16

Йерархични структури от данни двоично дърво и граф

16.1 Двоично дърво

16.1.1 Дефиниране на двоично дърво

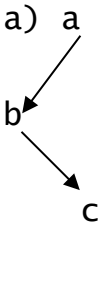
Логическо описание

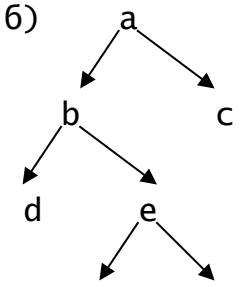
Двоично дърво от тип T е структура от данни, която е или празна, или е образувана от

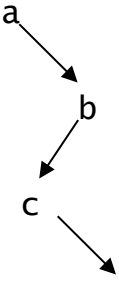
- данна от тип T, наречена **корен** (връх, възел) на двоичното дърво от тип T;
- двоично дърво от тип T, наречено **ляво поддърво** на двоичното дърво от тип T (ЛПД);
- двоично дърво от тип T, наречено **дясно поддърво** на двоичното дърво от тип T (ДПД).

Примери:

Нека a, b, c, d, e, f и g са данни от тип T. Тогава следните графични представяния определят двоични дървета от тип T.

а) 

б) 

в) 

f g d

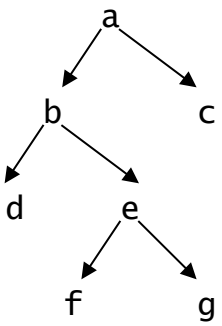
Посоката на линиите, свързващи върховете с поддървета, позволява да се различи ляво от дясно поддърво. Двоичните дървета

са различни. В единия случай дясното поддърво е празно, а в другия - дясното дърво не е празно.

Някои определения

Листо - това е връх с празни поддървета.

Пример: Върховете d, c, f и g на двоичното дърво



са листа.

Върховете, които не съвпадат с корена и листата, се наричат **вътрешни върхове**.

Пример: b и e, от примера по-горе, са вътрешни върхове на двоичното дърво от тип T.

Всяко поддърво се нарича **наследник (син)** по отношение на своето дърво и **родител (баща)** по отношение на своите поддървета.

На всеки връх може да се съпостави "**ниво**". Приемаме, че коренът има ниво 1 (или 0) и ако един връх има ниво i и има наследници, то те имат ниво i+1. Така **нивото на връх** е всъщност броя на върховете, които трябва да бъдат обходени като се започне от корена и се стигне до върха. Най-голямото ниво на двоично дърво, се нарича негова **височина** (**дълбочина**).

Над структурата от данни двоично дърво са възможни следните операции:

- достъп до връх

Възможен е пряк достъп до корена и непряк достъп до всеки от останалите върхове на двоичното дърво.

- включване и изключване на връх

Включването и изключването са възможни в произволно място на двоичното дърво, но в резултат трябва да се получи двоично дърво от тип T.

Обхождане на двоично дърво

Обхождането на двоично дърво дава метод, позволяващ да се осъществи достъп до всеки връх на дървото един единствен път.

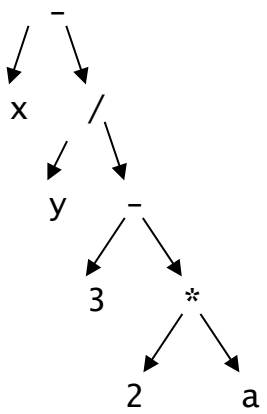
Осъществява се чрез изпълнение на следните три действия в някаква последователност:

- обработка на корена;
- обхождане на лявото поддърво;
- обхождане на дясното поддърво,

т.е. процесът на обхождане е рекурсивен. Съществуват шест различни начина за обхождане на двоично дърво: ЛКД (**смесено обхождане**), КЛД (**низходящо обхождане**), ЛДК (**възходящо обхождане**), ДКЛ, КДЛ и ДЛК, където К - означава корен, Л - ляво поддърво, Д - дясно поддърво. Обхождането ЛКД например означава, че първо се обхожда лявото поддърво, след него се обработва коренът и накрая се обхожда дясното поддърво.

Всеки аритметичен израз може да се представи чрез двоично дърво, в листата на което са операндите, а във вътрешните върхове и корена - операциите.

Пример: Изразът x-y/(3-2*a) може да се представи чрез двоичното дърво:



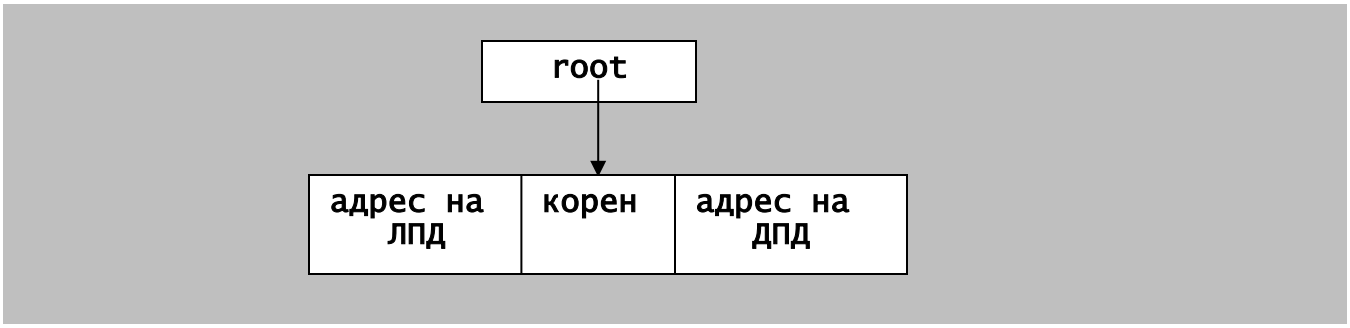
Възходящият обход на двоично дърво, представящо аритметичен израз, дава обратния полски запис на аритметичния израз. Смесеният обход на двоично дърво, представящо аритметичен израз, дава общоприетия (инфиксен) запис на аритметичния израз, но без скобите, а низходящият обход на двоично дърво, представящо аритметичен израз, дава представяне, което се нарича **прав полски запис**.

Физическо представяне

Използват се главно два начина за физическо представяне на двоично дърво от тип T - **свързано** и **верижно.**

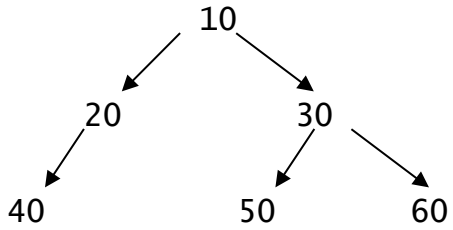
Свързано представяне

Реализира се чрез указател към кутия с три полета: информационно, съдържащо стойността на корена и две адресни, съдържащи представянията на ЛПД и ДПД съответно (Фиг. 16.1).

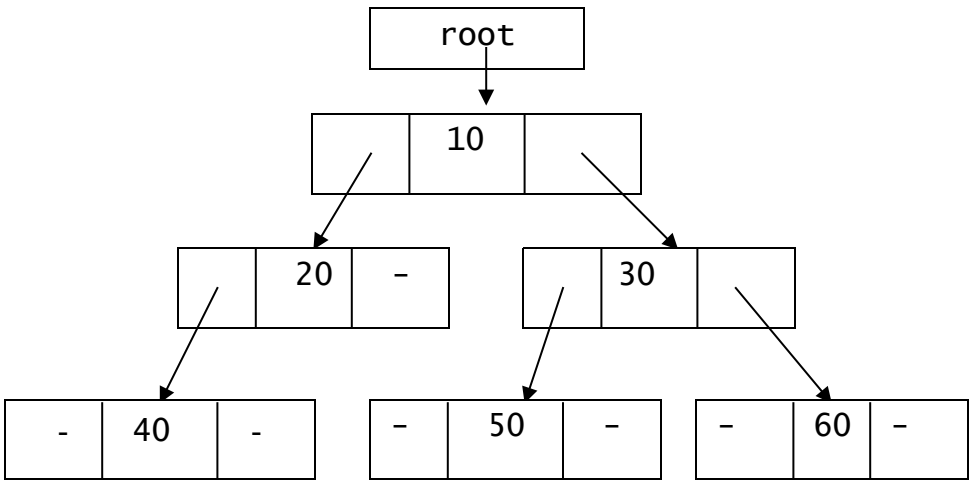


Фиг. 16.1 Свързано представяне на двоично дърво

Пример: Двоичното дърво



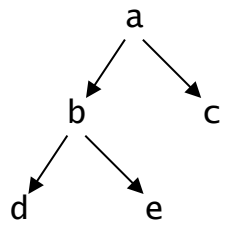
се представя по следния начин:



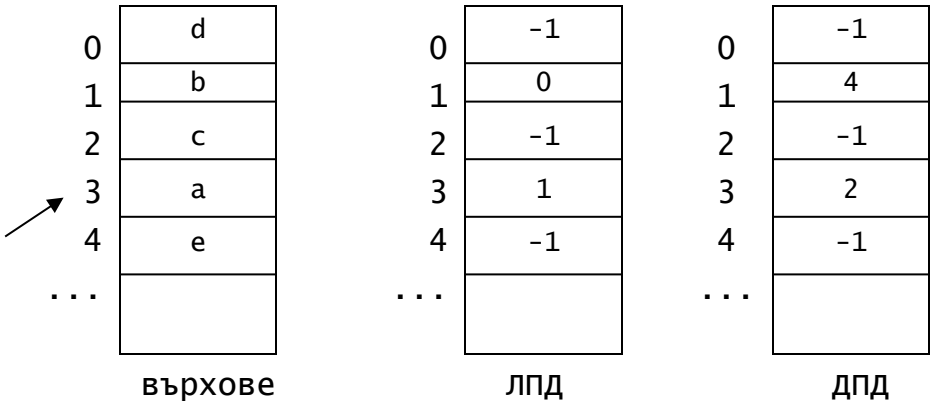
Верижно представяне

При това представяне се използват три масива - за върховете на дървото, за адресите на лявото и за адресите на дясното поддърво. Ролята на адреси се изпълнява от индекси. i-ят елемент на масива "върхове" съдържа стойността на връх на двоичното дърво, i-ят елемент на ЛПД съдържа адреса на лявото поддърво на поддървото с корен i-я елемент, а i-ят елемент на ДПД - адреса на дясното му поддърво. Поддържа се указател, който съдържа адреса на корена.

Пример: Верижното представяне на двоичното дърво от тип T



се представя по следния начин:



Указателят към корена е 3. Празното дърво е представено чрез –1.

В следващите разглеждания ще използваме свързаното представяне на двоично дърво.

16.1.2 Реализация на свързаното представяне

Тройната кутия, съдържаща корена и адресите на лявото и дясното поддървета, представяме чрез следния шаблон на структурата node.

```
template <class T>
struct node
{T inf;
 node *Left,
      *Right;
};
```

Двоичното дърво ще представим чрез указател към тройна кутия от вида, описан по-горе. Ще го реализираме чрез шаблона на класа tree

```
template <class T>
class tree
{public:
 tree();
 ~tree();
 tree(tree const&);
 tree& operator=(tree const&);
 bool empty() const;
 T RootTree() const;
 tree LeftTree() const;
 tree RightTree() const;
 void Create3(T, tree<T>, tree<T>);
 void print() const
 {pr(root);
  cout << endl;
 }
 void Create()
 {CreateTree(root);
 }
 private:
  node<T> *root;
  void DeleteTree(node<T>* &) const;
  void Copy(node<T> * &, node<T>* const&) const;
  void CopyTree(tree<T> const&);
  void pr(const node<T> *) const;
  void CreateTree(node<T> * &) const;
};
```

Конструкторът и функциите на голямата тройка реализират познати идеи.

```
template <class T>
tree<T>::tree()
{root = NULL;
}
template <class T>
tree<T>::~tree()
{DeleteTree(root);
}
template <class T>
tree<T>::tree(tree<T> const& r)
{CopyTree(r);
}
template <class T>
tree<T>& tree<T>::operator=(tree<T> const& r)
{if (this != &r)
{DeleteTree(root);
 CopyTree(r);
}
 return *this;
}
```

За реализирането им използваме член-функциите DeleteTree, CopyTree и Copy на шаблона на класа tree.

```
template <class T>
void tree<T>::DeleteTree(node<T>* &p)const
{if (p)
 {DeleteTree(p->Left);
  DeleteTree(p->Right);
  delete p;
  p = NULL;
 }
}
template <class T>
void tree<T>::CopyTree(tree<T> const& r)
{Copy(root, r.root);
```

```
}
template <class T>
void tree<T>::Copy(node<T> * & pos, node<T>* const & r) const
{pos = NULL;
 if (r)
 {pos = new node<T>;
  pos->inf = r->inf;
  Copy(pos->Left, r->Left);
  Copy(pos->Right, r->Right);
 }
}
```

Използването на допълнителния параметър от тип node<T>* в DeleteTree и Copy е заради рекурсията.

Проверката дали двоично дърво е празно се осъществява чрез булевата член-функция empty() на шаблона.

```
template <class T>
bool tree<T>::empty()const
{return root == NULL;
}
```

Достъпът до корена, до лявото и до дясното поддърво на дадено двоично дърво се осъществява чрез член-функциите: RootTree(), LeftTree() и RightTree().

```
template <class T>
T tree<T>::RootTree()const
{return root->inf;
}
template <class T>
tree<T> tree<T>::LeftTree() const
{tree<T> t;
 Copy(t.root, root->Left);
 return t;
}
template <class T>
tree<T> tree<T>::RightTree() const
{tree<T> t;
 Copy(t.root, root->Right);
```

```
 return t;
}
```

Извеждането на елементите на двоично дърво става чрез член-функцията print(). Тъй като реализацията й е рекурсивна, print() използва помощната член-функция pr.

```
template <class T>
void tree<T>::pr(const node<T>*p) const
{if (p)
 {pr(p->Left);
  cout << p->inf << " " ;
  pr(p->Right);
 }
}
```

Следните две член-функции създават двоично дърво. Функцията Create3 създава двоично дърво по дадени корен, ляво и дясно поддървета.

```
template <class T>
void tree<T>::Create3(T x, tree<T> l, tree<T> r)
{root = new node<T>;
 root->inf = x;
 Copy(root->Left, l.root);
 Copy(root->Right, r.root);
}
```

Член-функцията Create създава произволно двоично дърво. Тя използва капсолираната член-функция CreateTree, дефинирана рекурсивно по следния начин:

```
template <class T>
void tree<T>::CreateTree(node<T> * & pos) const
{T x; char c;
 cout << "root: ";
 cin >> x;
 pos = new node<T>;
 pos->inf = x;
 pos->Left = NULL;
 pos->Right = NULL;
 cout << "Left Tree of: " << x << "  y/n? ";
```

```
 cin >> c;
 if (c == 'y') CreateTree(pos->Left);
 cout << "Right Tree of: " << x << " y/n? ";
 cin >> c;
 if (c == 'y') CreateTree(pos->Right);
}
```

Записваме този шаблон във файла Tree.cpp и ще го демонстрираме чрез някои задачи.

Задачи върху двоично дърво

Задача 161. да се напише програма, която създава двоично дърво от цели числа. Намира и извежда корена, лявото и дясното му поддървета. Конструира и извежда двоично дърво с корен, съвпадащ с корена на първоначалното двоично дърво, с ляво поддърво – дясното поддърво на първоначалното и дясно поддърво – лявото поддърво на първоначалното двоично дърво.

Програма Zad161.cpp решава задачата.

```
// Program Zad161.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<int> IntTree;
void main()
{IntTree t;
 t.Create(); // създаване на двоично дърво
 t.print(); // извеждане на двоично дърво
 IntTree t1 = t.LeftTree(),   // намиране на ЛПД
         t2 = t.RightTree();  // намиране на ДПД
 int x = t.RootTree();  // намиране на корена
 cout << "Root: " << x << endl;
 cout << "LeftTree: \n";
 t1.print();
 cout << "RightTree: \n";
 t2.print();
```

```
 IntTree t3;
 t3.Create3(x, t2, t1); // генериране на новото дърво
 t3.print();
}
```

Задача 162. Да се напише функция, която увеличава всеки от върховете на двоично дърво от цели числа с дадено цяло число.

Програма Zad162.cpp решава задачата.

```
// Program Zad162.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<int> IntTree;
IntTree AddElem(int a, IntTree const& t)
{IntTree t1;
 if (!t.empty())
 t1.Create3(t.RootTree() + a,
            AddElem(a, t.LeftTree()),
            AddElem(a, t.RightTree()));
 return t1;
 }
void main()
{IntTree t;
 t.Create();
 t.print();
 cout << "Number: ";
 int a; cin >> a;
 AddElem(a, t).print();
}
```

Задача 163. Да се дефинира функция от по-висок ред map, прилагаща едноаргументната функция f към всеки от елементите на дадено двоично дърво.

В програма Zad163.cpp е дадена дефиницията на функцията map и използването ѝ за увеличаване с 1 на всеки от елементите на двоично

дърво от тип int, а също за прилагане на функцията “факториел” към всеки връх на дадено двоично дърво.

```
// Program Zad163.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<int> IntTree;
template <class T>
tree<T> map(T (*f)(T), tree<T> const &t)
{tree<T> t1;
 if(!t.empty())
 t1.Create3(f(t.RootTree()), map(f, t.LeftTree()),
                             map(f, t.RightTree()));
 return t1;
}
int f(int x)
{return x + 1;
}
int g(int x)
{if (x == 0) return 1;
 return x * g(x - 1);
}
void main()
{IntTree t;
 t.Create();
 t.print();
 map(f, t).print();
 map(g, t).print();
}
```

Задача 164. Да се дефинира функция от по-висок ред accumulate, прилагаща бинарната лявоасоциативна операция op над елементите на дадено двоично дърво в реда ЛКД. Да се използва accumulate за намиране на сумата и произведението на елементите на дадено двоично дърво от тип int.

```
// Program Zad164.cpp
```

```
#include <iostream.h>
#include "Tree.cpp"
typedef tree<int> IntTree;
template <class T>
T accumulate(T (*op)(T, T), T null_val, tree<T> const &t)
{if (!t.empty())
return op(op(accumulate(op,null_val,t.LeftTree()), t.RootTree()),
          accumulate(op, null_val, t.RightTree()));
return null_val;
}
int sum(int x, int y)
{return x + y;
}
int po(int x, int y)
{return x * y;
}
void main()
{IntTree t;
 t.Create();
 t.print();
 cout << accumulate(sum, 0, t) << endl;
 cout << accumulate(po, 1, t) << endl;
}
```

Задача 165. Да се дефинира булева функция equal, която установява дали двоичните дървета t1 и t2 от тип T, са равни.

```
template <class T>
bool equal(tree<T> const &t1, tree<T> const &t2)
{if (t1.empty() && t2.empty()) return true;
 if (t1.empty() && !t2.empty() ||
     !t1.empty() && t2.empty()) return false;
 if (t1.RootTree() != t2.RootTree()) return false;
 return equal(t1.LeftTree(), t2.LeftTree()) &&
        equal(t1.RightTree(), t2.RightTree());
}
```

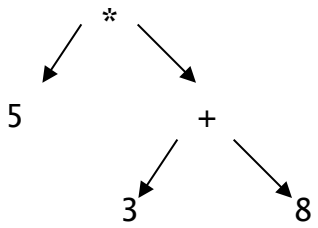
Задача 166. Да се напише функция depth, която намира дълбочината на двоично дърво от тип T, т.е. броя на върховете в най-дългия път от корена до листо.

```
template <class T>
int depth(tree<T> const&t)
{if (t.empty()) return 0;
 int n, m;
 n = depth(t.LeftTree());
 m = depth(t.RightTree());
 if (n > m) return n + 1;
 return m + 1;
}
```

Задача 167. Формулата

```
<формула> ::= <терминал>|
              (<формула><знак><формула>)
<знак> ::= +|-|*|/
<терминал> ::= 0|1| ...|9
```

може да се представи във вид на двоично дърво от тип char съгласно следното правило: формула състояща се от един терминал се представя чрез двоично дърво с един връх - цифрата; формула от вида (f1 s f2) се представя чрез двоично дърво, коренът на което е знака s, ЛПД съответства на формулата f1, ДПД - на формулата f2. Например, двоичното дърво



представя формулата (5 * (3 + 8))

Да се напише рекурсивна функция или процедура, която:

а) създава двоично дърво, представящо формула от горния вид;

б) проверява, явява ли се двоично дърво, дърво на формула;

в) намира стойността на формула, представена с двоично дърво;

г) извежда двоично дърво във вид, съответстващ на формула.

Програма Zad167.cpp решава задачата. Дървото създаваме чрез член-функцията Create на шаблона на класа tree.

```
// Program Zad167.cpp
#include <iostream.h>
#include "Tree.cpp"
typedef tree<char> CharTree;
// б)
bool IsForm(CharTree const& t)
{if (t.empty()) return false;
 char c = t.RootTree();
 if (c >= '0' && c <= '9')
    return t.LeftTree().empty() && t.RightTree().empty();
 if (c != '+' && c != '-' &&
     c != '*' && c != '/') return false;
 return IsForm(t.LeftTree()) && IsForm(t.RightTree());
}
// в)
int ArExpr(CharTree const&t)
{char c = t.RootTree();
 if (c >= '0' && c <= '9') return (int)c - (int)'0';
 switch(c)
 {case '+': return ArExpr(t.LeftTree()) + ArExpr(t.RightTree());
  case '-': return ArExpr(t.LeftTree()) - ArExpr(t.RightTree());
  case '*': return ArExpr(t.LeftTree()) * ArExpr(t.RightTree());
  case '/': return ArExpr(t.LeftTree()) / ArExpr(t.RightTree());
 }
 return 99999; // добавено е за коректност на кода
}
// г)
void print_tree(CharTree const&t)
{char c = t.RootTree();
 if (c >= '0' && c <= '9') cout << c;
 else
 {cout << '(';
  print_tree(t.LeftTree());
```

```
  cout << c;
  print_tree(t.RightTree());
  cout << ')';
 }
}
void main()
{CharTree t;
 t.Create();
 if (IsForm(t))
 {print_tree(t);
  cout << endl << ArExpr(t) << endl;
 }
 else cout << "Is not a formula.\n";
}
```

Задача 168. Нека в двоичното дърво от тип char е записана формула според синтаксиса от предишната задача, но в качество на терминали се използват не само цифри, а и букви, играещи ролята на променливи. Да се напише функция, която:

а) опростява двоично дърво, представящо формула като заменя в него всички поддървета, съответстващи на формулите (f+0), (0+f), (f-0), (f*1), (1*f) и (f/1) с поддърво, съответстващо на формулата f, а поддърветата, съответстващи на формулите (f*0), (0*f) и (0/f) - с върха 0.

б) преобразува двоично дърво, представящо формула като заменя в него всички поддървета, съответстващи на формулите ((f1+f2)*f3), ((f1-f2)*f3), (f1*(f2+f3)), (f1*(f2-f3)), ((f1+f2)/f3) и ((f1-f2)/f3) - с поддървета, съответстващи на формулите ((f1*f3)+(f2*f3)), ((f1*f3)-(f2*f3)), ((f1*f2)+(f1*f3)), ((f1*f2)-(f1*f3)), ((f1/f3)+(f2/f3)) и ((f1/f3)-(f2/f3)) съответно.

Програма Zad168.cpp решава условие а) на задачата. Условие б) оставяме за самостоятелна работа.

```
// Program Zad168.cpp
#include <iostream.h>
#include "Tree.cpp"
```

```
typedef tree<char> CharTree;
void Reduce(CharTree &t)
{char c = t.RootTree();
 if (c == '+' || c== '-' || c == '*' || c ==' /')
 {CharTree t1, t2;
  t1 = t.LeftTree();
  t2 = t.RightTree();
  Reduce(t1);
  Reduce(t2);
  t.Create3(c,t1,t2);  // много важно!
  if (c == '+' && t1.RootTree() == '0' ||
      c == '*' && t1.RootTree() == '1')
  t = t2;
  else
  if ((c == '*' || c == '/') && t2.RootTree() == '1' ||
      (c == '+' || c == '-') && t2.RootTree() == '0')
  t = t1;
  else
  if (c == '*' && (t1.RootTree() == '0' ||
                   t2.RootTree() == '0') ||
      c == '/' &&  t1.RootTree() == '0')
  {CharTree t3;
   t.Create3('0', t3, t3);
  }
 }
}
void main()
{CharTree t;
 t.Create();
 Reduce(t);
 t.print();
}
```

Задача 169. Да се напише шаблон на булева функция, който реализира проверка за принадлежност на елемент в двоично дърво.

```
template <class T>
bool member(T a, tree<T> const&t)
{if (t.empty()) return false;
 if (a == t.RootTree()) return true;
 return member(a, t.LeftTree()) || member(a, t.RightTree());
}
```

Това решение е неефективно, тъй като ако елементът не се съдържа в дървото се налага обхождане на дървото с пълно изчерпване. Операциите включване и изключване на връх не се реализират добре за обикновените дървета. По-удобно за редица цели е използването на т. нар. **двоично наредени дървета** или **двоични справочници**.

16.2 Двоично наредено дърво

Предполагаме, че за елементите от тип T е установена наредба.

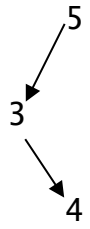
Дефиниция: Празното двоично дърво е двоично наредено дърво. Непразно двоично дърво, върховете на лявото поддърво на което са по-малки от корена, върховете на дясното поддърво са по-големи от корена и лявото, и дясното поддърво са двоично наредени дървета, се нарича **двоично наредено дърво от тип T**.

Нека t е двоично наредено дърво от тип T. *Включването на елемента a от тип T в t* се осъществява по следния начин:

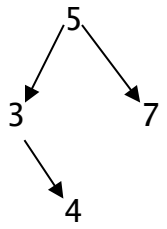
- ако t е празното двоично дърво, новото двоично наредено дърво е с корен елемента a и празни ляво и дясно поддървета.
- ако t не е празно и a е по-малко от корена му, елементът a се включва в лявото поддърво на t,
- ако t не е празно и a е не по-малко от корена му, елементът a се включва в дясното поддърво на t.

Пример: Ако в двоично нареденото дърво с корен 5 и празни поддървета се включи 3, ще се получи:

Ако към полученото двоично наредено дърво се включи 4, ще се получи:



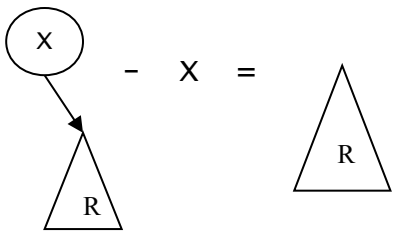
Ако към полученото двоично наредено дърво се включи 7, ще се получи:



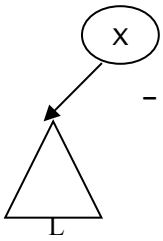
Включваният елемент "се спуска" по двоичното дърво, започвайки от корена и се насочва към лявото или дясното поддърво в зависимост от стойността си, докато намери празно място, което да заеме. Този начин на включване на елемент в двоично наредено дърво ще използваме за създаване на такива дървета.

Двоично наредената дърво притежава следното свойство: Обхождането на такова дърво по метода ЛКД сортира във възходящ ред елементите от върховете на дървото, а обхождането му по метода ДКЛ сортира в низходящ ред елементите от върховете на дървото.

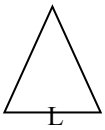
Изтриването на елемент от двоично наредено дърво се осъществява по следната схема:

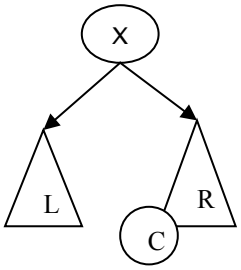


- X =

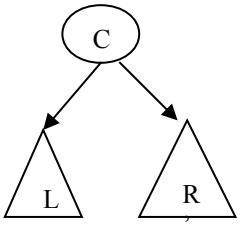


- X =

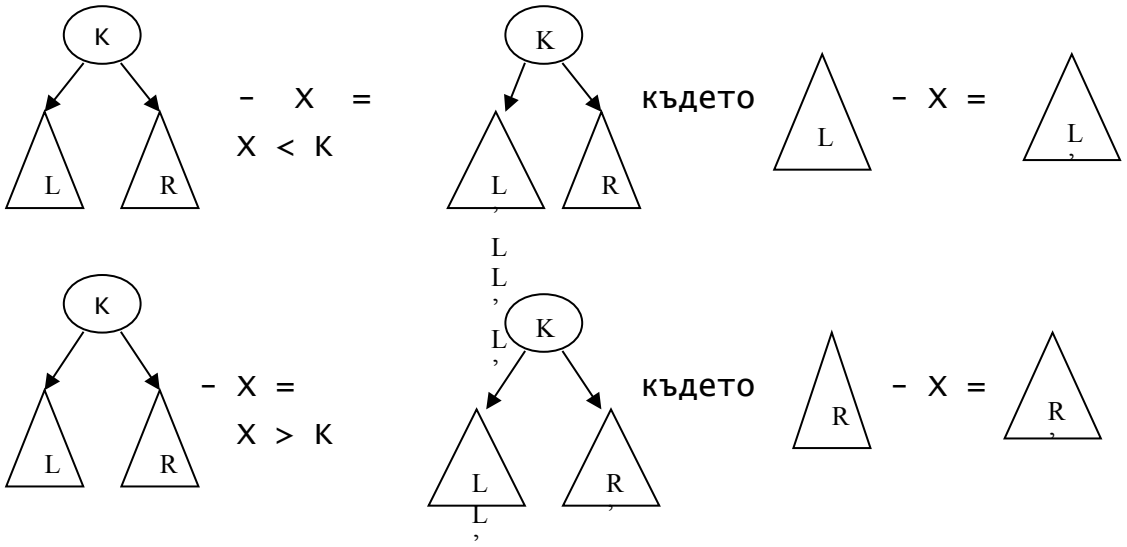




- X =



където C е минималмия елемент на R, а R' е R без минималмия му елемент.



Шаблонът на класа BinOrdTree реализира двоично наредено дърво от тип T.

```
template <class T>
struct node
{T inf;
 node *Left;
 node *Right;
};
template <class T>
class BinOrdTree
{public:
 BinOrdTree();
 ~BinOrdTree();
 BinOrdTree(BinOrdTree const&);
 BinOrdTree& operator=(BinOrdTree const&);
 bool empty()const;
 T RootTree() const;
 BinOrdTree LeftTree() const;
 BinOrdTree RightTree() const;
 void print() const
 {pr(root);
  cout << endl;
```

```
 }
 void AddNode(T const & x)
 {Add(root, x);
 }
 void DeleteNode(T const&);
 void Create3(T, BinOrdTree, BinOrdTree);
 void Create();
 void MinTree(T &, BinOrdTree &)const;
 private:
  node<T> *root;
  void DeleteTree(node<T>* &) const;
  void Copy(node<T> * &, node<T>* const&) const;
  void CopyTree(BinOrdTree const&);
  void pr(const node<T> *) const;
  void Add(node<T> *&, T const &) const;
 };
```

Голямата четворка е реализирана по същия начин като при обикновените двоични дървета.

```
template <class T>
BinOrdTree<T>::BinOrdTree()
{root = NULL;
}
template <class T>
BinOrdTree<T>::~BinOrdTree()
{DeleteTree(root);
}
template <class T>
BinOrdTree<T>::BinOrdTree(BinOrdTree<T> const& r)
{CopyTree(r);
}
template <class T>
BinOrdTree<T>& BinOrdTree<T>::operator=(BinOrdTree<T> const& r)
{if (this != &r)
 {DeleteTree(root);
  CopyTree(r);
 }
```

```
 return *this;
}
template <class T>
void BinOrdTree<T>::DeleteTree(node<T>* &p) const
{if (p)
 {DeleteTree(p->Left);
  DeleteTree(p->Right);
  delete p;
  p = NULL;
 }
}
template <class T>
void BinOrdTree<T>::CopyTree(BinOrdTree<T> const& r)
{Copy(root, r.root);
}
template <class T>
void BinOrdTree<T>::Copy(node<T> * &pos, node<T>* const &r) const
{pos = NULL;
 if (r)
 {pos = new node<T>;
  pos->inf = r->inf;
  Copy(pos->Left, r->Left);
  Copy(pos->Right, r->Right);
 }
}
```

Член-функциите empty, RootTree, LeftTree, RightTree, pr и print също са като тези на обикновените двоични дървета.

```
template <class T>
bool BinOrdTree<T>::empty()const
{return root == NULL;
}
template <class T>
T BinOrdTree<T>::RootTree()const
{return root->inf;
}
template <class T>
```

```
BinOrdTree<T> BinOrdTree<T>::LeftTree()const
{BinOrdTree<T> t;
 Copy(t.root, root->Left );
 return t;
}
template <class T>
BinOrdTree<T> BinOrdTree<T>::RightTree()const
{BinOrdTree<T> t;
 Copy(t.root, root->Right);
 return t;
}
template <class T>
void BinOrdTree<T>::pr(const node<T>*p) const
{if (p)
 {pr(p->Left);
  cout << p->inf << " ";
  pr(p->Right);
 }
}
```

Член-функцията Add е помощна. Използва се за реализиране на член-функцията AddNode на шаблона, чрез която се включва елемент в двоично наредено дърво по описания по-горе начин.

```
template <class T>
void BinOrdTree<T>::Add(node<T>* &p, T const & x)const
{if (!p)
 {p = new node<T>;
  p->inf = x;
  p->Left = NULL;
  p->Right = NULL;
 }
 else
 if (x < p->inf) Add(p->Left, x);
 else Add(p->Right, x);
}
```

Създаването на непразно двоично наредено дърво се осъществява чрез член-функцията Create. Тя реализира въвеждане на елементи и включването им чрез AddNode в двоично наредено дърво.

```
template <class T>
void BinOrdTree<T>::Create()
{root = NULL;
 T x; char c;
 do
 {cout << "> ";
  cin >> x;
  AddNode(x);
  cout << "next elem  y/n? "; cin >> c;
  } while (c == 'y');
}
```

Член-функцията Create3 е същата като тази при ненаредените двоични дървета.

```
template <class T>
void BinOrdTree<T>::Create3(T x, BinOrdTree<T> l, BinOrdTree<T> r)
{root = new node<T>;
 root->inf = x;
 Copy(root->Left, l.root);
 Copy(root->Right, r.root);
}
```

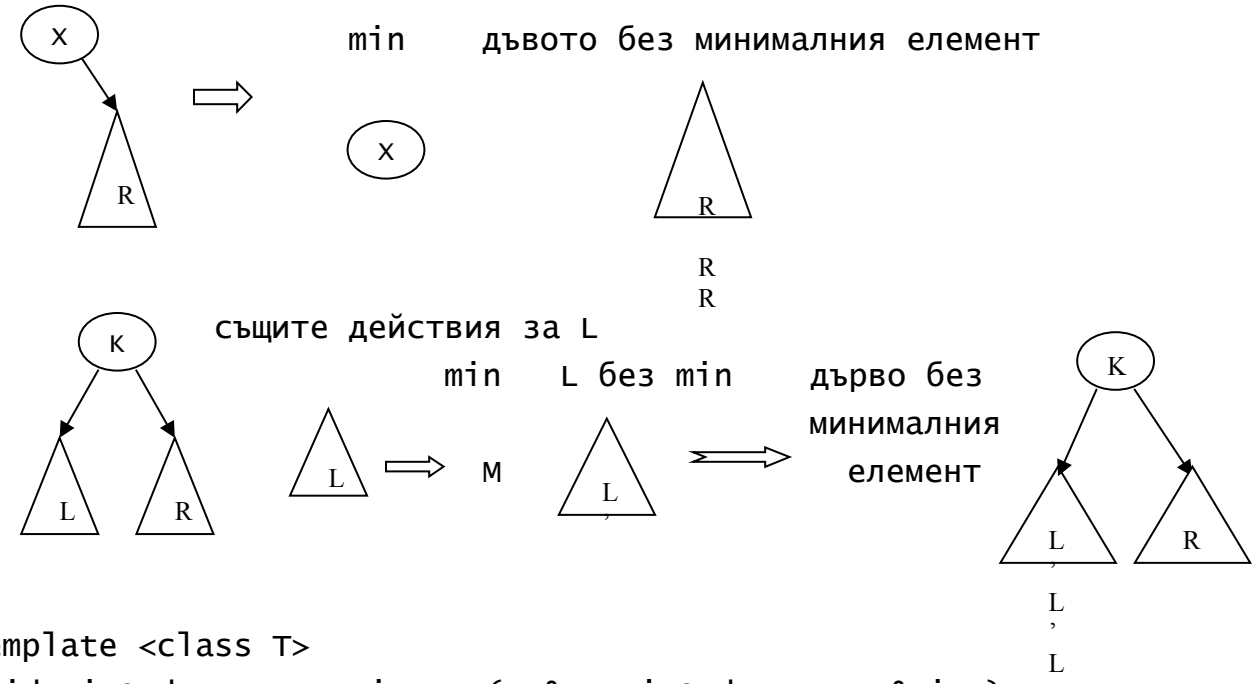
Член-функцията MinTree намира минималния елемент на подразбиращото се (соченото от указателя this) двоично наредено дърво и подразбиращото се двоично наредено дърво без минималния му елемент. Реализира следния алгоритъм.

- Ако лявото поддърво на подразбиращото се двоично наредено дърво е празно, минималният му елемент е корена, а дървото без минималния елемент е дясното му поддърво.

- Ако лявото му поддърво е непразно, на това двоично наредено дърво се намира минималния елемент и дървото без минималния елемент. След това се конструира двоично наредено дърво от корена, лявото

поддърво без минималния му елемент и дясното поддърво на подразбиращото се дърво.

Тези действия могат да се опишат графично така:




```
template <class T>
void BinOrdTree<T>::MinTree(T &x, BinOrdTree<T> &mint) const
{T a = RootTree();
 if (LeftTree().empty())
 {x = a;
  mint = RightTree();
 }
 else
 {BinOrdTree<T> t1;
  LeftTree().MinTree(x, t1);
  mint.Create3(a, t1, RightTree());
 }
}
```

Член-функцията DeleteNode изтрива връх на подразбиращото се двоично наредено дърво. Реализира описания по-горе алгоритъм. Изключването трябва да се предшества от проверка дали изключваният елемент принадлежи на двоично наредената дърво.

```
template <class T>
void BinOrdTree<T>::DeleteNode(T const& x)
```

```
{if (root)
 {T a = RootTree();
  BinOrdTree<T> t;
  if (a == x && LeftTree().empty()) *this = RightTree();
  else
  if (a == x && RightTree().empty()) *this = LeftTree();
  else
  if (a == x)
  {T c;
   RightTree().MinTree(c, t);
   Create3(c, LeftTree(), t);
  }
  else
  if (x < a)
  {t = *this;
   *this = LeftTree();
   DeleteNode(x);
   Create3(a, *this, t.RightTree());
  }
  else
  if (x > a)
  {t = *this;
   *this = RightTree();
   DeleteNode(x);
   Create3(a, t.LeftTree(), *this);
  }
 }
}
```

Записваме този шаблон във файла BinOrdTree.cpp и ще го експериментираме с няколко задачи.

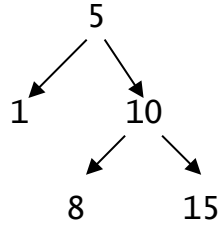
Задачи върху двоично наредено дърво

Задача 170. Да се напише програма, която въвежда редица от цели числа, сортира ги във възходящ ред, след което изключва указани върхове като запазва наредеността на елементите.

```
// Program Zad170.cpp
#include <iostream.h>
#include "BinOrdTree.cpp"
typedef BinOrdTree<int> IntTree;
void main()
{IntTree t;
 t.Create();
 t.print();
 int x;
 char c;
 do
 {cout << "x: ";
  cin >> x;
  t.DeleteNode(x);
  t.print();
  cout << "next: y/n: ";
   cin >> c;
 } while (c == 'y');
 t.print();
}
```

Експериментите с този и предходния шаблони извеждат в линеен вид двоичното дърво. За задача 170 този вид е подходящ, но има случаи, при които това не е така. Ще добавим към шаблона на класа BinOrdTree капсолираната процедура за извеждане pr1 и интерфейсната print_tree, които извеждат дървото, но завъртяно на 90 градуса по посока обратна на часовниковата стрелка.

Пример: Двоично-наредено дърво



ще бъде изведено по следния начин:

```
              15
           10
               8
         5
           1
```

```
template <class T>
void BinOrdTree<T>::pr1(const node<T>* p, int n) const
{if (p)
 {n = n + 5;
  pr1(p->Right, n);
  cout << setw(n) << p->inf << endl;
  pr1(p->Left, n);
 }
}
```

и

```
tempalate <class T>
void BinOrdTree<T>::print_tree()const
 {pr1(root, 0);
  cout << endl;
 }
```

Забележка: Заради използването на модификатора setw се налага включването на заглавния файл iomanip.h.

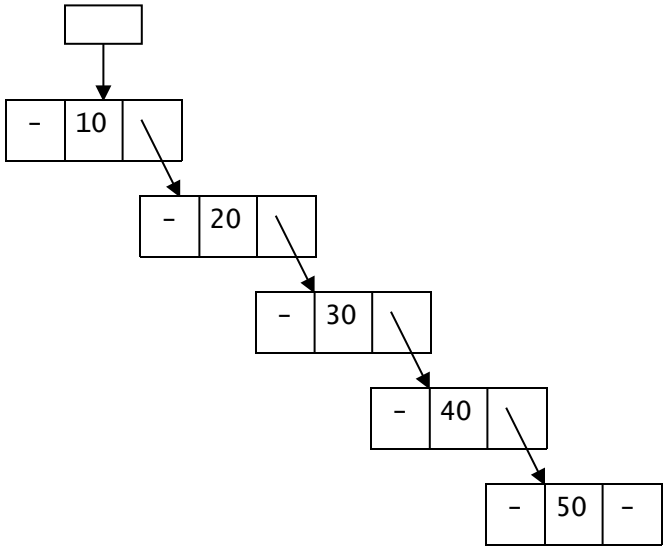
Задача 171. Да се напише шаблон на булева функция, чрез който се проверява дали елемент се съдържа в двоично наредено дърво.

```
template <class T>
bool member(T a, BinOrdTree<T> const& t)
{if (t.empty()) return false;
 if (a == t.RootTree()) return true;
 if (a < t.RootTree()) return member(a, t.LeftTree());
 else return member(a, t.RightTree());
}
```

16.3. Балансирани и идеално балансирани

двоично наредени дървета

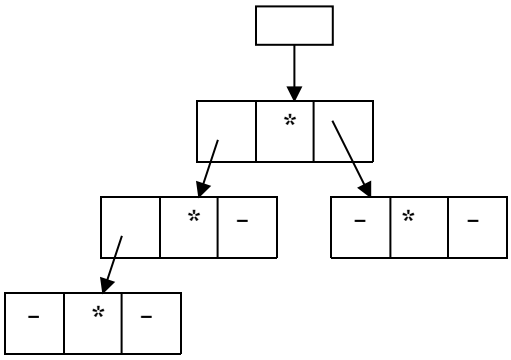
Ако елементите, които се включват в двоично наредено дърво са наредени в нарастващ ред, се получава двоично наредено дърво с линейна структура. Например, ако към празното двоично наредено дърво включим елементите: 10, 20, 30, 40, 50, се получава:



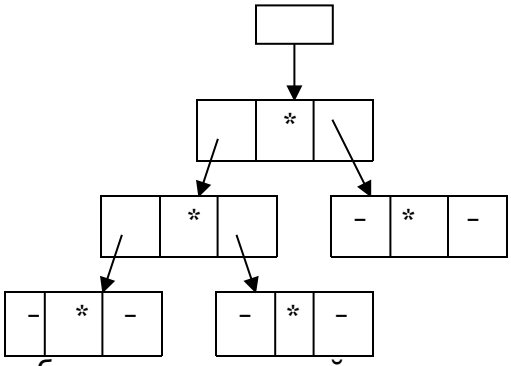
Търсенето на елемент в такова "изродено" дърво не е по-ефективно от това в свързаните списъци. Двоични дървета от този вид се наричат **силно небалансирани.**

Дефиниция. Двоично дърво се нарича **идеално (перфектно) балансирано** ако всеки негов връх има ляво и дясно поддърво, в които броят на възлите се различава най-много с 1.

Примери: Двоично дърво със следната структура



е идеално балансирано, но двоично дърво със структура:



не е идеално балансирано, тъй като лявото му поддърво има три върха, а дясното му – 1 (разликата е по-голяма от 1). Това двоично дърво е двоично дърво с балансирана височина.

Дефиниция. Двоично дърво се нарича **двоично дърво с балансирана височина** или по-просто **балансирано дърво**, ако за всеки негов връх височините (дълбочините) на лявото и дясното му поддървета се различават най-много с 1. Този вид двоични дървета се наричат също **AVL-дървета**.

От дефинициите и примера се вижда, че всяко идеално балансирано двоично дърво е дърво с балансирана височина, но обратното не е вярно.

Обикновено се изисква тези два вида двоични дървета да бъдат и наредени. Обаче алгоритъмът за включване на елемент в двоично наредено дърво, който използвахме за създаване на двоично наредено дърво, не създава идеално балансирано двоично наредено дърво, нито балансирано двоично наредено дърво. Операцията за изключване на елемент също не запазва идеалната балансираност или само балансираността на дървото.

Има ефективен алгоритъм за създаване на балансирани двоично наредени дървета, който при включване и изключване на елемент запазва балансираността на двоично наредоното дърво. За идеално балансираното двоично наредено дърво такъв не съществува.

Съществува прост алгоритъм за създаване на идеално балансирано двоично наредено дърво при следните ограничения:

- елементите, които ще се включват към празното двоично наредено дърво се подават в нарастващ ред;
- предварително е известен броят на върховете на дървото.

Към шаблона на класа BinOrdtree ще добавим и конструктор, който създава идеално балансирано двоично наредено дърво с n елемента.

```
template <class T>
BinOrdTree<T>::BinOrdTree(int n)
{if (n == 0) root = NULL;
 else
 {int nLeft = (n-1)/2,
      nRight = n – nLeft - 1;
  BinOrdTree<T> t1(nLeft);
  T x; cin >> x;
  BinOrdTree<T> t2(nRight);
  Create3(x, t1, t2);
 }
}
```

Като използва n, алгоритъмът намира теглата nLeft и nRight на поддърветата на дървото, където

n = nLeft + nRight + 1 и |nLeft-nRight| ≤ 1.

Създава тройната кутия и я запълва в последователността: конструиране на лявото поддърво с nLeft върха, въвеждане на корена и конструиране на дясното поддърво с nRight върха. Конструирането на лявото и дясното поддърво се осъществява чрез рекурсивно обръщение към конструктора на идеално балансирано двоично наредено дърво.

16.4 Граф

16.4.1 Дефиниране на граф

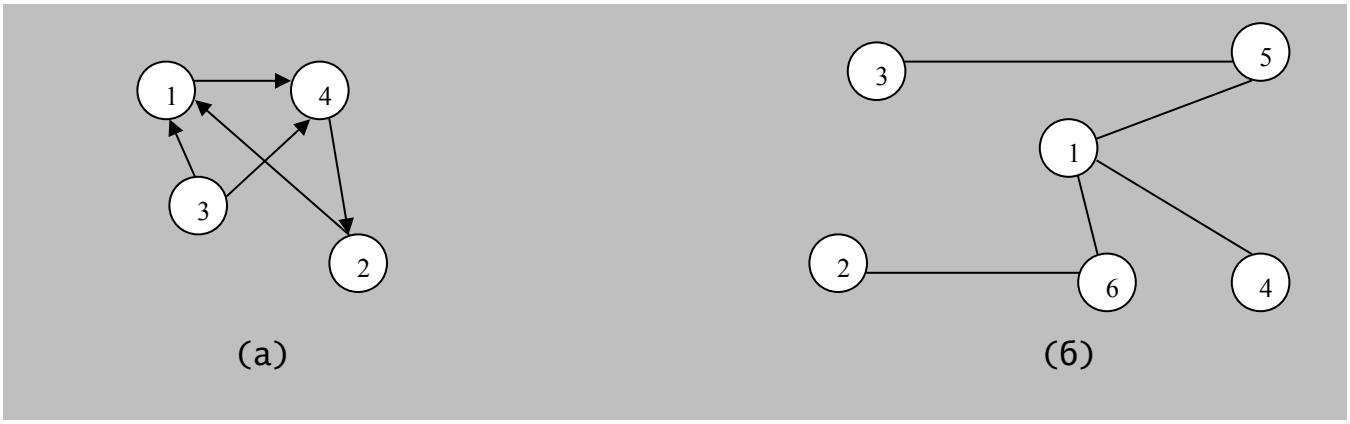
Структурата от данни граф ни е вече доста позната, тъй като свързаните списъци и двоичните дървета са нейни частни случаи. Освен това в Глави 10 и 13 на книгата Увод в програмирането на базата на езика C++, с която предполачм, че читателят е запознат, реализирахме последователното представяне на граф и предложихме алгоритми за търсене на път в граф.

Логическо описание

Графът се определя като множество от върхове заедно с множество от ребра. Всяко ребро се задава чрез двойка върхове. Ако ребрата са зададени като наредени двойки <i, j> от върхове, се наричат **дъги**. Такъв граф се нарича **ориентиран**. Ако ребрата на графа са зададени само чрез ненаредени двойки от върхове, графът се нарича **неориентиран**.

Ребрата (дъгите) могат да се свързват с етикети от произволен вид (имена, числа), в зависимост от конкретната задача. Тези етикети се наричат още **тегла**.

Примери: На Фиг. 16.2 (а) е даден ориентиран граф с цели числа във върховете, а на Фиг. 16.2 (б) – неориентиран граф.



Фиг. 16.2 Примери за ориентиран и неориентиран граф

Теорията на графите е интересен раздел на математиката. На базата на нея са реализирани редица полезни приложения. Ще разгледаме някои приложения на тази теория.

Ще използваме само ориентирани графи. Неориентираните графи ще представяме като ориентирани.

физическо представяне

Съществуват два основни начина за представяне на граф:

- последователно;
- свързано.

Последователното представяне се реализира чрез така наречената **матрица на съседство**. За граф с n върха, номерирани с 1, 2, ..., n, матрицата на съседство има n реда и n стълба. Ако <i, j> е дъга ориентирана от връх i до връх j в графа, елементът в i-тия ред и j-тия стълб на матрицата на съседство е равен на 1, в противен случай той е 0.

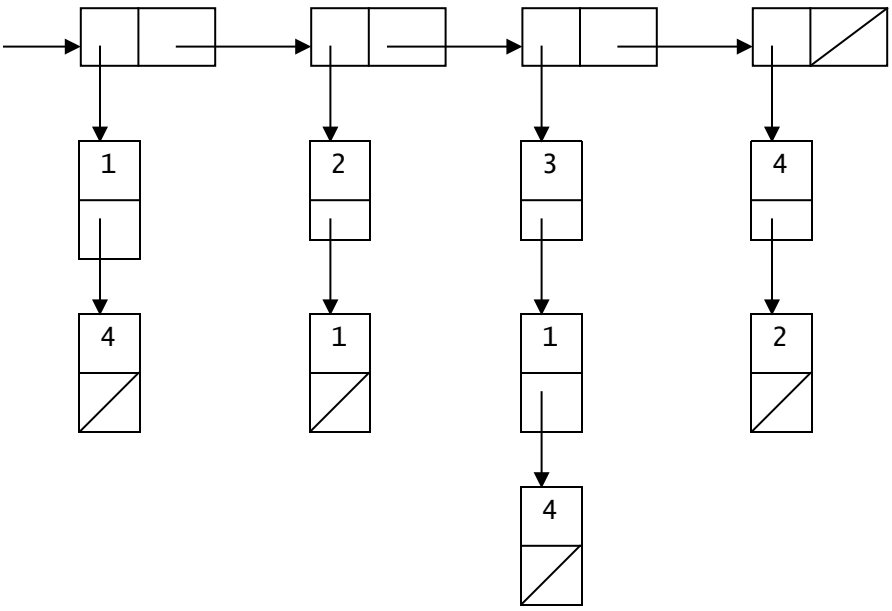
Пример: Матрицата на съседство за графа от Фиг 16.2 (а) има вида:

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Това представяне на практика не се използва често, тъй като при големи графи матриците на съседство заемат прекалено много памет и при много алгоритми водят до квадратично време за изпълнение. Матрици на съседство, с голям брой нулеви елементи, се наричат **разредни**.

Свързаното представяне се базира на свързаните списъци. Има различни реализации. Сравнително просто и удобно е да се използва свързан списък от толкова елемента, колкото са върховете на графа. Елементите на списъка са списъци като всеки подсписък да започва с връх i на графа и да съдържа всички върхове j, така че има дъга от i до j.

Пример: Това представяне за графа от Фиг. 16.2 (а) има вида:



Това представяне ще реализираме чрез шаблона на класа LList, дефиниращ свързан списък с една връзка и записан във файла L-List.cpp. Специализациите

typedef LList<int> IntList;

typedef LList<IntList> IntGraph;

определят класа IntGraph, задаващ граф с цели числа във върховете и реализиращ предложеното по-горе представяне.

Задача 172. Да се създаде непразен граф с цели числа във върховете, след което да се изведе.

За създаването на графа ще реализираме следния алгоритъм. Докато желаем, въвеждаме цяло число, означаващо връх на графа, след което го включваме като връх. След това отново докато желаем въвеждаме наредени двойки от върхове, означаващи дъги в графа и ги включване. За реализиране на тези действия ще дефинираме следните помощни процедури:

- void AddTop(int a, IntGraph &g), включваща върха a в графа g;
- void AddRib(int a, int b, IntGraph &g), включвща ребро от връх a до връх b на графа g.

За реализиране на включването на ребро, а също и за други цели, е полезна функцията

elem<int>* Point(int a, IntGraph &g)

която намира указател към двойната кутия, в информационната част на която е записан върхът a на графа g.

```
// Program Zad172.cpp
#include <iostream.h>
#include "L-List.cpp"
typedef LList<int> IntList;
typedef LList<IntList> IntGraph;
elem<int>* Point(int a, IntGraph &g)
{g.IterStart();
 elem<IntList>*p;
```

```
 elem<int> *q;
 do
 {p = g.Iter();
  p->inf.IterStart();
  q = p->inf.Iter();
 } while(q->inf != a);
 return q;
}
void AddTop(int a, IntGraph &g)
{IntList l;
 l.ToEnd(a);
 g.ToEnd(l);
}
void AddRib(int a, int b, IntGraph &g)
{elem<int> * q = Point(a, g), *p;
 while (q->link) q = q->link;
 p = new elem<int>;
 p->inf = b;
 p->link = NULL;
 q->link = p;
}
void create_graph(IntGraph &g)
{char c;
 do
 {cout << "top_of_graph: ";
  int x; cin >> x;
  AddTop(x, g);
  cout << "Top y/n? "; cin >> c;
 } while (c == 'y');
 cout << "Ribs:\n";
 do
 {cout << "start top: ";
  int x; cin >> x;
  cout << "end top: ";
  int y; cin >> y;
  AddRib(x, y, g);
```

```
     cout << "next: y/n? "; cin >> c;
    } while (c == 'y');
   }
   void LList<IntList>::print()
   {elem<IntList> *p = Start;
    while (p)
    {p->inf.print();
     p = p->link;
    }
    cout << endl;
   }
   void main()
   {IntGraph g;
    create_graph(g);
    g.print();
   }
```

Друга реализация на create_graph, която в някои случаи е по-удобна е:

```
   void create_graph(IntGraph &g)
   {cout << "Number of tops: ";
    int n; cin >> n;
    for (int i = 1; i <= n; i++)
    {cout << "top_of_graph: ";
     int x; cin >> x;
     AddTop(x, g);
     cout << "Number of Tops from " << x <<  " To? ";
     int k; cin >> k;
     for (int j = 1; j <= k; j++)
     {cout << "top: ";
      int y; cin >> y;
      AddRib(x, y, g);
     }
    }
   }
```

Задача 173. Да се напише булева функция, която установява дали съществува ацикличен път от един до друг връх на граф.

Обикновено се реализира следният алгоритъм. Съществува път от връх a до връх b, ако:

- a==b или
- съществува връх c, така че от a до c има ребро и има път от връх c до връх b.

Ако a не съвпада с b и е намерен връх c, така че от a до c има ребро, възможно е задачата за търсене дали има път от c до b да се сведе до търсене на път от a до b и да се стигне до зацикляне. За да се избегне зациклянето може да се използва помощен списък l, в който да се записват върховете, които са преминати в процеса на търсене на път. За следващ връх от пътя да се избира само такъв, който не се съдържа в списъка l. Отначало l ще бъде празния списък, а ако път съществува, в l ще се намери един път от връх a до връх b на графа g. Булевата функция way реализира търсене с връщане назад, разгледан подробно в глава 13.

```
bool way(int a, int b, IntGraph &g, IntList &l)
{l.ToEnd(a);
 if (a == b) return true;
 elem<int> * q = Point(a, g);
 q = q->link;
 while (q)
 {if (!member(q->inf, l) && way(q->inf, b, g, l)) return true;
  DeleteLast(l);  // връщане назад
  q = q->link;
 }
 return false;
}
```

Функцията way използва вече известната функция

```
bool member(int x, IntList l)
{l.IterStart();
 elem<int> *p = l.Iter();
 if (!p) return false;
 int y;
 l.DeleteElem(p, y);
```

```
 return x == y || member(x, l);
}
```

а също и процедурата DeleteLast, която изтрива последния елемент на списък.

```
void DeleteLast(IntList &l)
{l.IterStart(); int x;
 elem<int>*p = l.Iter();
 while (p->link) p = l.Iter();
 l.DeleteElem(p, x);
}
```

Задача 174. Да се напише булева функция, която намира всички ациклични пътища от един до друг връх на граф.

Процедурата

```
void allways(int a, int b, IntGraph &g, IntList &l)
```

намира всички пътища от връх a до връх b на графа g. Всеки път се конструира в списъка l, след което се извежда. Глобалната булева променлива flag получава стойност true ако съществува път от a до b в g. Инициализирана е с false. Процедурата allways реализира търсене с връщане назад.

```
bool flag = false;
void allways(int a, int b, IntGraph &g, IntList &l)
{l.ToEnd(a);
 if (a == b)
 {flag = true;
  l.print();
 }
 else
 {elem<int> * q = Point(a, g);
  q = q->link;
  while (q)
  {if (!member(q->inf, l))
   {allways(q->inf, b, g, l);
    DeleteLast(l); // връщане назад
```

```
      }
      q = q->link;
     }
    }
   }
```

Задачи

Задача 1. Да се дефинира процедура, която извежда отначало всички положителни, а след това всички отрицателни върхове на двоично дърво от тип int.

Задача 2. Дадено е двоично дърво от тип int. Да се напише функция, която определя има ли сред върховете на двоичното дърво поне два върха с равни стойности.

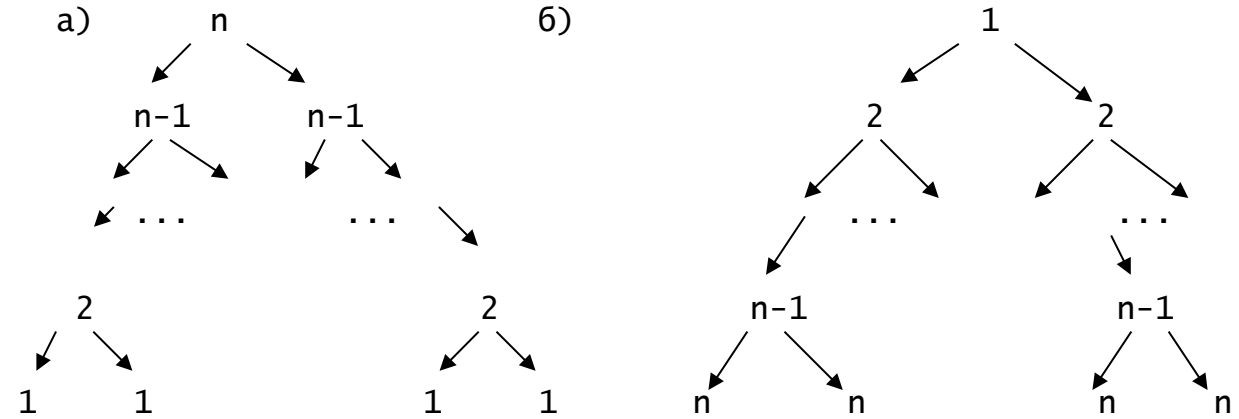
Задача 3. Дадено е двоично дърво от тип int. Да се напише функция, която намира сумата на четните елементите от върховете на двоичното дърво.

Задача 4. Да се напише функция или процедура, която:

а) определя броя на включванията на елемента a в двоичното дърво d;

б) намира броя на върховете на n-то ниво на непразното двоично дърво d (коренът се счита за връх от 0-во ниво).

Задача 5. Да се напише процедура CreateBinTree(d, n), където n е положително цяло число, която създава следното двоично дърво:



Задача 6. Да се напише програма, която за даден аритметичен израз от вида:

<АИ> ::= <терминал>|
(<АИ> <знак> <АИ>);
<знак> ::= +|-|*|/;
<терминал> ::= 0|1|...|9,

намира и извежда правия полски запис, който съответства на израза.

Задача 7. Да се напише програма, която за даден аритметичен израз от вида:

<АИ> ::= <терминал>|
(<АИ> <знак> <АИ>);
<знак> ::= +|-|*|/;
<терминал> ::= 0|1|...|9,

намира и извежда обратния полски запис, който съответства на израза.

Задача 8. Даден е свързан списък, съдържащ реални числа. Да се напише програма, която от елементите на списъка генерира двоично наредено дърво.

Задача 9. Даден е стек от реални числа. Да се напише програма, която сортира елементите на стека като за целта използва двоично наредено дърво.

Задача 10. Да се напише програма, която установява дали съществува път между два върха на дадено двоично дърво. Ако съществува път, да се намери пътя, а също и дължината му.

Допълнителна литература

1. B. Stroustrup, C++ Programming Language. Third Edition, Addison – Wesley, 1997.
2. А. Берстисс, Структуры данных, Москва, Статистика, 1974.
3. Ст. Липман, Езикът C++ в примери, “КОЛХИДА ТРЕЙД” КООП, София, 1993.
4. Л. Амерал, Алгоритми и структури от данни в C++, София, ИК СОФТЕХ, 2001.
5. М. Тодорова, Програмиране на Паскал, София, Полипринт, 1993.