Михаил Густокашин. Бинарные деревья.

1 Деревья

В программировании деревом называется структура, которая напоминает обычное дерево, поэтому терминология довольно понятна. Дерево, это динамическая структура данных, где, обычно, каждый элемент имеет несколько ссылок или на один элемент ссылается несколько других элементов. Как и в обычном дереве у нашей структуры данных есть «корень», «узлы» и «листья». Другая часть терминологии происходит уже от генеалогических деревьев, в структуре данных также определены понятия «предок» (непосредственного предка также называют «отцом»), «потомок», а также разнообразные «сыновья», «дедушки», «дяди» и проч. Обратите внимание, что у любого элемента дерева может быть только один непосредственный предок.

Самый простой, но в тоже время не всегда применимый способ хранения дерева — это т.н. «корневое дерево», когда каждый элемент содержит ссылку только на своего непосредственного предка. Этот способ хранения нельзя применять, когда нам необходимо обращаться к потомкам.

Другой распространенный способ хранения деревьев — это бинарные (двоичные) деревья. В этом случае каждый узел хранит указатели на двух потомков, которых называют левым и правым сыном. Иногда в бинарных деревьях также хранится ссылка на отца.

Существует также класс деревьев, в котором количество потомков произвольно. В классической литературе для этого обычно используют схему с двумя ссылками «левый сын и правый брат», которая, по сути, аналогична списку детей, но мы будем использовать в таких случаях вектор детей.

Общим свойством деревьев является то, что, выбирая в качестве корня произвольный элемент, мы можем обращаться с поддеревом (т.е. с деревом, у которого корень — выбранный узел) точно так же, как и с целым деревом. Это позволяет удобно обрабатывать деревья с помощью рекурсивных функций.

Размером дерева называется количество элементов в нем. Высота дерева — максимальный путь от корня до листа.

Более подробно каждый из этих способов мы изучим при рассмотрении конкретных примеров.

2 Бинарное дерево поиска

Мы уже рассматривали пример дерева, обладающего некоторым свойством (кучу). Теперь рассмотрим дерево с другим свойством: левый сын имеет ключ, меньший, чем в данном узле, а правый сын — больший. Такие деревья называются бинарными деревьями поиска. Из этого свойства следует, что все элементы в левом поддереве меньше данного элемента, а в правом, наоборот, больше. Обычно, в бинарных деревьях поиска нет двух элементов с одинаковыми ключами, но если это условие не выполнено, то можно либо в каждом узле хранить количество элементов с таким ключом или добавлять его в произвольное поддерево. Ключами в бинарном дереве поиска могут быть любые сравнимые элементы (мы рассмотрим случай для целых чисел).

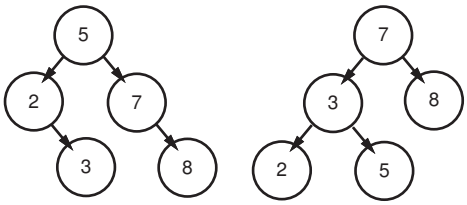

Рис. 1: Примеры бинарных деревьев

На рисунке приведены примеры двух бинарных деревьев поиска. Как видно, деревья могут отличаться, хотя и содержат одинаковые элементы. Это зависит от порядка добавления элементов, хотя для разных порядков добавления могут получиться одинаковые деревья.

Для чего же нужно бинарное дерево поиска и почему оно так называется? На самом деле, бинарное дерево поиска позволяет искать элемент по ключу за $O(\log N)$ в среднем (как и бинарный поиск в отсортированном массиве) и, в отличие от отсортированного массива, позволяет производить вставку элемента за $O(\log N)$ в среднем (для отсортированного массива эта операция занимает $O(N)$). Кроме того, из бинарного дерева поиска можно за $O(N)$ получать отсортированный массив и делать с ним другие полезные операции, такие как поиск порядковой статистики за $O(\log N)$ или определение порядкового номера элемента также за $O(\log N)$.

Опишем структуру, хранящую узел дерева, а затем будем вводить функции работы с деревом.

```
typedef struct _node
{
  int key;
```

```
  struct _node *left, *right;
} node;
```

Вначале введем функцию добавления, которая будет получать на вход указатель на корень дерева и ключ добавляемого элемента:

```
node* crnode(int val)
{
  node* nnode = (node*) malloc(sizeof(node));
  nnode->key = val;
  nnode->left = NULL;
  nnode->right = NULL;
  return nnode;
}

node* add_tree(node *root, int val)
{
  if (NULL == root) root = crnode(val);
  if (val < root->key)
    if (NULL == root->left)
      root->left = crnode(val);
    else
      add_tree(root->left, val);
  if (val > root->key)
    if (NULL == root->right)
      root->right = crnode(val);
    else
      add_tree(root->right, val);
  return root;
}
```

Здесь `crnode` — вспомогательная функция, создающая новый узел дерева без потомков с заданным ключом. Основная функция подбирает позицию для вставки нового элемента и приделывает его в качестве листа. Если такой элемент уже был в дереве, то функция добавления оставит дерево без изменения. Как видно, сложность этой функции $O(H)$, где H — высота дерева.

Функция поиска элемента будет еще проще. Она будет возвращать указатель на элемент, содержащий искомый ключ или `NULL`, если элемента с таким ключом в дереве нет:

```
node* find_tree(node *root, int val)
{
  if (NULL == root)
    return NULL;
  if (val == root->key)
    return root;
  if (val < root->key)
    return find_tree(root->left, val);
  if (val > root->key)
    return find_tree(root->right, val);
}
```

Несколько более интересна функция удаления элемента из дерева. Листья удаляются очень просто, путем очистки указателя из предка. Узлы с одним потомком также удаляются несложно — в предке ссылка перекидывается на этого потомка. Несколько сложнее дело обстоит с удалением узла, у которого два потомка. Для этого нужно обменять его ключ с самым левом потомком из правого поддерева (или с самым правым потомком левого поддерева), а затем уже удалять этого потомка (он либо не будет иметь потомков, либо будет иметь всего одного потомка, иначе он не может оказаться самым левым или правым). Для начала проиллюстрируем это на рисунке. Пусть мы хотим удалить элемент с ключом 7 и выбираем самого правого потомка из левого поддерева:

В левой части рисунка мы находим самый правый элемент из левого поддерева и записываем его значение в «удаляемый» узел. На средней части рисунка мы осуществляем поиск узла с ключом, совпадающего с корневым ключом, в левом поддереве. В правой части рисунка показано удаление этого элемента (у него обязательно только одно поддерево). Опишем удаление элемента в виде функции:

```
int rightmost(node *root)
```

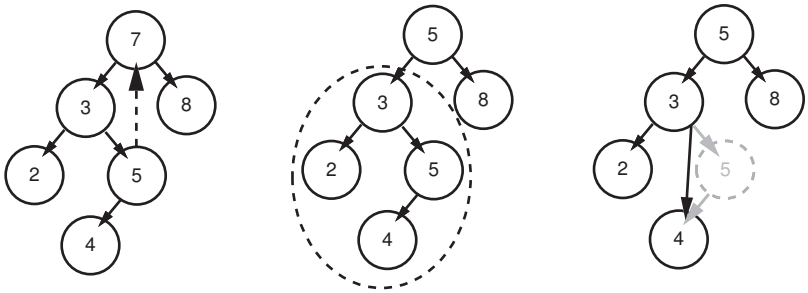

Рис. 2: Удаление элемента из бинарного дерева поиска

```
{
  while (root->right != NULL)
  root = root->right;
  return root->key;
}

node* del_tree(node *root, int val)
{
  if (NULL == root) return NULL;
  if (root->key == val) {
    if (NULL == root->left && NULL == root->right) {
      free(root);
      return NULL;
    }
    if (NULL == root->right && root->left != NULL) {
      node *temp = root->left;
      free(root);
      return temp;
    }
    if (NULL == root->left && root->right != NULL) {
      node *temp = root->right;
      free(root);
      return temp;
    }
    root->key = rightmost(root->left);
    root->left = del_tree(root->left, root->key);
    return root;
  }
  if (val < root->key) {
    root->left = del_tree(root->left, val);
    return root;
  }
  if (val > root->key) {
    root->right = del_tree(root->right, val);
    return root;
  }
  return root;
}
```

Все эти функции имеют сложность $O(H)$ и используют $O(H)$ вспомогательной памяти.

Сначала дерево должно инициализироваться `NULL`, например, так: `node *tree = NULL;`

Каждый вызов функции добавления или удаления должен выглядеть так:

```
tree = add_tree(tree, x);
tree = del_tree(tree, x);
```

Рассмотрим задачу вывода упорядоченного массива по дереву. Такой способ называется прямым обходом дерева: сначала мы вызываем рекурсивынй обход для левого поддерева (меньшие числа), затем выводим текущее значение, а затем вызываем рекурсию для правого поддерева (большие числа).

Функция вывода массива по дереву выглядит так:

```
void print_tree(node *root)
```

```
{
  if (root != NULL) {
    print_tree(root->left);
    printf("%d ", root->key);
    print_tree(root->right);
  }
}
```

Сложность этой функции составляет $O(N)$.

Несложно реализовать функцию вывода, например, вывести все числа из диапазона от x до y в возрастающем порядке, для этого достаточно добавть в функцию вывода дерева несколько проверок.

Большинство функций имеет сложность $O(H)$, т.е. линейно зависят от высоты дерева. Сама высота дерева может меняться, например, если добавлять в дерево возрастающую последовательность, то дерево выродится в список и N будет равно H.

В случае, если дерево составляется из случайных элементов (что в среднем и бывает), H будет равно $O(\log N)$ и функции будут работать достаточно быстро. В простейшем случае, если нам сразу дается список всех элементов, которые будут добавлены в дерево, мы можем использовать функцию `random_shuffle` из **STL**, которая за достаточно быстрое время осуществит случайные перестановки в исходном массиве и избавит нас от проблем специально подобранных «плохих» данных.

К сожалению, воспользоваться таким способом получается далеко не всегда и необходимо искать другое решение проблемы.