Глава 1

Деревья

Дерево — иерархическая абстрактная структура данных. Используется в различных областях.

Формально,

Определение 1. Дерево — набор элементов, связанных отношениями «родитель – ребенок», удовлетворяющих следующим условиям:

1. Если дерево непустое, то существует вершина, называемая **корнем дерева**, не имеющая родителя.

2. Каждый узел v дерева имеет одного родителя w. Тогда v является ребенком w.

Дерево можно также определить рекурсивно:

Определение 2. Дерево — набор элементов, удовлетворяющих следующим условиям:

1. Если дерево непустое, то существует вершина, называемая **корнем дерева**, не имеющая родителя.

2. Каждый узел v можно трактовать как корень своего дерева. Такие деревья называют **поддеревом**.

Визуально, дерево представляется в виде узлов и ребер, соединяющих родителя и его детей. Не бывает изолированных узлов. Если два узла имеют одного родителя, то эти узлы не могут быть соединены ребром (тогда это будет граф.) В английских источниках узлы, имеющие одного родителя, называются **siblings**. В русском языке аналога этого слова нет (только в биологии встречается сибс.)

Родитель и ребенок — это пара смежных узлов. Если говорить о более дальних связях, то вводится понятие **пути**. Путь из узла A до узла B — это набор узлов $A, a_1, \ldots, a_n, B$ таких, что узел a_{i+1} является ребенком узла a_i. **Длина пути** — число таких узлов минус единица.

Если существует путь из узла A в узел B, тогда узел A является **предком** узла B, а узел B — **потомком** узла A. Корень дерева является предком всех остальных узлов.

Узлы разделяются на **внутренние** и **внешние**. Внешние узлы не имеют детей, очень часто называются также **листьями**.

Введем понятие **высоты узла** — максимальная длина пути от узла до листа. (Находим длины путей от узла до всех листьев и выбираем максимальную из них). Соответственно, **высота дерева** — максимальная длина пути от корня до листа.

Глубина узла — длина пути от корня до узла. Глубина узла находится однозначно, так как у каждого узла только один родитель.

Часто говорят, что узлы расположены по уровням. По умолчанию корень расположен на нулевом уровне. Дети корня — на первом, «внуки» — на втором и т. д.

Например, для дерева, изображенного на рисунке 1.1, корнем является узел A, листьями — узлы H, F, G, I. На втором уровне расположены узлы D, G, K. Длина пути от B до H равна 3 (путь: $B \to D \to E \to F \to H$.) Пути от B до H не существует, так как они расположены в разных поддеревьях. Глубина узла F равна 3 (путь от корня до F: $A \to B \to D \to F$.) Длины пути от корня до листьев: $L(A \to H) = 4$, $L(A \to G) = 2$, $L(A \to F) = 3$, $L(A \to I) = 3$. Следовательно, высота данного дерева равна 4.

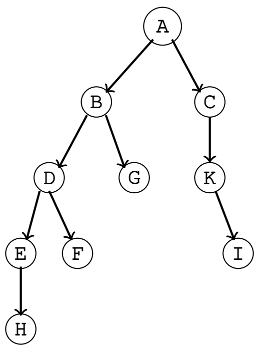


Рис. 1.1. Пример дерева

Деревья бывают **упорядоченными** и **неупорядоченными**. В случае упорядоченного дерева является важным порядок следования узлов на уровне, обычно следование идет слева направо. Это определяется решаемой задачей, т. е., в случае упорядоченного дерева, деревья, изображенные на рисунке 1.2а) и 1.2б) будут различными, в случае неупорядоченного — нет.

1.1. Бинарное дерево

В общем случае, родитель может иметь любое количество детей, но использование такой структуры затруднено. Значительно удобнее использовать **бинарное дерево**, т. е.,

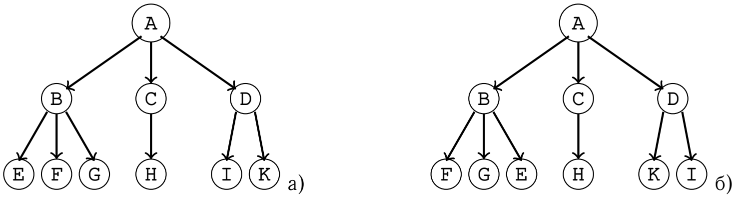


Рис. 1.2. Пример дерева (а) и (б)

дерево, каждый узел которого имеет не более двух детей. Будем считать, что бинарное дерево является упорядоченным. В дальнейшем будем рассматривать только бинарное дерево.

На уровне d бинарное дерево содержит максимум 2^d узлов (на нулевом — корень, на первом — два, на втором — 4 и т. д.) Следовательно, если дерево полностью заполнено и имеет высоту h, то оно содержит максимум $2^{h+1}-1$ элементов и минимум $h+1$ элемент (если предположить, что дерево содержит только одно из поддеревьев). В полностью заполненном бинарном дереве число листьев на единицу больше, чем число внутренних узлов. (Листья находятся на уровне h, следовательно их максимум 2^h. Внутренние узлы находятся на уровнях 0–$h-1$. Сумма внутренних узлов: $\sum\limits_{i=0}^{h-1} 2^i = 2^h - 1$.)

Таким образом, выполняются следующие условия: Если бинарное дерево содержит n узлов, n_e листьев, n_i внутренних узлов и высоту h, то

- $h+1 \leqslant n \leqslant 2^{h+1}-1$,
- $1 \leqslant n_e \leqslant 2^h$,
- $h \leqslant n_i \leqslant 2^h - 1$,
- $\log_2(n+1) - 1 \leqslant h \leqslant n-1$.

Из последнего свойства и определяется оценка времени работы алгоритмов сортировки, связанных с методом «разделяй и властвуй» (разделение задачи на более мелкие подзадачи). Т. е., в случае сортировки слиянием массив всегда делится на два подмассива, следовательно, высота минимальна и время работы $O(n \log n)$, а быстрая сортировка может привести к дереву, состоящему из одного поддерева, поэтому в худшем случае дерево имеет максимальную высоту и время работы $O(n^2)$.

Бинарное дерево легко представить как в виде массива (i—ый элемент массива является родителем для $2i+1$ и $2i+2$ элемента (в случае нулевой индексации)), так и в виде динамических структур данных.

Это связная структура, имеющая, как минимум, три поля (информационное, указатель на левого ребенка и указатель на правого ребенка). Иногда удобно добавить поле, хранящее информацию о родителе узла:

```
struct tree
{
    Item inf;
    tree *left;
    tree *right;
    tree *parent;
};
```

Для удобства дальнейшего рассмотрения напишем функцию `node(Item x)`, которая будет создавать новый узел, информационное поле будет равно `x`, остальные — `NULL`.

Листинг 1.1.

```
tree *node(Item x){
  tree *n = new tree;
  n->inf = x;
  n->parent = NULL;
  n->right = NULL;
  n->left = NULL;
}
```

1.2. Обходы

Построенное дерево необходимо вывести на экран. Для этого используются обходы деревьев.

Обход — способ вывода всех узлов дерева ровно один раз. Поскольку бинарное дерево состоит из левого поддерева (L), правого поддерева (R), корня (N). Понятно, что существует шесть вариантов обходов: LRV, LVR, VLR, RLV, RVL, VRL. Последние три являются симметричными первым трем, поэтому обычно рассматриваются три обхода:

- **Обратный**:
  1. Посетили левое поддерево;
  2. Посетили правое поддерево;
  3. Посетили корень.

- **Прямой**:

  1. Посетили корень;
  2. Посетили левое поддерево;
  3. Посетили правое поддерево.

- **Симметричный**:

  1. Посетили левое поддерево;
  2. Посетили корень;
  3. Посетили правое поддерево.

Для дерева, изображенного на рисунке 1.3 результаты обходов:

- Прямой: A, B, D, H, I, E, K, L, C, F, M, N, G, O, P.
- Обратный: H, I, D, K, L, E, B, M, N, F, O, P, G, C, A.
- Симметричный: H, D, I, B, K, E, L, A, M, F, N, C, O, G, P.

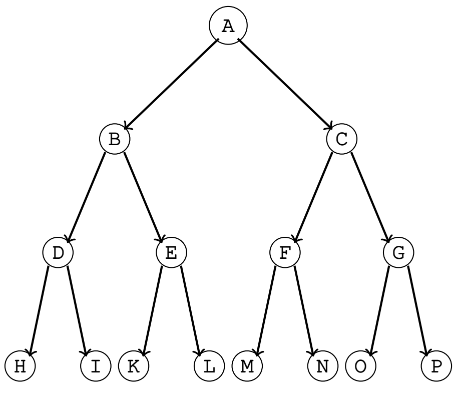


Рис. 1.3. Пример дерева

Если узел дерева описан как динамическая структура, то обходы описываются рекурсивно.

Листинг 1.2.

```
void preorder (tree *tr){ // прямой обход (К-Л-П)
```

```
  if (tr){
    cout << tr->inf;      //корень
    preorder(tr->left);   //левое
    preorder(tr->right);  //правое
  }
}

void postorder (tree *tr){ // обратный обход (Л-П-К)
  if (tr){
    postorder(tr->left);   //левое
    postorder(tr->right);  //правое
    cout << tr->inf;      //корень
  }
}

void inorder (tree *tr){ // симметричный обход (Л-К-П)
  if (tr){
    inorder(tr->left);   //левое
    cout << tr->inf;      //корень
    inorder(tr->right);  //правое
  }
}
```

1.3. Дерево математических выражений

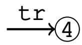
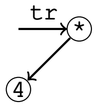
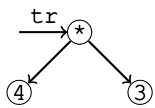
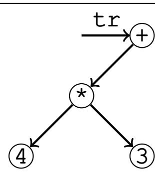
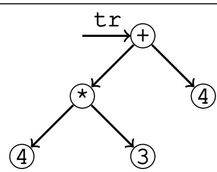
Часто математические выражения изображаются в виде дерева, где листья — это операнды, а внутренние узлы — знаки математических операций. Каждый внутренний узел будет обязательно иметь два ребенка.

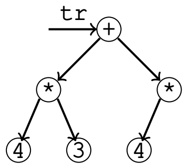
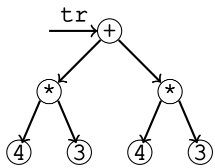
Корнем будет являться самая последняя операция. Вычисления производятся, начиная с листьев и идут вверх. Прямой, симметричный и обратный обходы приводят, соответственно, к префиксной (знак операции перед операндами (+34)), инфиксной (знак операции между операндами (3+4)) и постфиксной (знак операции после операндов (34+)) записи выражений.

Рассмотрим алгоритм построения подобного действия. Для простоты будем считать, что выражение без скобок и содержит только однозначные числа. Корректность выражения должна проверяться заранее, при разработке алгоритма считается, что выражение записано корректно.

Например, выражение: $4 * 2 + 4 * 3/2 + 8/4 + 5$.

Считываем последовательно символы и превращаем каждый из них в узел: `tree *n = node(str[i]);`

<table>
<tr><th></th><th></th><th></th></tr>
<tr><td>Считана цифра. Если дерево пустое, она становится корнем.</td><td>Листинг 1.3.<pre><code>if (!tr) tr = n;</code></pre></td><td>
</td></tr>
<tr><td>Считан знак операции (∗ или /). Если корень — это цифра, становится корнем, а цифра — левым ребенком.</td><td>Листинг 1.4.<pre><code>if (isdigit(tr->inf)){
  tr->parent = n;
  n->left = tr;
  tr = n;
}</code></pre></td><td>
</td></tr>
<tr><td>Считана цифра. Если у корня нет правого ребенка, становится правым ребенком.</td><td>Листинг 1.5.<pre><code>if (!tr->right){
  n->parent = tr;
  tr->right = n;
}</code></pre></td><td>
</td></tr>
<tr><td>Считан знак операции (+ или −). Становится корнем, а текущее дерево — левым поддеревом.</td><td>Листинг 1.6.<pre><code>if (str[i] == '-' || str[i] == '+'){
      tr->parent = n;
      n->left = tr;
      tr = n;
    }</code></pre></td><td>
</td></tr>
<tr><td>Считана цифра (см. строку 3).</td><td></td><td>
</td></tr>
</table>

<table>
<tr>
<td>Считан знак операции (/ или ∗). Становится правым ребенком для корня, а текущий правый ребенок корня становится левым ребенком для нового узла.</td>
<td>Листинг 1.7.
<pre><code>if (str[i] == '/' || str[i] == '*'){
  n->parent = tr;
  n->left = tr->right;
  tr->right->parent = n;
  tr->right = n;
}</code></pre></td>
<td></td>
</tr>
<tr>
<td>Считана цифра. Если корень имеет правого ребенка, ищем в правом поддереве узел, не имеющий правого ребенка. И новый узел становится правым ребенком для найденного узла.</td>
<td>Листинг 1.8.
<pre><code>if (isdigit(str[i])){
  tree *x = tr->right;
  while (x->right)
    x = x->right;
  n->parent = x->parent;
  x->right = n;
}</code></pre></td>
<td></td>
</tr>
</table>

Выше рассмотрены все варианты расположения узлов. Достроим дерево, для выражения, приведенного выше:

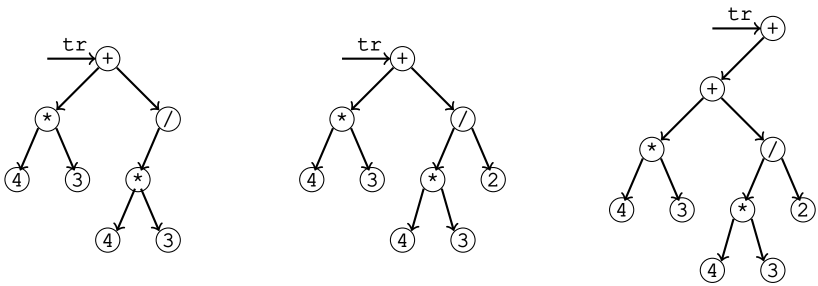


Рис. 1.4. Дерево, построенное для части выражения $4 * 2 + 4 * 3/2+$

Прямой обход полученного дерева (рисунок 1.7) даст префиксную (польскую) запись выражения (знак операции, операнд1, операнд2): $+ + + * 43/ * 432/845$.

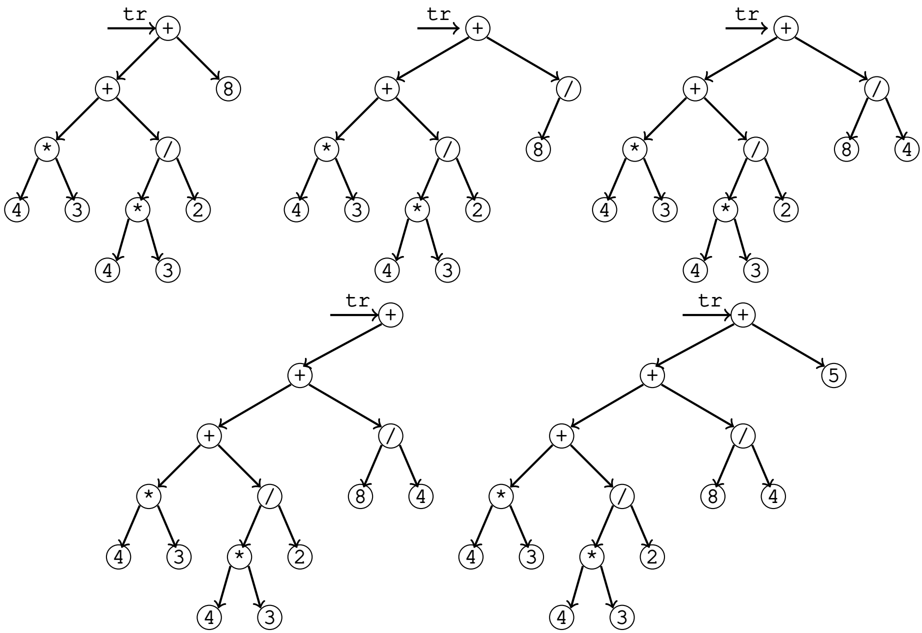


Рис. 1.5. Дерево, построенное для части выражения $4 * 3 + 4 * 3/2 + 8/4 + 5$

Симметричный обход дает инфиксную запись выражения (операнд1, знак операции, операнд2): $4 * 3 + 4 * 3/2 + 8/4 + 5$.

Обратный обход дает постфиксную (обратную польскую) запись выражения (операнд1, операнд2, знак операции): $43 * 43 * 2/ + 84/ + 5 +$.

Проще всего вычислить значение выражения, используя обратный обход. Записываем числа в стек. Если встретили знак операции, то извлекаем из стека два верхних элемента, вычисляем результат и записываем его в стек:

					3	*		2	/					4	/						
	3	*		4	4		12	12		6	+		8	8		2	+		5	+	
4	4		12	12	12	12	12	12	12	12		18	18	18	18	18		20	20		25

В листинге 1.9 представлен код программы, позволяющей построить дерево математических выражений и вычислить значение выражения.

Листинг 1.9.

```
#include <iostream>
#include <string>
```

```
#include <stack>
using namespace std;

struct tree{ //узел
  char inf;
  tree* right;
  tree *left;
  tree *parent;
};

tree *node(char x){ //создание узла
  tree *n = new tree;
  n->inf = x;
  n->left = n->right = NULL;
  n->parent = NULL;
  return n;
}

tree *create_tree(string str){ //создание дерева
  tree *tr = NULL;
  for (unsigned int i = 0; i < str.length(); i++){ //проход по строке
    tree *n = node(str[i]);
    if (str[i] == '-' || str[i] == '+'){//становится корнем
      tr->parent = n;
      n->left = tr; //имеющееся дерево становится левым
      tr = n;
    }
    else if (str[i] == '/' || str[i] == '*'){
        if (isdigit(tr->inf)){ //если первый знак операции в выражении - корень
          tr->parent = n;
          n->left = tr;
          tr = n;
        }
        else{ //добавляем справа от корня
          n->parent = tr;
          n->left = tr->right;//имеющийся элемент становится левым
          tr->right->parent = n;
          tr->right = n;
        }
        }
```

```
    else { //цифра
      if (!tr) tr = n; //если первая в выражении - становится корнем
      else{ //нет
        if (!tr->right){//у корня нет правого сына, становится им
           n->parent = tr;
           n->left = tr->right;
           tr->right = n;
        }
        else {//ищем операнд без правого сына
          tree *x = tr->right;
          while (x->right) x = x->right;
          n->parent = x->parent;
          x->right = n;
        }
      }
    }
  }
  return tr;
}

void postorder(tree *tr, stack<int> &a){//обратный обход
  if (tr){
    postorder(tr->left);
    postorder(tr->right);
    if (isdigit(tr->inf)){ //если узел -число, записываем в стек
      int n = tr->inf - '0';
      a.push(n);
    }
    else{//знак операции
      int b = a.top();//извлекаем 2 последних элемента стека
      a.pop();
      int c = a.top();
      a.pop();
      if(tr->inf = '+') a.push(b + c); //и записываем в стек
      if(tr->inf = '-') a.push(c - b); //результат в зав.
      if(tr->inf = '*') a.push(b * c); //от знака операции
      if(tr->inf = '/') a.push(c / b);
    }
  }
}
```

```
int main(){
  string str;
  getline(cin, str);
  string znak = "+-/*0123456789()";
  bool flag = true;
  for (unsigned int i = 0; i < str.length(); i++)//частичная проверка на корре
    if (znak.find_first_of(str[i]) == string::npos) {
      flag = false;
      break;
    }
  if (!flag) cout << "error";
  else {
    tree *tr = create_tree(str);//создали дерево
    stack<int> a;
    postorder(tr, a); //вызвали обход
    cout << a.top(); //в стеке один элемент - извлекаем его
    a.pop();
  }
  return 0;
}
```

1.4. Дерево бинарного поиска

Рассмотрим еще один случай бинарного дерева: дерево бинарного поиска. В этом случае добавляется дополнительное условие на узлы: для любого узла левый ребенок меньше своего родителя, правый — больше. В случае случайного распределения данных такое дерево подходит для поиска данных, так как необходимо пройти только по одной ветке дерева. Но, можно подобрать данные таким образом, что дерево будет представлять собой одну ветку, что увеличивает поиск до $O(n)$, где n — это количество элементов в дереве.

Для обхода дерева удобно использовать симметричный обход, так как в этом случае на экран будет выведена отсортированная последовательность.

Поскольку неравенства строгие, то дерево не содержит повторяющихся элементов, при вставке в дерево они просто игнорируются.

Простейшая реализация дерева бинарного поиска состоит в следующем:

1. Первый элемент всегда является корнем;

2. Если вставляемый элемент меньше корня, ищем подходящее место на левой ветке;

3. Если вставляемый элемент больше корня — на правой.

Например, построим дерево бинарного поиска для следующей последовательности: 5, 3, 7, 1, 9, 4, 2, 8, 6, 0

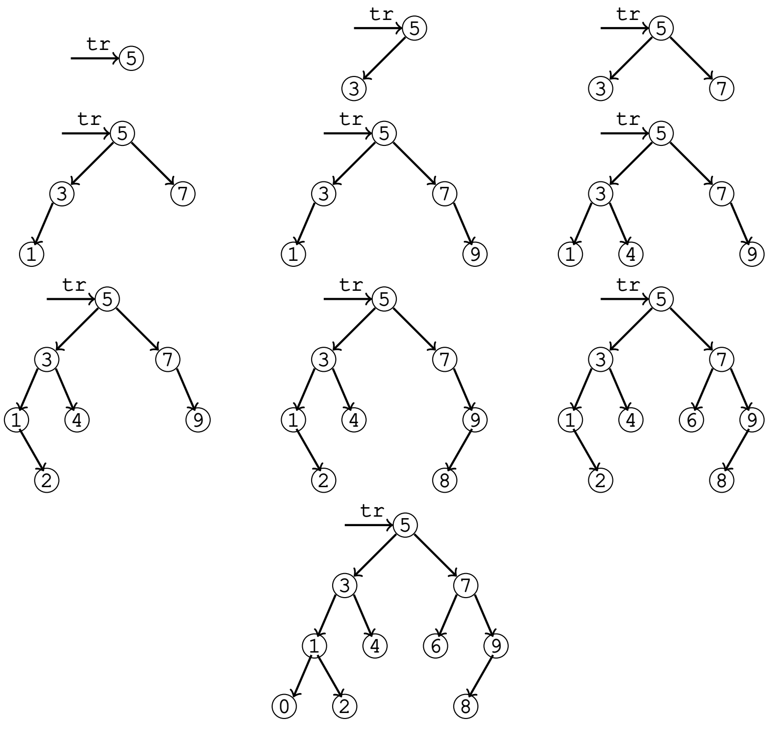


Рис. 1.6. Дерево бинарного поиска

Для дерева бинарного поиска элементарными функциями являются поиск элемента, нахождение минимального и максимального элементов, а также поиск предшествующего и следующего элеметнов (в смысле симметричного обхода).

Поиск элемента

- Если указатель равен `NULL`, значит элемента в дереве нет, возвращаем `NULL`;
- Если значение текущего элемента равно искомому значению, возвращаем указатель на этот элемент;

- Если значение текущего элемента больше искомого, рекурсивно вызываем поиск по левой ветке;
- Иначе рекурсивно вызываем поиск по правой ветке.

Поиск минимального элемента

- Если нет левого ребенка, элемент минимальный и возвращаем указатель на данный элемент.
- Иначе рекурсивно вызываем функцию по левой ветке.

Поиск максимального элемента

- Если нет правого ребенка, элемент максимальный и возвращаем указатель на данный элемент.
- Иначе рекурсивно вызываем функцию по правой ветке.

Поиск следующего элемента

- Если существует правый ребенок, то ищем минимальный по правой ветке.
- Иначе, идем вверх по дереву, до тех пока не дойдем до корня или пока текущий элемент остается правым ребенком. Возвращаем указатель на родителя.

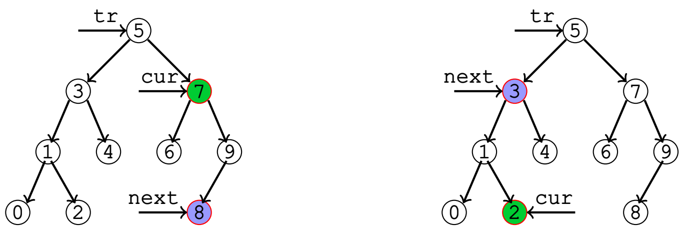


Рис. 1.7. Поиск следующего элемента при наличии правого ребенка (слева) и при отсутствии правого ребенка (справа)

Поиск предыдущего элемента

- Если существует левый ребенок, то ищем максимальный по левой ветке.
- Иначе, идем вверх по дереву, до тех пока не дойдем до корня или пока текущий элемент остается левым ребенком. Возвращаем указатель на родителя.

Удалить элемент из дерева надо таким образом, чтобы дерево по-прежнему оставалось деревом бинарного поиска.

Возможно три случая:

1. Удаление листа. Просто заменяем у родителя указатель на этот лист на `NULL.`

Рис. 1.8. Удаление листа

2. Удаление узла с одним ребенком. Для ребенка родителем становится «дед» (родитель удаляемого узла). Для «деда» ребенком становится «внук».

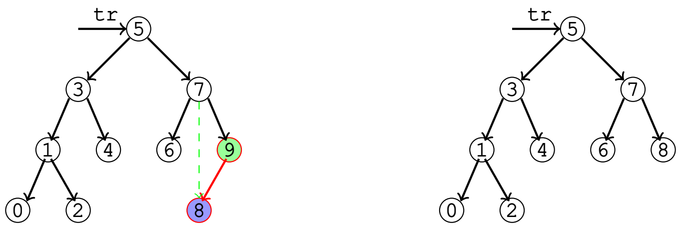


Рис. 1.9. Удаление узла с одним ребенком (узел 9)

3. Удаление узла с двумя детьми. Находим следующий за удаляемым узел. У него гарантировано не будет левого ребенка. Меняем значения удаляемого и найденного узлов. Если у найденного узла нет детей, выполняем пункт 1, если есть правый ребенок — выполняем пункт 2.

В листинге 1.11 приведена реализация описанных выше функций.

Листинг 1.10.

```
#include <iostream>
#include <string>
using namespace std;
```

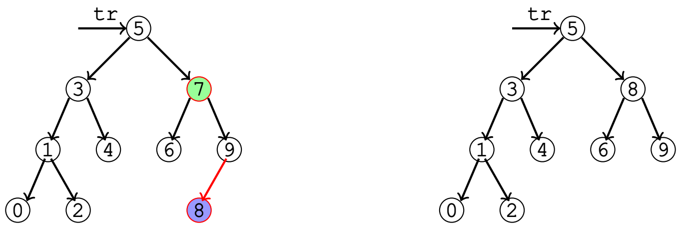


Рис. 1.10. Удаление узла с двумя детьми (узел 7)

```
struct tree{
  int inf;
  tree* right;
  tree *left;
  tree *parent;
};

tree *node(int x){//начальный узел
  tree *n = new tree;
  n->inf = x;
  n->left = n->right = NULL;
  n->parent = NULL;
  return n;
}

void insert(tree *&tr, int x){//вставка
  tree *n = node(x);
  if (!tr) tr = n; //если дерево пустое - корень
  else {
    tree *y = tr;
    while(y){ //ищем куда вставлять
      if (n->inf > y->inf) //правая ветка
        if (y->right)
          y = y->right;
        else{
          n->parent = y; //узел становится правым ребенком
          y->right = n;
          break;
        }
      else if (n->inf < y->inf)//левая ветка
```

```
        if (y->left)
          y = y->left;
        else{
          n->parent = y;//узел становится левым ребенком
          y->left = n;
          break;
        }
      }
    }
}

void inorder(tree *tr){//симметричный обход
  if(tr){
    inorder(tr->left);
    cout << tr->inf << " ";
    inorder(tr->right);
  }
}

tree *find(tree *tr, int x){//поиск
  if (!tr || x == tr->inf)//нашли или дошли до конца ветки
    return tr;
  if (x < tr->inf)
    return find(tr->left, x);//ищем по левой ветке
  else
    return find(tr->right, x);//ищем по правой ветке
}

tree *Min(tree *tr){//поиск min
  if (!tr->left) return tr;//нет левого ребенка
  else return Min(tr->left);//идем по левой ветке до конца
}

tree *Max(tree *tr){//поиск max
  if (!tr->right) return tr;//нет правого ребенка
  else return Max(tr->right);//идем по правой ветке до конца
}

tree *Next(tree*tr, int x){//поиск следующего
  tree* n = find(tr, x);
```

```
  if (n->right)//если есть правый ребенок
    return Min(n->right);//min по правой ветке
  tree *y = n->parent; //родитель
  while (y && n == y->right){//пока не дошли до корня или узел - правый ребенок
    n = y;//идем вверх по дереву
    y = y->parent;
  }
  return y;//возвращаем родителя
}

tree *Prev(tree *tr, int x){//поиск предыдущего
  tree *n = find(tr, x);
  if (n->left)//если есть левый ребенок
    return Max(n->left);//max по левой ветке
  tree *y = n->parent;//родитель
  while(y && n == y->left){//пока не дошли до корня или узел - левый ребенок
    n = y;//идем вверх по дереву
    y = y->parent;
  }
    return y;//возвращаем родителя
}


void Delete(tree *&tr, tree *v){//удаление узла
   tree *p = v->parent;
   if (!p) tr = NULL; //дерево содержит один узел
   else if (!v->left && !v->right){//если нет детей
     if (p->left == v) //указатель у родителя меняем на NULL
       p->left = NULL;
     if (p->right == v)
       p->right = NULL;
     delete v;
   }
   else if (!v->left || !v->right){//если только один ребенок
     if (!p) { //если удаляем корень, у которого 1 ребенок
       if (!v->left){ //если есть правый ребенок
         tr = v->right; //он становится корнем
         v->parent = NULL;
      }
      else { //аналогично для левого
```

```
       tr  = v->left;
       v->parent = NULL;
    }
  }
  else {
     if (!v->left){//если есть правый ребенок
       if (p->left == v) //если удаляемый узел явл. левым ребенком
         p->left = v->right; //ребенок удаляемого узла становится левым ребенком своего "деда"
       else
         p->right = v->right; ////ребенок удаляемого узла становится правым ребенком своего "деда"
     v->right->parent = p; //родителем ребенка становится его "дед"
    }
    else{//аналогично для левого ребенка
       if (p->left == v)
         p->left = v->left;
       else
         p->right = v->left;
       v->left->parent = p;
    }
   delete v;
  }
}
 else{//есть оба ребенка
   tree *succ = Next(tr, v->inf);//следующий за удаляемым узлом
   v->inf = succ->inf; //присваиваем значение
   if (succ->parent->left ==succ){//если succ левый ребенок
     succ->parent->left = succ->right; //его правый ребенок становится левым ребенком своего "деда"
     if (succ->right) //если этот ребенок существует
       succ->right->parent = succ->parent; //его родителем становится "дед"
   }
   else {//аналогично если succ - правsq ребенок
     succ->parent->right = succ->right;
     if (succ->right)
       succ->right->parent = succ->parent;
   }
   delete succ;
 }
```

```
}


int main(){
  int n, x;
  cout << "n="; cin >> n;
  tree *tr = NULL;
    for(int i = 0; i < n; i++){
    cout << i <<": ";
    cin >> x;
    insert(tr, x);
  }
  inorder(tr);
  cout << endl;
  cout << "min = " << Min(tr)->inf << endl;
  cout << "max = " << Max(tr)->inf << endl;
  cout << "x = "; cin >> x;
  if (find(tr,x)){
    cout << "next = " << Next(tr, x)->inf << endl;
    cout << "prev = " << Prev(tr, x)->inf << endl;
    Delete(tr, find(tr, x));
    inorder(tr);
    cout << endl;
  }
  else cout << "Such node not exist in this tree\n";
  return 0;
}
```

1.5. Идеально сбалансированное дерево

Дерево бинарного поиска обладает плохим свойством, что при определенном наборе данных (например, отсортированном) может обладать только одной веткой и, соответственно, его высота будет равна $N-1$.

Рассмотрим сначала *идеально сбалансированное дерево* — дерево, для каждого узла которого число потомков левого узла отличается от числа потомков правого узла не более чем на единицу. При этом пока уберем требование бинарного поиска (значения узлов не важно).

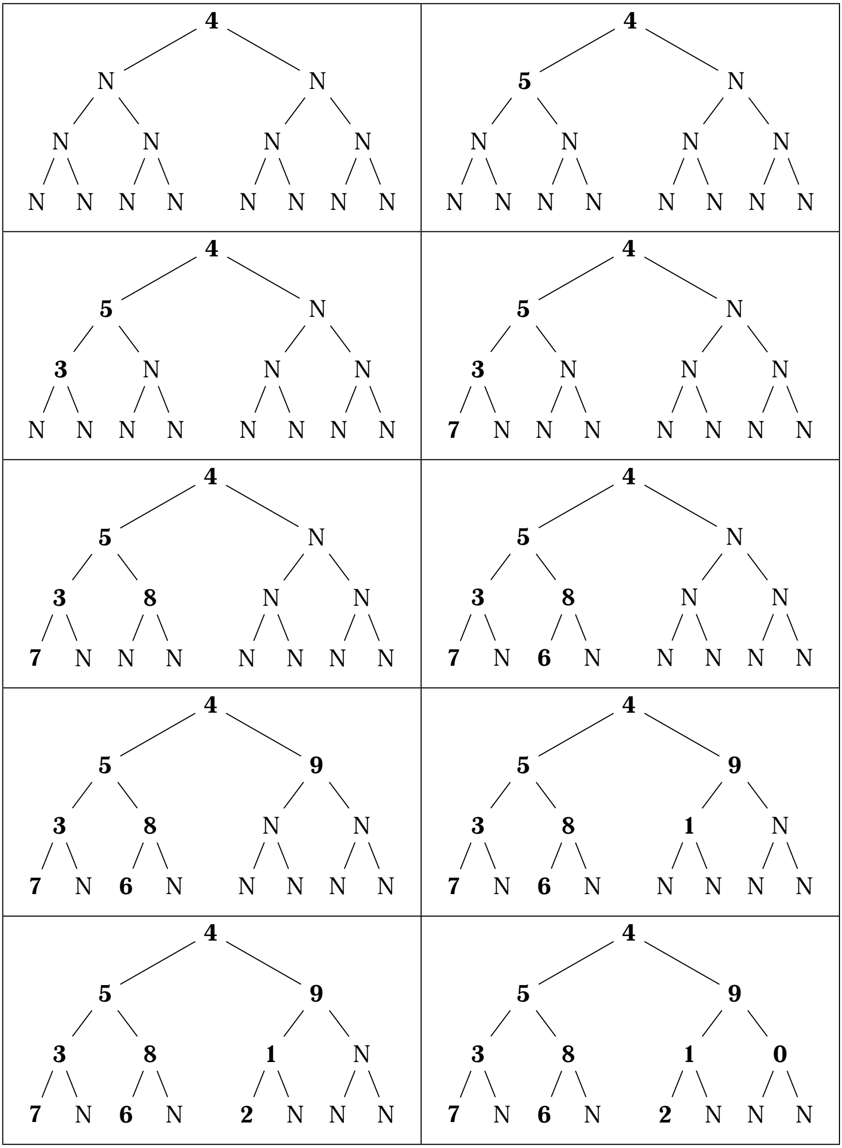
Для построения идеально сбалансированного дерева необходимо заранее знать общее количество элементов. Тогда дерево строится по следующему алгоритму:

- Пусть дано n элементов. Первый элемент списка является корнем.
- В левом поддереве будет $\frac{N}{2}$ элементов, в правом — $\frac{N}{2} - 1$ элемент (общее число элементов минус левое поддерево минус корень.)
- Сначала рекурсивно заполняем левую ветку, потом правую.

Такой способ построения позволяет гарантировать, что высота будет минимально возможной $\log_2 N$. Сначала заполняются левые ветки, потом правые.

Например, дано 10 элементов: $4\,5\,3\,7\,8\,6\,9\,1\,2\,0$.

- Корень — 4. Левое поддерево содержит 5 узлов: $5\,3\,7\,8\,6$. Правое — 4 узла: $9\,1\,2\,0$.
- Узел — 5. Левое поддерево содержит 2 узла: $3\,7$. Правое — 2 узла: $8\,6$.
- Узел — 3. Левое поддерево содержит 1 узел: 7. Правое — 0 узлов.
- Узел — 7. Это лист.
- Узел — 8. Левое поддерево содержит 1 узел: 6. Правое — 0 узлов.
- Узел — 6. Это лист. Левое поддерево для корня построено, переходим к правому.
- Узел — 9. Левое поддерево содержит 2 узла: $1\,2$. Правое — 1 узел: 0.
- Узел — 1. Левое поддерево содержит 1 узел: 2. Правое — 0 узлов.
- Узел — 2. Это лист.
- Узел — 0. Это лист. Дерево построено.



В идеально сбалансированном дереве, построенном по описанному выше алгоритму, можно определить одинаковое ли количество элементов в левом и правом поддеревьях, просто найдя путь от узла до крайнего левого листа и от узла до крайнего правого листа. Так как сначала заполняются левые ветки, потом правые, то эти пути будут одинаковы только в случае полностью заполненного дерева. Назовем их h_l и h_r.

Можно попробовать добавить один узел в идеально сбалансированное дерево. Самый простой вариант добавить узел либо в крайнюю правую ветку, если h_l не совпадает с h_r. Если у крайнего правого узла нет детей, новый узел становится левым ребенком этого узла, если левый ребенок есть — вставляемый узел становится правым ребенком. Если «высоты» совпадают, то дерево полностью заполнено и новый узел становится левым ребенком крайне левого листа. Но таким образом можно добавить только один узел. Если необходимо добавить несколько узлов, лучше просто перестроить дерево.

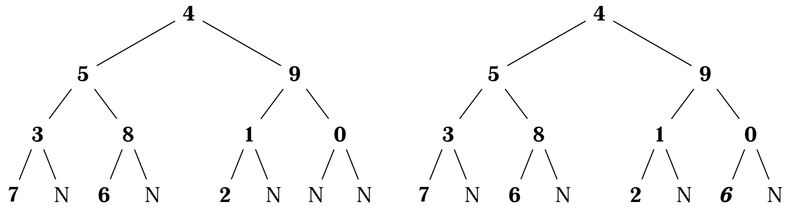


Рис. 1.11. Добавление узла ***6*** в дерево, у которого $h_l \neq h_r$.

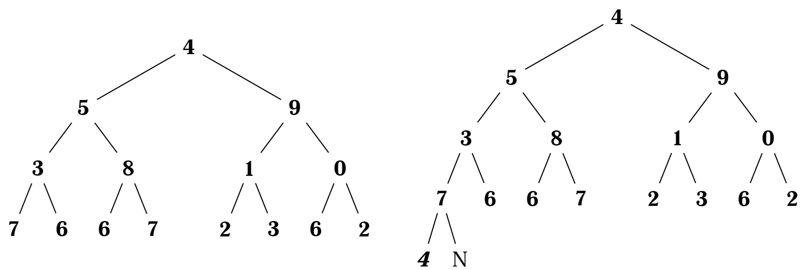


Рис. 1.12. Добавление узла ***4*** в дерево, у которого $h_l = h_r$.

Аналогично можно удалить один узел. Если «высоты» одинаковые — заменяем значение удаляемого листа и крайнего правого и удаляем крайний правый лист, если разные — заменяем значение крайнего левого листа и удаляемого узла и удаляем крайний левый лист. Если надо удалить несколько узлов, проще каждый раз перестраивать дерево.

Поиск элемента в дереве можно проводить с помощью любого обхода. Обходим дерево до тех пор, пока не встретим элемент или пока не закончатся узлы. В примере, приведенном ниже, поиск происходит с помощью прямого обхода.

Вывод элементов на экран тоже происходит с помощью прямого обхода, так как в этом случае, порядок следования элементов совпадает с порядком ввода элементов из файла или с клавиатуры.

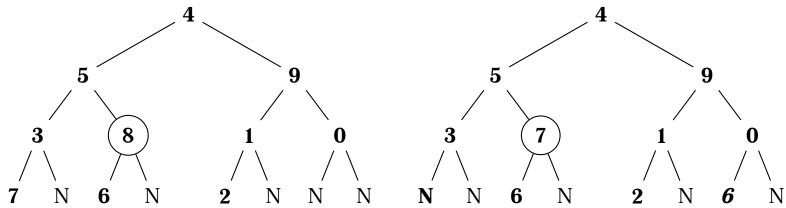


Рис. 1.13. Добавление узла ***6*** в дерево, у которого $h_l \neq h_r$.

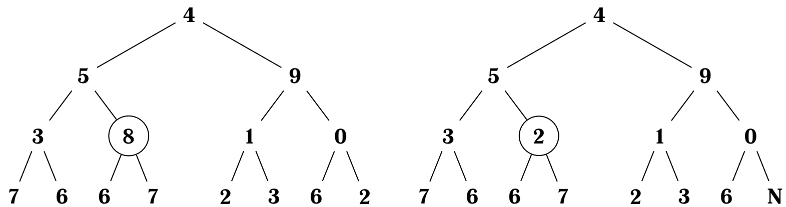


Рис. 1.14. Добавление узла ***8*** из дерева, у которого $h_l = h_r$.

Листинг 1.11.

```
#include <iostream>
#include <fstream>
#include <cmath>
#include <queue>
using namespace std;

ifstream in ("input.txt");

struct tree {
  int inf;
  tree *right;
  tree *left;
};

tree *node(int x){
  tree *n = new tree;
  n->inf = x;
  n->left = n->right = NULL;
  return n;
}

```

```
void create(tree *&tr, int n){
  int x;
  if (n > 0){
    in >> x;
    tr = node(x);
    int nl = n/2;
    int nr = n - nl - 1;
    create(tr->left, nl);
    create(tr->right,nr);
  }
}

void preorder(tree *tr){

  if (tr) {
    cout << tr->inf << "  ";
    preorder(tr->left);
    preorder(tr->right);
    }
}

int lefth(tree *tr){
  int k = 0;
  tree *x = tr;
  while (x){
    k++;
    x = x->left;
  }
    return k - 1;
}

int righth(tree *tr){
  int k = 0;
  tree *x = tr;
  while (x){
    k++;
    x = x->right;
  }
    return k - 1;
}
```

```
void add(tree *&tr, int x){
   tree *n = node(x);
   tree *y = tr;
   if (lefth(tr) == righth(tr)){
     do{
       y = y->left;
     }
     while (y->left);
     if (!y->left) y->left = n;
     else y->right = n;
   }
   else{
     do{
       y = y->right;
     }
     while (y->right);
     if (!y->left) y->left = n;
     else y->right = n;
   }

}

void find(tree *tr, int x, tree *&res){
  if (tr){
  if (tr->inf == x){
    res = tr;
    }
      else {
        find(tr->left, x, res);
          find(tr->right, x, res);
      }
  }
}

void del_n(tree *tr, int val){
  tree *y;
  find(tr, val, y);
  if (y){
    if(lefth(tr) == 0) tr = NULL;
```

```
    else if(lefth(tr) != righth(tr)){
      tree *x = tr->left;
      do{
        x = x->left;
      }
      while(x->left->left);
      if(x->right){
        if(x->right->inf == val){
          x->right = NULL;
        }
        else{
          y->inf = x->right->inf;
          x->right = NULL;
        }
        delete x->right;
      }
      else{
        if(x->left->inf == val){
          x->left = NULL;
        }
        else{
          y->inf = x->left->inf;
          x->left = NULL;
        }
        delete x->left;
      }
    }
    else{
      tree *x = tr->right;
      do{
        x = x->right;
      }
      while(x->right->right);
      if(x->right){
        if(x->right->inf == val){
          x->right = NULL;
        }
        else{
          y->inf = x->right->inf;
          x->right = NULL;
```

```
          }
          delete x->right;
        }
        else{
          if(x->left->inf == val){
            x->left = NULL;
          }
          else{
            y->inf = x->left->inf;
            x->left = NULL;
          }
          delete x->left;
        }
      }
      }

}

void print(tree *tr, int k){
  if (!tr)  cout << "Empty tree\n";
  else{
    queue<tree*> cur, next;
    tree *r = tr;
    cur.push(r);
    int j = 0;
    while (cur.size()){
      if (j == 0) {
        for (int i = 0; i < (int)pow(2.0, k) - 1; i++)
            cout << ' ';
      }
      tree *buf = cur.front();
      cur.pop();
      j++;
      if (buf){
        cout << buf->inf;
        next.push(buf->left);
        next.push(buf->right);
        for (int i = 0; i < (int)pow(2.0, k + 1) - 1; i++)
           cout << ' ';
      }
```

```
      if (!buf){
        for (int i = 0; i < (int)pow(2.0, k + 1) - 1; i++)
            cout << ' ';
        cout << ' ';
      }
      if(cur.empty()){
        cout << endl;
        swap(cur, next);
        j = 0;
        k--;
      }
    }
  }

}
int main(){
  tree *tr = NULL;
  int n, x;
  in >> n;
  create(tr, n);
  int k = int (log((float)n)/log((float)2.0));
  print(tr, k);
  preorder(tr);
  cout << endl;
  cout << lefth(tr) << "  " << righth(tr);
  cout << endl;
  cout << "x=";
  cin >> x;
  add(tr,x);
  n++;
  k = int (log((float)n)/log((float)2.0));
    print(tr, k);
  preorder(tr);
  cout << endl;
  cout << " del node: ";
  cin >> x;
  del_n(tr, x);
    n--;
  k = int (log((float)n)/log((float)2.0));
  print(tr, k);
```

```
  preorder(tr);
  cout << endl;
  return 0;
}
```