Бинарные деревья

А. Г. Фенстер, http://info.fenster.name

6 декабря 2010 г.

Определение

Бинарное или *двоичное дерево* — структура данных, представляющая собой либо пустое множество, либо узел, состоящий из поля данных и двух бинарных деревьев, называемых левым и правым поддеревьями или потомками этого узла. Узел, не являющийся потомком ни одного из узлов дерева, мы будем называть корнем бинарного дерева.

В программе на C мы будем представлять дерево такой структурой:

```
struct node
{
    int data;
    struct node *left, *right;
}
```

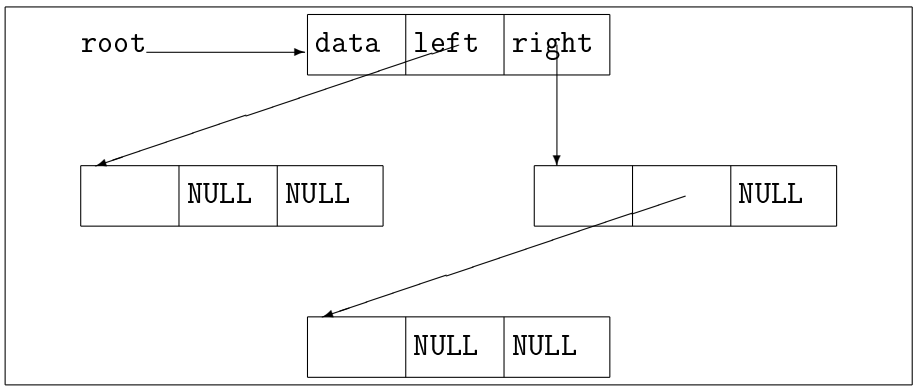

Рис. 1. Бинарное дерево в программе на C.

Переменная `struct node *root` будет указывать на корневой узел дерева (мы будем считать дерево пустым, если `root == NULL`).

Конечно, никто обычно не рисует все три поля (`data`, `left`, `right`) и стрелки на картинках. Обычно бинарные деревья рисуют так, как показано на рис. 2 (числа — значения полей `data`).

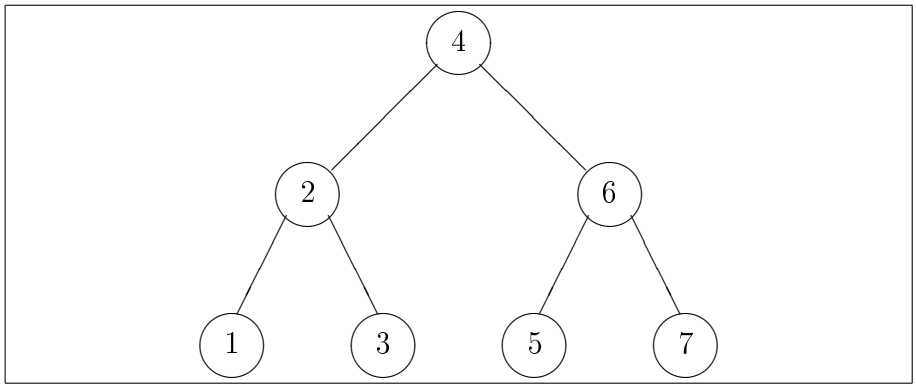

Рис. 2. Пример бинарного дерева.

Простейшие операции с деревьями

Пусть `root` — корень бинарного дерева, и нам нужно подсчитать сумму всех его элементов. На примере этой простой задачи мы разберём один из способов перебора всех узлов дерева, а далее задача перебора узлов будет описана подробнее.

Будем считать сумму при помощи рекурсии. Наша функция будет принимать параметром корень дерева и возвращать результат (число):

```
int sum(struct node *root)
{
    ...
}
```

Говоря о рекурсии, в первую очередь всегда нужно задумываться о «терминальной ветви», то есть понять, в каком случае ответ очевиден и не требует никаких рекурсивных вызовов. В нашей задаче самый простой случай — пустое дерево, не содержащее ни одного узла. Суммой пустого набора чисел принято (и логично) считать 0, так что пишем

```
int sum(struct node *root)
{
    if (!root)  // то же самое, что if (root == NULL)
        return 0;
    ...
}
```

Как вычислить сумму элементов непустого дерева? Сведём задачу к аналогичной задаче меньшего размера: сумма элементов дерева — это число в корне дерева плюс суммы левого и правого поддеревьев, а эти суммы можно вычислить при помощи функции `sum` (той самой, которую мы сейчас пишем). Получаем рекурсию:

```
int sum(struct node *root)
{
    if (!root)
        return 0;
    return root->data // значение корня
         + sum(root->left)   // сумма левого поддерева
         + sum(root->right); // сумма правого поддерева
}
```

Если вызвать эту функцию от корня дерева, рекурсия рано или поздно спустится до самых нижних узлов дерева (*листьев* — узлов, не имеющих ни левого, ни правого поддерева). Будут посещены все узлы дерева, и в итоге результатом функции будет сумма всех узлов. О порядке, в котором функция `sum` перебирает узлы дерева, мы поговорим далее.

Скобочная запись дерева

Назовём скобочной записью дерева представление дерева, которое генерируется следующей функцией:

```
void print(struct node *root)
{
    if (!root)
    {
        putchar('x');
        return;
```

```
    }
    putchar('(');
    printf("%d", root->data);
    print(root->left);
    print(root->right);
    putchar(')');
}
```

Для дерева на рис. 2 эта функция напечатает следующую строку:
`(4(2(1xx)(3xx))(6(5xx)(7xx)))`
Эта строка однозначно определяет дерево, по которому она была напечатана.

Напишем функцию, которая создаст дерево по его скобочной записи. Функция будет возвращать указатель на корень созданного дерева. Как и обе предыдущие функции, она будет рекурсивной. Начнём с терминальной ветви:

```
struct node *read(void)
{
    int c = getchar();
    if (c == 'x')
        return NULL;
    ...
}
```

Эта функция корректно обрабатывает запись `x`. Если же дерево содержит хотя бы один узел, запись будет начинаться со скобки (мы считаем для простоты, что ввод всегда корректен). Прочитав скобку и число (значение узла), нужно перейти к чтению левого и правого поддерева — а для этого мы просто рекурсивно вызовем ту же функцию `read`. В конце останется лишь прочитать закрывающую скобку.

```
struct node *read(void)
{
    struct node *p;
    int c = getchar();
    if (c == 'x')
        return NULL;
    // иначе c -- скобка и мы читам узел
```

```
    p = malloc(sizeof(*p));
    scanf("%d", &p->data);
    p->left = read();
    p->right = read();
    getchar(); // закрывающая скобка
    return p;
}
```

Заметим, что закрывающая скобка не оказывает никакого влияния на разбор и в принципе может быть опущена (тогда нужно будет убрать соответствующий вызов `getchar` из функции `read`). Она нужна лишь «для красоты».

Обходы деревьев

Функции `sum` и `print` в предыдущих разделах выполняют одну и ту же работу: перебирают все узлы дерева и для каждого из них делают какие-либо действия. Этот процесс называется *обходом дерева*. Выделяют два основных способа обойти все узлы: *в глубину* (рекурсивно или при помощи стека) и *в ширину* (при помощи очереди). Рассмотрим эти способы подробнее.

Обход в глубину

При обходе в глубину функция обхода вызывается рекурсивно для левого и правого поддерева дерева. Функции `sum` и `print` выполняют именно такой обход. Пусть `visit` — некоторая операция, которую мы должны выполнить для каждого узла дерева (например, напечатать значение). В зависимости от порядка выполнения трёх действий (посетить корень, уйти влево и уйти вправо) возможны три варианта обхода дерева в глубину, которые называют соответственно префиксным (прямым), инфиксным (симметричным) и постфиксным (обратным). Мы рассматриваем только варианты обхода, в которых левое поддерево посещается раньше правого; понятно, что снятие этого ограничения даст ещё три варианта.

Исключительно для экономии места объявим тип:

```
typedef struct node n;
```

и выпишем три варианта обхода в глубину рядом, чтобы отличие было хорошо заметно. Под каждой из функций указан порядок перебора узлов для дерева с рис. 2. Пожалуйста, убедитесь, что вы понимаете, почему каждая из функций даст именно такой порядок перебора (особенно много проблем с пониманием обычно вызывают инфиксный и постфиксный обходы).

префиксный (корень, левое, правое)	инфиксный (левое, корень, правое)	постфиксный (левое, правое, корень)

`void prefix(n *root)`
`{`
`  if (!root)`
`  return;`
`  visit(root);`
`  prefix(root->left);`
`  prefix(root->right);`
`}`
4 2 1 3 6 5 7
	`void infix(n *root)`
	`{`
	`  if (!root)`
	`  return;`
	`  infix(root->left);`
	`  visit(root);`
	`  infix(root->right);`
	`}`
	1 2 3 4 5 6 7
		`void postfix(n *root)`
		`{`
		`  if (!root)`
		`  return;`
		`  postfix(root->left);`
		`  postfix(root->right);`
		`  visit(root);`
		`}`
		1 3 2 5 7 6 4

Далеко не в каждой задаче принципиально, какой из трёх обходов использовать, но для некоторых задач выбрать правильный обход очень важно (например, инфиксный обход пригодится при работе с деревьями поиска, о которых будет рассказано ниже).

Общепринятое название обхода в глубину — DFS (сокращение от английского depth-first search).

Обход в ширину

Принципиально иной способ перебрать узлы дерева — воспользоваться для запоминания узлов очередью. Изначально поместим в очередь корень дерева, а затем, пока очередь не пуста, будем извлекать из неё узлы по одному и помещать в очередь левого и правого потомков извлечённого узла. Таким образом, рано или поздно переберём все узлы дерева:

```
void bfs(struct node *root)
{
    if (!root)
        return;
    enqueue(root);
```

```
    while (!empty())
    {
        struct node *curr = dequeue();
        visit(curr);
        if (curr->left)
            enqueue(curr->left);
        if (curr->right)
            enqueue(curr->right);
    }
}
```

В этом примере мы считаем, что очередь с интерфейсными функциями `enqueue` (положить элемент в очередь), `dequeue` (взять элемент из очереди) и `empty` (проверка на пустоту) хранит элементы типа `struct node *`.

Выполнив обход в ширину для какого-либо дерева на листе бумаги, можно заметить, что узлы дерева перебираются в порядке удаления от корня (скажем, для дерева с рис. 2 порядком обхода будет 4 2 6 1 3 5 7). Это свойство делает обход в ширину удобным для решения задачи типа «найти ближайший к корню узел, удовлетворяющий некоторому условию».

Общепринятое название обхода в ширину — BFS (сокращение от английского breadth-first search).

Обход в глубину при помощи стека

Любую рекурсивную функцию можно переписать без использования рекурсии, но с использованием стека. В случае с обходом в глубину это весьма просто сделать. Для обхода графа в **глубину** при помощи стека мы в функции обхода в **ширину** заменим обращения к очереди на обращения к интерфейсным функциям стека (`push`, `pop`, `empty`) и поменяем порядок запоминания левого и правого узлов, чтобы новая функция обходила узлы в том же порядке, что и функция `prefix`. Получим следующий код:

```
void dfs_stack(struct node *root)
{
    if (!root)
        return;
```

```
    push(root);
    while (!empty())
    {
        struct node *curr = pop();
        visit(curr);
        if (curr->right)
            push(curr->right);
        if (curr->left)
            push(curr->left);
    }
}
```

В заключение нужно сказать, что те же принципы обхода применяются и при обходе вершин графов, но с одним важным дополнением: в графах существуют циклы, так что необходимо вставить в функции обхода «защиту» от зацикливания. Подробнее об этом будет рассказано в соответствующей теме.

Бинарные деревья поиска

Прежде чем начать разговор по теме, стоит сказать, что в русской литературе (и особенно в интернете) можно встретить словосочетание «дерево двоичного поиска» или «дерево бинарного поиска», что мне сейчас кажется не вполне корректным: к бинарному поиску (поиску делением пополам в упорядоченном массиве) рассматриваемая тема никакого отношения не имеет. Проблема, вероятно, в переводе английского словосочетания «binary search tree», в котором мне хочется делать акцент на search tree, а не на binary search, и переводить его, соответственно, как «бинарное (или двоичное) дерево поиска». А вот к использованию слова «бинарное» вместо «двоичное» в этом контексте никакого отвращения я не испытываю.

Итак, *бинарное дерево поиска* — такое бинарное дерево, в котором для любого узла `p` выполняется следующее условие: значение каждого узла из левого поддерева узла `p` меньше значения узла `p`, а значение каждого узла из правого поддерева узла `p` больше значения узла `p` (равенство можно включить в одно из условий). Дерево, изображённое на рис. 2, является деревом поиска, поскольку для каждого его узла условие выполняется.

Такие деревья всем хороши:

- значения узлов очень легко вывести в порядке возрастания, достаточно выполнить инфиксный обход дерева (это напрямую следует из определений бинарного дерева поиска и инфиксного обхода);

- узел легко добавить в дерево с сохранением свойства дерева поиска: узел должен добавляться на нижний уровень дерева; двигаясь от корня вниз (влево или вправо на каждом шаге), ищем узел, у которого нет левого (если значение нового узла меньше) или правого (если больше) сына, и вставляем новый узел туда;

- узел сравнительно легко удалить из дерева: нужно рассмотреть четыре случая. Узел без сыновей, т. е. лист — самый простой случай; удаляемый узел имеет одного сына — тоже просто; и два сложных случая: узел с двумя сыновьями и корень. В двух последних случаях левое поддерево удаляемого узла «прицепляется» к самому левому узлу правого поддерева, а правое поддерево становится на место удаляемого узла (или наоборот: можно «прицепить» правое поддерево к самому правому узлу левого поддерева).

Можно даже легко реализовать сортировку при помощи вставки в бинарное дерево поиска: вставлять каждое число в дерево и затем напечатать дерево при помощи инфиксного обхода. К сожалению, в худшем случае сложность такого алгоритма сортировки будет $O(N^2)$, и худшими случаями могут быть как тривиальные (когда числа заранее упорядочены и дерево выстраивается в «линеечку»), так и более интересные последовательности типа 1 10 2 9 3 8 4 7 5 6, для которой будет построено «зигзагообразное» дерево, всё равно являющееся по сути линейным списком. Однако, если известно, что числа на вход подаются совершенно произвольные, можно надеяться, что сортировка при помощи дерева поиска отработает за разумое время.

Существуют алгоритмы, позволяющие поддерживать глубину дерева поиска более-менее равномерной (порядка $O(\log_2 N)$) (самые известные — AVL-деревья и красно-чёрные деревья). В обоих случаях после любого изменения структуры дерева (вставки или удаления узла) проверяется, не нарушена ли сбалансированность дерева, и производятся необходимые преобразования дерева. Мы не будем сейчас рассматривать эти темы подробно.