ГЛАВА 17

Двоичные деревья

ПРИМЕЧАНИЕ

В этой главе рассматривается одна из самых интересных и полезных базовых структур данных — двоичное дерево. Двоичные деревья представляют собой идеальный пример использования рекурсии и указателей для выполнения очень полезных действий. Тем не менее работа с ними требует уверенного понимания рекурсии и концепций в основе связанных списков. Почему я так считаю? Я неоднократно видел, как студенты сталкиваются с проблемами при изучении двоичных деревьев, оставляя указатели и рекурсию без должного внимания. В сущности, в двоичных деревьях нет ничего сложного, чтобы их не понять, однако вам необходимы прочные фундаментальные знания. Если освоение концепций этой главы вызовет сложности, скорее всего, стоит глубже разобраться в указателях или рекурсии. Перечитайте соответствующие главы книги и выполните упражнения.

Для чего нужны двоичные деревья?

Связанные списки — отличный способ перечисления предметов, но поиск конкретного элемента в таком списке может занять много времени. Более того, если данные представляют собой один большой неструктурированный

список, даже массивы не помогут с ним работать. Можно попытаться упорядочить массив: это даст возможность очень быстро искать в нем данные. Тем не менее вставлять элементы в массив будет трудно: если вы не захотите нарушить упорядоченность массива, то при добавлении нового придется выполнять много дополнительной работы. Более того, быстро искать данные очень важно. Вот несколько примеров.

1. Если вы создаете глобальную сетевую многопользовательскую ролевую игру вроде World Of Warcraft и хотите, чтобы игроки могли оперативно подключаться, вы должны быстро находить конкретного игрока.
2. Если вы создаете программное обеспечение для работы с кредитными картами и требуется обрабатывать миллионы транзакций в час, вы должны быстро определять баланс банковского счета по номеру кредитной карты.
3. Если вы отображаете адресную книгу на устройстве с небольшими ресурсами (например, смартфоне), недопустимо, чтобы пользователю приходилось ждать из-за того, используется медленная структура данных.

В этой главе речь пойдет об инструментах, которые позволяют решать эти и многие другие задачи.

Основная идея в том, что элементы хранятся в структуре, напоминающей связанный список. Она позволяет использовать указатели для структурирования памяти (так же, как мы делали со связанными списками), но упрощает поиск значений. Такая структура памяти сложнее обычного списка.

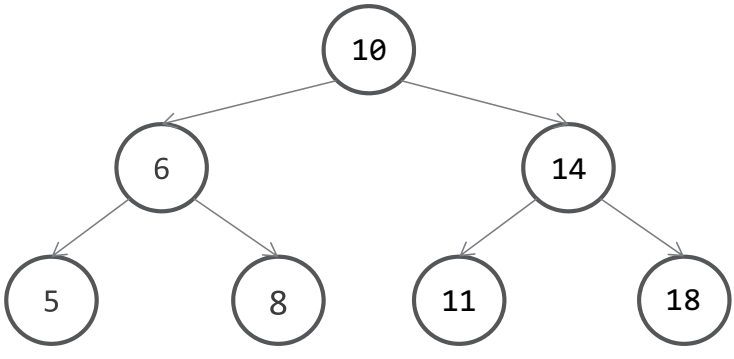
Что такое двоичное дерево?

Рассмотрим суть структурирования данных в виде двоичного дерева. Когда вы начинали изучать программирование, в вашем распоряжении были только массивы, которые позволяли организовывать данные лишь в виде последовательного списка. Связанный список использует указатели для наращивания последовательного списка, но не использует их гибкость, позволяющую строить более сложные структуры.

Что я понимаю под сложной структурой в памяти? Прежде всего, можно создать структуру данных, у элементов которой может быть несколько следующих. Для чего? Если есть два следующих узла, один из которых может содержать элементы со значением меньше, чем у текущего элемента, а другой — элементы со значением больше, чем у текущего. Такая структура называется двоичным деревом.

Название «двоичное дерево» говорит о том, что у каждого узла есть не более двух ветвей. «Следующие» узлы называются дочерними, а текущий узел по отношению к дочернему — родительским.

Двоичное дерево можно изобразить так:



Обратите внимание, что в этом дереве каждый левый дочерний элемент меньше своего родителя, а правый дочерний элемент — больше. Узел 10 является родителем всего дерева, а его дочерние узлы, 6 и 14, — родителями своих деревьев меньшего размера, которые называются поддеревьями.

Важное свойство двоичного дерева в том, что каждый дочерний элемент узла сам является двоичным деревом. Если учесть это свойство и принять во внимание, что дочерние узлы слева меньше своих родителей, а справа — больше, легко разработать алгоритм поиска узла в дереве. Сначала нужно сравнить текущий узел с искомым значением: если они равны, поиск завершен. Если искомое значение меньше, чем у текущего узла, нужно перейти влево, в противном случае — вправо. Этот алгоритм работает, потому что все узлы в левом поддереве меньше текущего, а в правом — больше.

Желательно, чтобы дерево было сбалансированным, то есть содержало одинаковое число узлов в левом и правом поддеревьях. В этом случае размер каждого дочернего двоичного дерева равен примерно половине размера всего дерева, и, ища значение в дереве, при каждом переходе на дочерний узел можно исключать из поиска половину элементов. Если дерево состоит из 1000 элементов, моментально отбрасывается около 500 элементов и поиск продолжается в 500-элементном дереве, в котором снова можно отбросить половину (то есть 250) элементов. Таким образом, поиск значения не занимает много времени, если на каждом шаге количество элементов сокращается вдвое. Чтобы найти искомый элемент, необходимо разделить дерево $\log_2 n$ раз, где n — число элементов дерева. Это число невелико даже

для очень больших деревьев (оно равно 32 для дерева с 4 млрд элементов; поиск в таком дереве будет почти в 100 млн раз быстрее, чем в связанном списке такого же размера, где придется просматривать каждый элемент). Тем не менее если дерево не сбалансировано, его деление точно пополам может оказаться невозможным. В худшем случае у каждого узла будет один дочерний, и дерево превратится в псевдосвязанный список (с дополнительными указателями), время поиска в котором равно *n*.

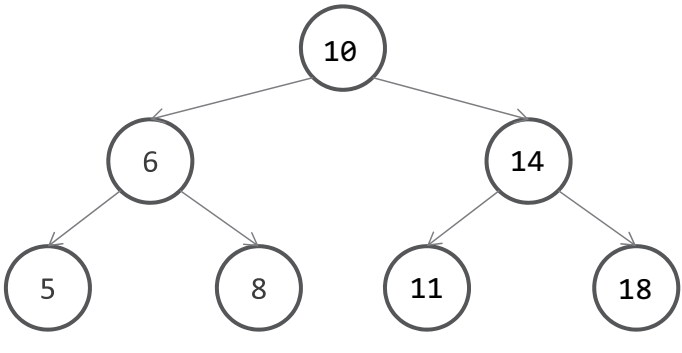
Если дерево относительно (необязательно идеально) сбалансировано, поиск узлов в нем существенно быстрее, чем в связанном списке. Все это возможно благодаря тому, что вы способны структурировать память по своему желанию, не ограничиваясь простыми списками[1].

Соглашение о терминах

Для удобства изучения примеров кода, работающего с двоичными деревьями, сначала договоримся о принципах изображения и именования различных частей дерева.

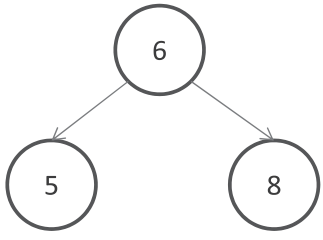
Простейшее дерево — это **пустое дерево**, которому соответствует значение `NULL`. Я не буду изображать ссылки на пустые деревья на схеме.

Конкретные поддеревья я буду называть так: `<дерево с вершиной [значение родительского узла]>`. Например,



[1] Простейшее двоичное дерево редко имеет такую же структуру, как связанный список, что обусловливается порядком вставки элементов в дерево. Существуют более сложные виды двоичных деревьев, которые всегда сбалансированы, однако их рассмотрение выходит за рамки этой книги. Примером такой структуры является красно-черное дерево: https://ru.wikipedia.org/wiki/Красно-черное_дерево

`<дерево с вершиной 6>` относится к следующему поддереву:



Реализация двоичных деревьев

Рассмотрим код простой реализации двоичного дерева. Сначала объявим структуру узла:

```
struct node
{
    int key_value;
    node *p_left;
    node *p_right;
};
```

Этот узел хранит значение в виде простого целого числа `key_value` и содержит два дочерних поддерева с названиями `p_left` и `p_right`.

Основные действия над двоичными деревьями — вставка нового узла, поиск значения в дереве, удаление узла и уничтожение дерева для освобождения памяти.

```
node* insert (node* p_tree, int key);
node *search (node* p_tree, int key);
void destroyTree(node* p_tree);
node *remove(node* p_tree, int key);
```

Вставка в дерево

Начнем с того, что реализуем вставку узла в дерево с помощью рекурсивного алгоритма. Рекурсия отлично сочетается с деревьями, поскольку каждое дерево содержит два дерева меньшего размера, а следовательно, имеет рекурсивную структуру (если бы дерево включало, скажем, массив или указатель на связанный список, а не на другие деревья, оно не было бы рекурсивным).

Наша функция принимает ключ и существующее дерево (возможно, пустое) и возвращает новое дерево, содержащее вставленное значение.

```
node* insert(node *p_tree, int key)
{
    // Базовый вариант: мы достигли пустого дерева и должны
    // вставить в него новый узел
    if(p_tree == NULL)
    {
        node* p_new_tree = new node;
        p_new_tree->p_left = NULL;
        p_new_tree->p_right = NULL;
        p_new_tree->key_value = key;
        return p_new_tree;
    }
    // В зависимости от значения узла решаем,
    // в какое поддерево его вставить,
    // в левое или правое
    if(key < p_tree->key_value)
    {
        // Строим новое дерево на основе p_tree->left,
        // добавляя в него ключ. Заменяем текущее
        // значение p_tree->left указателем на новое дерево.
        // Нужно задать значение указателя p_tree->p_left,
        // если p_tree->left равен NULL.
        // (Если он не равен NULL,
        // то p_tree->p_left не изменится,
        // но в этом нет ничего страшного.)
        p_tree->p_left = insert(p_tree->p_left, key);
    }
    else
    {
        // Вставка с правой стороны, в точности
        // симметричная вставке с левой стороны
        p_tree->p_right =
        insert(p_tree->p_right, key);
    }
    return p_tree;
}
```

Базовая логика этого алгоритма такова: если дерево пустое — создать новое дерево; в противном случае, если вставляемое значение больше текущего узла, вставить его в левое поддерево и заменить новым поддеревом; в противном случае вставить значение в правое поддерево и заменить его новым поддеревом.

Рассмотрим этот код в действии, превратив пустое дерево в дерево с двумя узлами. При вставке значения 10 в пустое дерево (`NULL`) мы сразу попадаем в базовый вариант. В результате получается очень простое дерево:

Оба его дочерних дерева являются пустыми.

При вставке значения 5 в это дерево мы совершаем вызов:

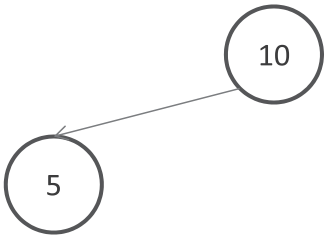
```
insert(<tree with parent 10>, 5)
```

Поскольку 5 меньше 10, получаем рекурсивный вызов в левое поддерево:

```
insert(NULL, 5)
insert(<дерево с родителем 10>, 5)
```

Вызов `insert(NULL, 5)` создаст новое дерево и вернет его:

Получив возвращенное дерево, функция `insert(<дерево с родителем 10>, 5)` свяжет два дерева воедино. В данном случае дочерний узел элемента 10 слева раньше был равен `NULL`, поэтому он становится абсолютно новым деревом:



Если теперь добавить в дерево значение 7, получим:

```
insert(NULL, 7)
insert(<дерево с родителем 5>, 7)
insert(<дерево с родителем 10>, 7)
```

Вызов

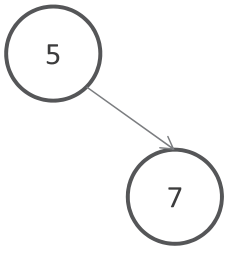
```
insert(NULL, 7)
```

возвращает новое дерево:

Затем

```
insert(<дерево с родителем 5>, 7)
```

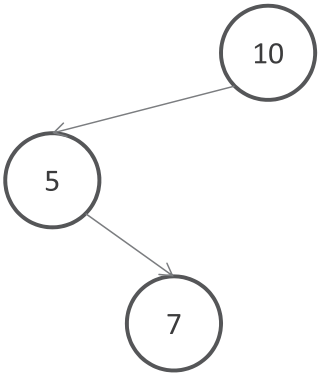
подключает поддерево 7 следующим образом:



И наконец, это дерево возвращается в вызов

```
insert( <дерево с родителем 10>, 7 )
```

который пристыковывает его обратно:



Поскольку у узла 10 уже был указатель на узел, содержащий число 5, повторное связывание левого дочернего узла элемента 10 с родителем 5 не является строго обязательным, однако оно устраняет необходимость дополнительно проверять, является ли поддерево пустым.

Поиск в дереве

Теперь рассмотрим реализацию поиска в дереве. Ключевая логика поиска почти в точности повторяет логику вставки в дерево: сначала мы проверяем два базовых варианта (мы нашли узел или ищем в пустом дереве), а затем выясняем, в каком поддереве выполнять поиск, если не попадаем в базовый вариант.