## Двоичное дерево поиска

**Двоичное дерево поиска** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *binary search tree*, BST) — это [двоичное дерево](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE), для которого выполняются следующие дополнительные условия (*свойства дерева поиска*):

* Оба поддерева — левое и правое — являются двоичными деревьями поиска.
* У всех узлов *левого* поддерева произвольного узла X значения ключей данных *меньше либо равны*, нежели значение ключа данных самого узла X.
* У всех узлов *правого* поддерева произвольного узла X значения ключей данных *больше либо равны*, нежели значение ключа данных самого узла X.

Данные в каждом узле должны обладать ключами, на которых определена операция сравнения *меньше*.

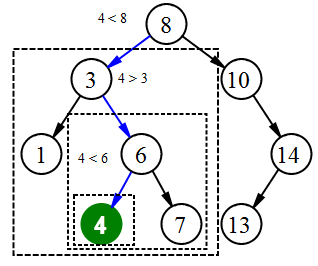
Как правило, информация, представляющая каждый узел, является записью, а не единственным полем данных. Однако это касается реализации, а не природы двоичного дерева поиска.

Для целей реализации двоичное дерево поиска можно определить так:

* Двоичное дерево состоит из узлов (вершин) — записей вида (data, left, right), где data — некоторые данные, привязанные к узлу, left и right — ссылки на узлы, являющиеся детьми данного узла — левый и правый сыновья соответственно. Для оптимизации алгоритмов конкретные реализации предполагают также определения поля parent в каждом узле (кроме корневого) — ссылки на родительский элемент.
* Данные (data) обладают ключом (key), на котором определена операция сравнения «меньше». В конкретных реализациях это может быть пара (key, value) — (ключ и значение), или ссылка на такую пару, или простое определение операции сравнения на необходимой структуре данных или ссылке на неё.
* Для любого узла X выполняются свойства дерева поиска: key[left[X]] < key[X] ≤ key[right[X]], то есть ключи данных родительского узла больше ключей данных левого сына и нестрого меньше ключей данных правого.

Двоичное дерево поиска не следует путать с [двоичной кучей](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D0%BA%D1%83%D1%87%D0%B0), построенной по другим правилам.

Основным преимуществом двоичного дерева поиска перед другими [структурами данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) является возможная высокая эффективность реализации основанных на нём алгоритмов [поиска](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) и [сортировки](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8).



int search = 0;

struct tree { // Структура дерева

int i; // Значение

tree\* left; // Указатель на левого ребенка

tree\* right; // Указатель на правого ребенка

tree\* parent; // Указатель на родителя

};

tree\* node(int x) { // Структура листа

tree\* X = new tree; // Создаем новый элемент

X->i = x; // Присваиваем значение

X->left = NULL; // Обнуляем левого и правого ребенка, а так же родителя

X->right = NULL;

X->parent = NULL;

return X;

}

void P\_obhod(tree\* X) { // Прямой

if (X) {

cout << X->i << " ";

P\_obhod(X->left);

P\_obhod(X->right);

}

}

void O\_obhod(tree\* X) { // Обратный

if (X) {

O\_obhod(X->left);

O\_obhod(X->right);

cout << X->i << " ";

}

}

void S\_obhod(tree\* X) { // Симметричный

if (X) {

S\_obhod(X->left);

cout << X->i << " ";

S\_obhod(X->right);

}

}

void Push(tree\*& X, int x) {

tree \*c = node(x); // Создаем элемент дерева

if (!X) // Если дерево не существует

X = c; // Делаем корень

else {

tree\* a = X; // Создаем копию для обхода

while (a) { // Пока не дошли до конца

if (a->i > x) {

if (a->left)

a = a->left;

else {

a->left = c;

c->parent = a;

break;

}

}

else if (a->right)

a = a->right;

else {

a->right = c;

c->parent = a;

break;

}

}

}

}

void Search(tree\* X, int i) {

if (X) {

if (X->i == i)

search++;

else {

Search(X->left, i);

Search(X->right, i);

}

}

}

void AddDel(tree\* A, tree\* X) { // Функция для восстановления ветки, родитель которой был удален

if (A) { // Если доп. ветка существет

Push(X, A->i); // Функция добавления

if (A->left) // Если существует левый ребенок, смещаемся к нему

AddDel(A->left, X);

if (A->right) // Если существует правый ребенок, смещаемся к нему

AddDel(A->right, X);

}

}

void Del(tree\* X, int i) { // Удаление элемента

if (!X) // Если элемент дерева не существует

search -= 0; // Заглушка

else if (X->i == i) {

if (!X->left && !X->right && X->parent) { // Нет левого и нет правого и есть родитель

if (X->parent->i > i)

X->parent->left = NULL;

else

X->parent->right = NULL;

search++; // Вспомогательная переменная для идентификации удаления

delete X;

}

else if (X->left && !X->right && X->parent) { // Если есть только левый и есть родитель

X->parent->left = X->left;

search++;

delete X;

}

else if (!X->left && X->right && X->parent) { // Если есть только правый и есть родитель

X->parent->right = X->right;

search++;

delete X;

}

else if (X->left && X->right && X->parent) { // Если есть и левый, и правый и есть родитель

tree\* A = X->left;

if (X->parent->i > i) // Если значение родителя больше искомого элемента

X->parent->left = X->right; // У родителя в левую ветку вставляем правую ветку текущего элемента

else

X->parent->right = X->right;

X = X->parent;

AddDel(A, X);

}

else if (X->left && X->right && !X->parent) { // Если есть левый и правый, но нет родителя / (Корень)

tree\* L = X->left; // Создаем дубликат левой ветки

tree\* R = X->right; // И правой

X->i = R->i; // Копируем информацию правой ветки как основную

X->left = R->left;

X->right = R->right;

search++;

AddDel(L, X); // И на нее цепляем значение

}

else if (!X->left && X->right && !X->parent) { // Если нет левого и родителя, но есть правый / (Корень)

tree\* R = X->right;

X->i = R->i;

X->left = NULL;

X->right = R->right;

search++;

}

else if (X->left && !X->right && !X->parent) { // Если нет правого и родителя, но есть левый / (Корень)

tree\* L = X->left;

X->i = L->i;

X->left = L->left;

X->right = NULL;

search++;

}

}

else { // Иначе сдвигаемся и делаем проверку заново

Del(X->left, i);

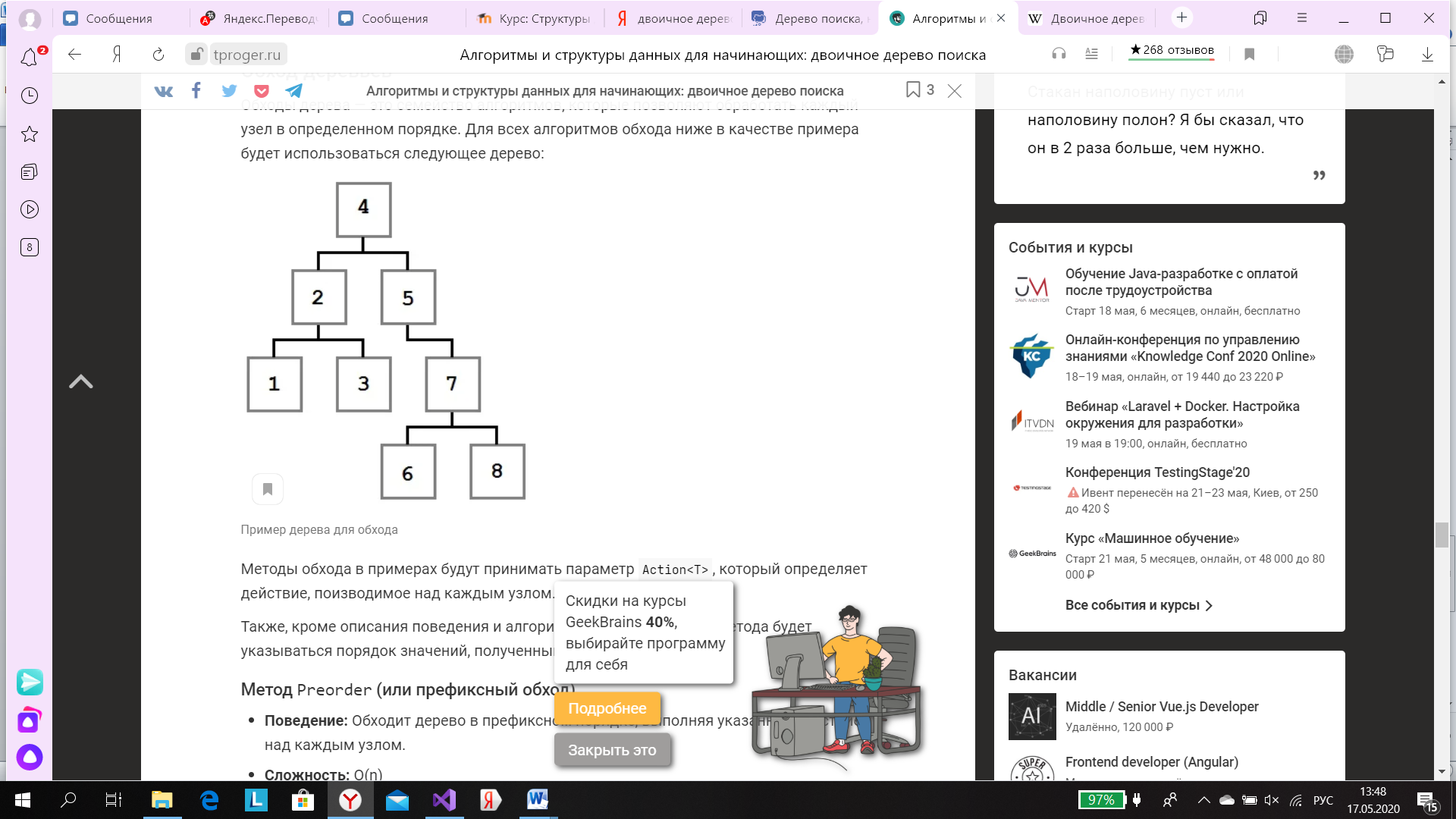
Del(X->right, i);

}

}

**Анализ двоичного дерева поиска**

Пример дерева для обхода



Симметричный обход

void S\_obhod(tree\* X) { // Симметричный

if (X) {

S\_obhod(X->left);

cout << X->i << " ";

S\_obhod(X->right);

}

}

Симметричный обход используется когда нам необходимо обойти дерево в порядке, соответствующем значениям узлов. Обходим их от наименьшего до наибольшего. То есть от левой части к правой через корень.

Сложность: O(n)

Порядок обхода: 1, 2, 3, 4, 5, 6, 7, 8

Прямой обход

void P\_obhod(tree\* X) { // Прямой

if (X) {

cout << X->i << " ";

P\_obhod(X->left);

P\_obhod(X->right);

}

}

При прямом обходе алгоритм получает значение текущего узла перед тем, как перейти сначала в левую часть, а затем в правую. Начиная от корня, сначала получаем значения корня, дальше таким же образом обходятся левый ребенок и его дети, затем правый ребенок и все его дети. Прямой обход обычно применяется для копирования дерева с сохранением его структуры.

Сложность: O(n)

Порядок обхода: 4, 2, 1, 3, 5, 7, 6, 8

Обратный обход

void O\_obhod(tree\* X) { // Обратный

if (X) {

O\_obhod(X->left);

O\_obhod(X->right);

cout << X->i << " ";

}

}

При обратном обходе посещаем левое поддерево, правое поддерево, а потом, после обхода всех детей, переходим к самому узлу. Обратный обход часто используется для полного удаления дерева, так как в некоторых языках программирования необходимо убирать из памяти все узлы явно или для удаления поддерева. Поскольку корень в данном случае обрабатывается последним, таким образом уменьшаем работу, необходимую для удаления узлов.

Сложность: O(n)

Порядок обхода: 1, 3, 2, 6, 8, 7, 5, 4

**Основные операции**

Операции присваивания ссылок, проверка условий выполняются за константное время. В худшем случае, если элемент находится в самом низу дерева, то потребуется пройти h вершин, где h- высота дерева. Значит время удаления из бинарного дерева будет составлять O(h). В другом варианте среднее время составит O(log h).

void AddDel(tree\* A, tree\* X) { // Функция для восстановления ветки, родитель которой был удален

if (A) { // Если доп. ветка существует

Push(X, A->i); // Функция добавления

if (A->left) // Если существует левый ребенок, смещаемся к нему

AddDel(A->left, X);

if (A->right) // Если существует правый ребенок, смещаемся к нему

AddDel(A->right, X);

}

}

void Del(tree\* X, int i) { // Удаление элемента

if (!X) // Если элемент дерева не существует

search -= 0; // Заглушка

else if (X->i == i) {

if (!X->left && !X->right && X->parent) { // Нет левого и нет правого и есть родитель

if (X->parent->i > i)

X->parent->left = NULL;

else

X->parent->right = NULL;

search++; // Вспомогательная переменная для идентификации удаления

delete X;

}

else if (X->left && !X->right && X->parent) { // Если есть только левый и есть родитель

X->parent->left = X->left;

search++;

delete X;

}

else if (!X->left && X->right && X->parent) { // Если есть только правый и есть родитель

X->parent->right = X->right;

search++;

delete X;

}

else if (X->left && X->right && X->parent) { // Если есть и левый, и правый и есть родитель

tree\* A = X->left;

if (X->parent->i > i) // Если значение родителя больше искомого элемента

X->parent->left = X->right; // У родителя в левую ветку вставляем правую ветку текущего элемента

else

X->parent->right = X->right;

X = X->parent;

AddDel(A, X);

}

else if (X->left && X->right && !X->parent) { // Если есть левый и правый, но нет родителя / (Корень)

tree\* L = X->left; // Создаем дубликат левой ветки

tree\* R = X->right; // И правой

X->i = R->i; // Копируем информацию правой ветки как основную

X->left = R->left;

X->right = R->right;

search++;

AddDel(L, X); // И на нее цепляем значение

}

else if (!X->left && X->right && !X->parent) { // Если нет левого и родителя, но есть правый / (Корень)

tree\* R = X->right;

X->i = R->i;

X->left = NULL;

X->right = R->right;

search++;

}

else if (X->left && !X->right && !X->parent) { // Если нет правого и родителя, но есть левый / (Корень)

tree\* L = X->left;

X->i = L->i;

X->left = L->left;

X->right = NULL;

search++;

}

}

else { // Иначе сдвигаемся и делаем проверку заново

Del(X->left, i);

Del(X->right, i);

}

}

После того, как определили узел, который нужно удалять, удаляем этот узел и пересобираем его детей функцией **AddDel**.

Вставка будет осуществляться за О(h), где h-высота дерева, т.к в худшем случае потребуется пройти h вершин, двигаясь вниз. Операция создания узла выполняется за константное время.

void Push(tree\*& X, int x) {

tree \*c = node(x); // Создаем элемент дерева

if (!X) // Если дерево не существует

X = c; // Делаем корень

else {

tree\* a = X; // Создаем копию для обхода

while (a) { // Пока не дошли до конца

if (a->i > x) {

if (a->left)

a = a->left;

else {

a->left = c;

c->parent = a;

break;

}

}

else if (a->right)

a = a->right;

else {

a->right = c;

c->parent = a;

break;

}

}

}

}