **Демонстрация**
20	БДП: Рандомизированная пирамида поиска (treap); действие: 1+2г: расщепление пирамиды	Сцепляемые очереди (упорядоченные линейные списки) на основе рандомизированных пирамид поиска ─ операции сцепления и расщепления. **Исследование**

Список литературы

1. Представление и обработка структурированных данных. Практикум по программированию. /С. А. Ивановский, В.А. Калмычков, а.а. Лисс, В.П. Самойленко. – СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2002.

2. Динамические структуры данных: учеб. пособие / А. Ю. Алексеев, С. А. Ивановский, Д. В. Куликов. – СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2004.

3. Ивановский С. А. Деревья кодирования и поиска: Учеб. пособие. СПб. : Изд-во СПбГЭТУ «ЛЭТИ», 2006.

4. Алексеев А. Ю., Ивановский С. А., Фролова С. А. Алгоритмы сортировки: учебное пособие. СПб. : Изд-во СПбГЭТУ «ЛЭТИ», 2009.

5. Кнут Д. Искусство программирования. Том 1: Основные алгоритмы : пер. с англ. – 3-е изд., испр. и доп. – М. : Издательский дом «Вильямс», 2007. (Доп. тираж 2009 г.)

6. Кнут Д. Искусство программирования. Том 3: Сортировка и поиск : пер. с англ. – 2-е изд., испр. и доп. – М. : Издательский дом «Вильямс», 2007.

7. Алгоритмы: построение и анализ : пер. с англ. / Т. Кормен и др. – 2-е изд. – М. : Издательский дом «Вильямс», 2007, 2009.

8. Вирт Н. Алгоритмы и структуры данных. Новая версия для Оберона +CD / пер. с англ. Ф. В. Ткачева. – М. : ДМК Пресс, 2010.

9. Бьерн Страуструп. Язык программирования C++ http://lib.ru/CPPHB/cpptut.txt

10. В. Е. Алексеев, В. А. Таланов Структуры данных и модели вычислений
http://www.intuit.ru/department/algorithms/dscm/ (ИНТУИТ)

11. Инструменты, алгоритмы и структуры данных Автор: Б. Мейер

12. http://www.intuit.ru/department/se/ialgdate/ (ИНТУИТ)

Содержание

Учебно-методическое пособие

Ивановский Сергей Алексеевич
Фомичева Татьяна Генриховна
Шолохова Ольга Михайловна

Методические указания к лабораторным работам

Издание публикуется в авторской редакции

СПбГЭТУ «ЛЭТИ»
197376, Санкт-Петербург, ул. Проф. Попова, 5