Центр дополнительного образования детей
Дистантное обучение

117630, Москва, ул. акад. Челомея, д. 8б, тел./факс 936-3104

Курс «Олимпиадные задачи по программированию».
Преподаватель: Михаил Сергеевич Густокашин

Лекция 6
Структуры данных (2)

Оглавление:

msg@list.ru
http://g6prog.narod.ru
http://desc.ru

Реализация динамического выделения памяти на статическом массиве.

Начнем лекцию с рассмотрения «алгоритма», с помощью которого можно реализовывать динамические структуры на статическом массиве. Это позволит нам понять, как операционная система выделяет память по запросам, а в некоторых ситуациях заметно ускорить выполнение программы. Не секрет, что выделение и очистка памяти (`malloc` и `free`) занимают довольно значительное время, которое тратится на вызов и работу функций ОС.

Будем реализовывать выделение и очистку памяти на массиве с помощью односвязного списка свободных ячеек. В качестве указателя у нас будет выступать индекс ячейки. Будем хранить «указатель» на первую свободную ячейку в переменной `ffree` (first free – первый свободный элемент), сам массив будет состоять из ячеек такого типа:

```
typedef struct {
     <type> data;
     int next;
} val;
```

Здесь <type> - тип переменной, в которой хранится осмысленное значение (data), а next – служебное поле, указывающее на следующий свободный элемент. Практически всегда можно обойтись без дополнительного поля next, т.к. такой тип организации используется для динамических структур (например, если мы хотим хранить список, который в свою очередь имеет «указатель» на следующий элемент, то дублировать этот указатель нет смысла).

Перед тем как приступить к работе следует для каждого индекса i установить поле next в i+1, а переменную ffree в 0. Это будет означать, что весь массив представляет собой пустой список, а первая свободная ячейка имеет индекс 0.

Пусть мы хотим выделить новый участок памяти в массиве mem и установить на него указатель it. Это будет выглядеть так:

```
it = ffree;
ffree = mem[ffree].next;
mem[it].next = -1;
```

Мы перемещаем указатель на начало списка на следующий свободный элемент из списка. В принципе, поле next для указателя it можно не менять или устанавливать осмысленное значение. Теперь рассмотрим удаление элемента по указателю it:

```
mem[it].next = ffree;
ffree = it;
```

Здесь все тоже более или менее понятно: освободившуюся ячейку мы записываем в начало списка свободных ячеек. Примерно так и работают алгоритмы выделения памяти в ОС, хотя, конечно, там используются более изощренные методы, т.к. ОС умеет выделять участки памяти разной длины и хранит для них эффективные индексы, в случае необходимости осуществляет перестановки и копирование в памяти и проч.

Такой метод дает некоторый выигрыш в скорости и его надо использовать только в тех случаях, когда вы эффективно реализуете неэффективный алгоритм в надежде обхитрить жюри, подсунув им алгоритмически неэффективное решение, укладывающееся в TL.

Обратите внимание, что реализация подобного «динамического выделения памяти» на динамически расширяемом массиве не имеет особого смысла, т.к. при этом теряется эффективность (теряется время на перевыделения памяти).

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.

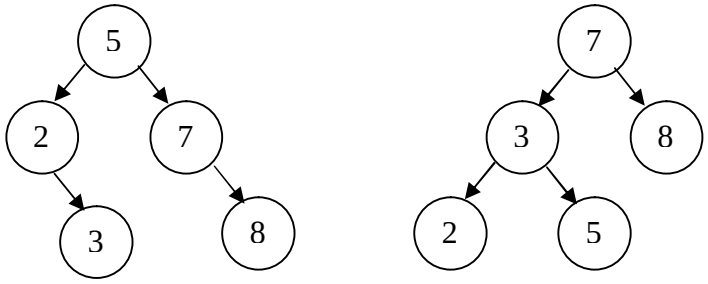
Общим свойством деревьев является то, что, выбирая в качестве корня произвольный элемент, мы можем обращаться с поддеревом (т.е. с деревом, у которого корень – выбранный узел) точно так же, как и с целым деревом. Это позволяет удобно обрабатывать деревья с помощью рекурсивных функций.

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.

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).



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(logN) в среднем (как и бинарный поиск в отсортированном массиве) и, в отличие от отсортированного массива, позволяет производить вставку элемента за O(logN) 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(logN) или определение порядкового номера элемента также за O(logN).

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:

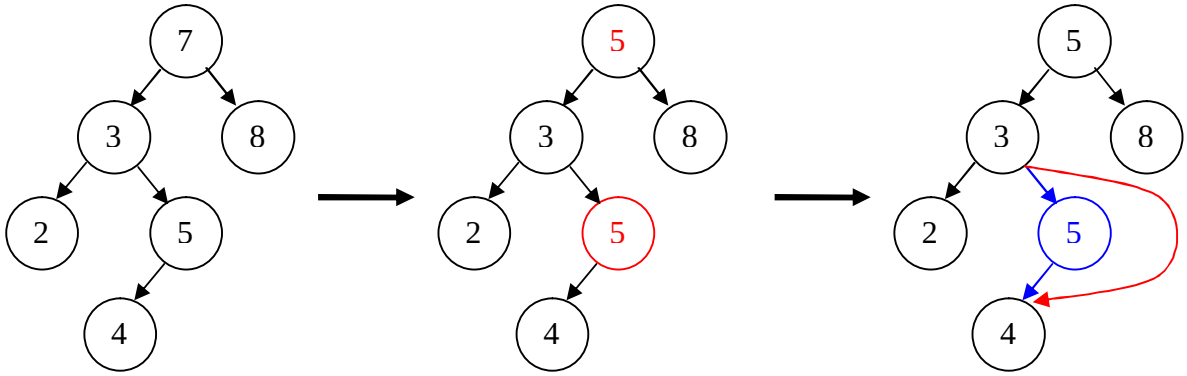
```
node* crnode(int val)
{
     node* nnode = (node*) malloc(sizeof(node));
     nnode->key = val;
     nnode->left = NULL;
     nnode->right = NULL;
     return nnode;
}

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:



Красным помечено то, что мы в данным момент изменяем, синим – элементы подлежащие удалению.

Опишем удаление элемента в виде функции:

```
int rightmost(node *root)
{
     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 прямой обход 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(logN) 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.

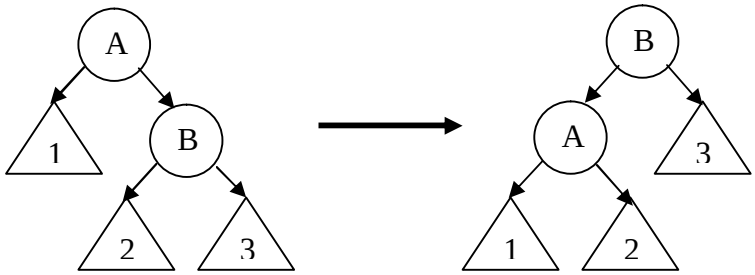
Сбалансированные деревья. АВЛ-дерево.

Сбалансированным деревом называется такое дерево, что его высота пропорциональна logN в любом случае, т.е. H = O(logN). Естественно, при этом требуются дополнительные достаточно сложные операции, которые будут балансировать наше дерево.

Мы рассмотрим алгоритм балансировки, придуманный советскими учеными Адельсоном-Вельским и Ландисом (и не рассмотрим заграничный алгоритм красно-черных деревьев из чувства здорового патриотизма ☺).

АВЛ-дерево обладает следующим свойством: высота его поддеревьев различается не более чем на 1. Для каждого узла можно посчитать дополнительный параметр, называемый балансом, который равен разности высот правого и левого поддеревьев. Тогда бинарное дерево поиска является АВЛ-деревом в том случае, если баланс для каждого узла равен -1, 0 или 1. Высота листа равна 1, а высота любого другого узла равна максимуму из высот его потомков + 1.

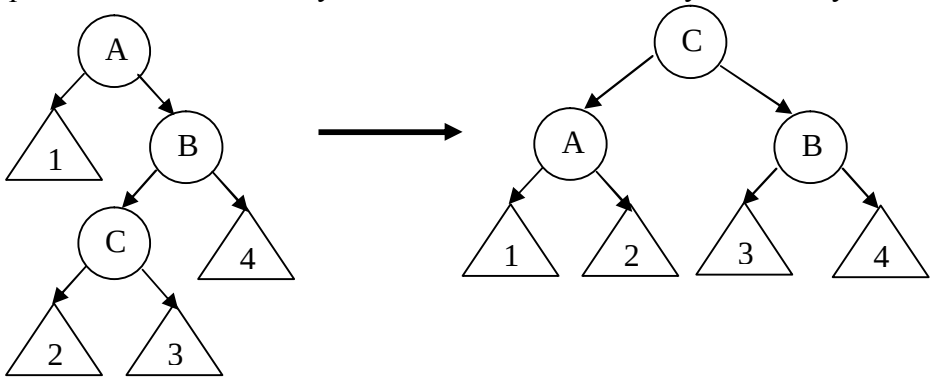
Добавление происходит точно так же, как и в обычном бинарном дереве поиска. После этого мы поднимаемся наверх до корня, пересчитываем высоту и, при необходимости, осуществляем вращение и продолжаем подъем. Точно также при удалении. Если модуль разности высот между сыновьями узла стал больше 1, то необходимо произвести вращение. Вращение бывает четырех типов, всего их два типа: малые и большие. Рассмотрим по одному представителю из каждого типа вращения (другое вращение симметрично). Начнем с простого вращения. Пусть A и B – некоторые узлы, а 1, 2 и 3 – произвольные поддеревья. Пусть в дереве 3 слишком много элементов (нарушился баланс), тогда решением этой проблемы будет следующее преобразование:



Здесь деревья 1 и 2 имеют некоторую высоту h, дерево 3 имеет высоту h+1, следовательно, узел B имеет баланс 1, а узел A – баланс 2 (что и вызывает перестроение дерева). Такая операция называется малым левым вращением. После вращения оба узла будут иметь баланс 0.

Теперь мысленно повернем стрелочку в другую сторону и рассмотрим следующую ситуация. Пусть на правом рисунке дерево 1 имеет высоту h+1, а деревья 2 и 3 – высоту h. Тогда баланс узла A равен -1, а узла B -2. Малое правое вращение (переход в левый рисунок) сделает баланс узлов A и B равным нулю.

Рассмотрим более сложный случай, когда возникает следующая ситуация:



Здесь уже нельзя обойтись малыми вращениями, т.к. они не будут решать проблему дисбаланса.

На рисунке приведен пример для большого правого вращения. Пусть деревья 1, 2 и 4 имеют высоту h, а дерево 3 – высоту h-1. Тогда мы получим баланс равный 2 в узле A (отличие от простого правого вращения состоит в том, что баланс у правого сына узла A отрицательный, т.е. балансы разных знаков). После такого рода вращения мы получаем, что

баланс узлов A и C равен нулю, а баланс узла B равен 1. Несложно проверить, что в случае, если высоту деревьев 1, 3 и 4 были равны h, а высота дерева 2 – h-1, то такое вращение также создает правильное АВЛ-дерево.

Большое левое вращение симметрично, т.е. получается из данного, если поменять местами левого и правого сыновей узла A.

Итак, общий алгоритм такой:

Если у текущего узла баланс равен 2, то:

1) Если у правого сына баланс равен 1, то применяем малое правое вращение.

2) Если у правого сына баланс равен -1, то применяем большое правое вращение.

Если у текущего узла баланс равен -2, то:

1) Если у левого сына баланс равен -1, то применяем малое левое вращение.

2) Если у левого сына баланс равен 1, то применяем большое левое вращение.

Реализация такого алгоритма – дело довольно трудоемкое, занимает много места и отнимает много времени. В рамках лекции мы не будем приводить реализацию этого метода, исходные тексты работы с АВЛ-деревом можно найти в Интернете.

В STL существует реализация красно-черного дерева, которая называется `set`. Красно-черное дерево имеет примерно такую же организацию и его высота также пропорциональна logN. В классе set определены следующие полезные методы: `insert`, `find`, `erase` и другие стандартные для контейнеров STL методы, такие как `count`, `clear` и др. `set` упорядочен, т.е. проход итератором от начала до конца даст упорядоченный массив, также с помощью комбинаций вызовов `find` и прохода можно получить упорядоченные части массива. Более подробно об этих операциях можно прочитать в файле помощи.

Поиск порядковых статистик и определение индекса элемента в дереве поиска.

Кроме ключа, в каждом узле дерева можно хранить дополнительную информацию, например, количество элементов в данном дереве (эта информация легко вычисляется как сумма размеров левого и правого деревьев и единицы). Можно также хранить высоту дерева (как мы уже делали в АВЛ-дереве).

Дополнительная информация позволяет получать новую функциональность. В первую очередь рассмотрим задачу поиска порядковой статистики в бинарном дереве поиска.

Для этого в каждом узле будем хранить количество элементов в поддереве, корнем которого является этот узел. Для листьев этот параметр равен 1, для всех остальных узлов он вычисляется по формуле, описанной выше. Удобнее всего осуществлять пересчет этого параметра на рекурсивном подъеме в функциях добавления и удаления элементов (при добавлении узла он будет иметь размер 1, а для остальных – как сумма сыновей плюс один).

Итак, как же вычислить k-ю порядковую статистику? Пусть у нас для всех узлов в поле size записан размер поддерева. В функцию поиска k-ой порядковой статистики (`find_stat(node *root, int k)`) будем передавать корень дерево и число k – номер статистики.

Если в `root->left->size` равен k, то мы уже нашли искомый элемент и возвращаем `root`. Если `root->left->size` > k, то возвращаем значение функции `find_stat(root->left, k)`, иначе возвращаем `find_stat(root->right, k - root->left->size - 1)`.

Несложно видеть, что сложность этой функции также составляет O(H).

Теперь пусть у нас есть значение ключа, а мы хотим узнать, на какой позиции будет стоять этот элемент в упорядоченном массиве.

Для этого модифицируем функцию поиска элемента. Позиция элемента будет сохранятся в глобальной переменной, которая перед запуском функции должна быть

инициализирована нулем, а в случае перехода на правое поддерево у ней будет прибавляться `root->left->size + 1`.

Дерево максимумов (RMQ).

В этом разделе будем рассматривать структуру данных, которая может выполнять следующие операции: прибавить число val ко всем элементам из интервала [l, r] (число может быть и отрицательным) и выдавать максимальное значение из интервала [l, r]. На обычном массиве обе операции будут выполняться за O(N). Несложно придумать структуру, на которой операция изменения будет занимать O(N), а поиск максимума за O(1) (это дополнительный массив, в котором для каждого элемента хранится сумма с 0 элемента по i-ый). Нас же интересует структура, которая будет уметь выполнять эти операции за O(logN).

Такая структура называется деревом максимумов (Range maximum query). Иногда эту структуру называют деревом минимумов (аббревиатура такая же – RMQ), различие состоит только в замене знака больше на знак меньше.

Для реализации дерева максимумов будем пользоваться бинарным деревом. Каждый узел будет кроме ссылок на двух своих потомков хранить еще несколько дополнительных числовых полей: l и r – левая и правая граница интервала, информация о котором содержится в этом узле; max – максимальное значение, которое встречается на этом интервале (будет выбираться из сыновей), а также поле add, которое будет хранить модификацию для всего этого интервала.

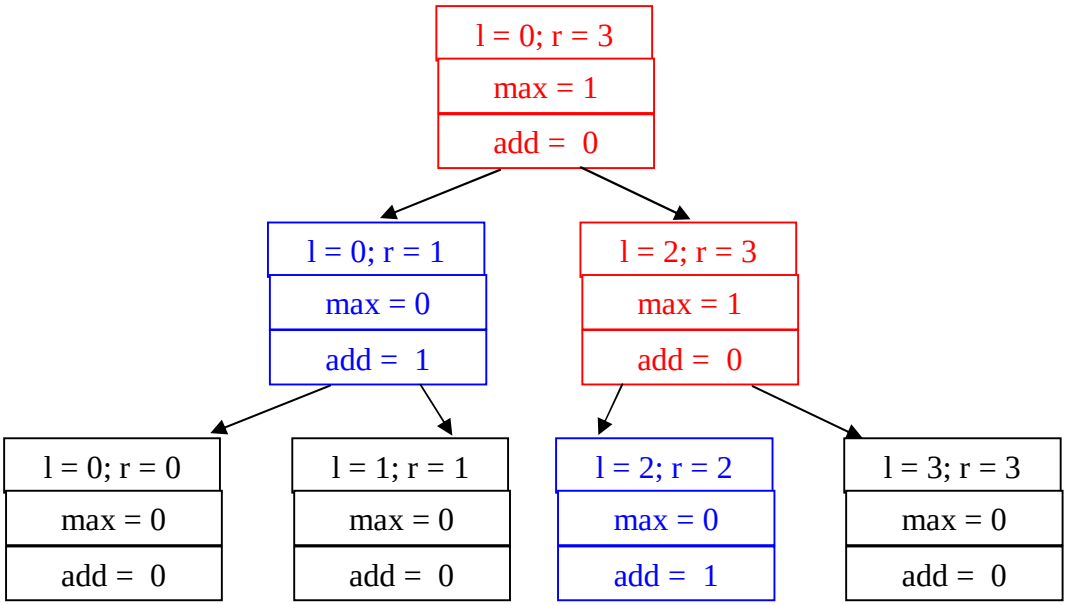
Интервал будет разбиваться на две равные половины, и информация об этих половинах будет храниться в сыновьях данного узла. Если при вызове функции `modify` (прибавление) узел полностью накрывается, то будем прибавлять к его полю `add`, число `value` (то значение, которое прибавляется ко всем элементам). Если узел накрывается интервалом частично, то мы оставляем поле `add` неизменным (оно будет отражать только изменение для всего интервала), вызываем функцию изменения рекурсивно для тех потомков, которые накрываются изменяемым интервалом с тем же значением `value` и с пересчитанным интервалом. В таком случае, на рекурсивном спуске следует пересчитывать значение поля max данного узла, заменяя его на максимальное из значений `max + add` из его сыновей.

Вначале все дерево будет инициализировано нулями (т.е. поля `add` и `max` будут равны нулю для каждого узла).

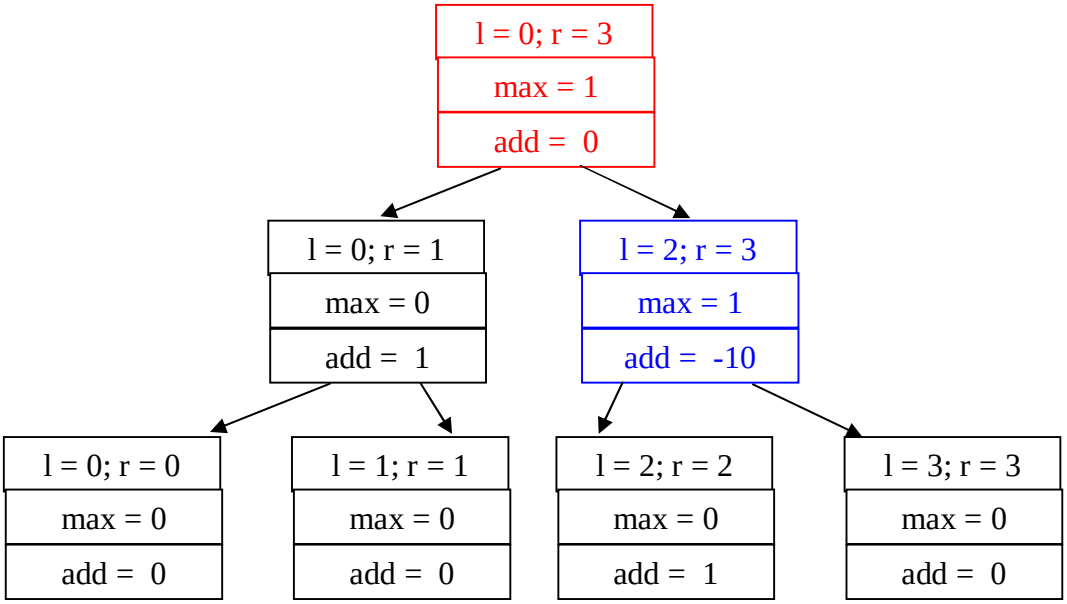
Теперь рассмотрим функцию максимума для интервала. Вначале введем понятие накопленной суммы (`sumadd`). Каждый раз, когда на рекурсивном спуске мы будем проходить через какую-то вершину, то будем прибавлять к сумме значение поля add (таким образом, мы учтем все изменения, которые совершались с данными объектами, которые меняют все элементы, в том числе максимум). Т.е. в каждую функцию в качестве `sumadd` должна передаваться сумма `sumadd` и поля `add` для узла, обрабатываемого вызываемой функцией. При вызове функции поиска должен передаваться указатель на корень дерева, а значение `sumadd` должно быть равно полю `add` для корня.

Если границы текущего интервала совпадают с запросом, то мы возвращаем наверх значение `max + sumadd` (т.е. значение максимального элемента плюс все изменения, которые накладывались на данный интервал). Если же интервал покрывается не полностью, то мы будем вызывать функцию для тех его детей, которые хотя бы частично пересекаются с интервалом и возвращать максимальное из значений, возвращенное этими функциями. Для дерева максимумов мы будем возвращать максимум из возвращенных значений. Для дерева минимумов в функции запроса надо просто возвращать не большее значение, а меньшее.

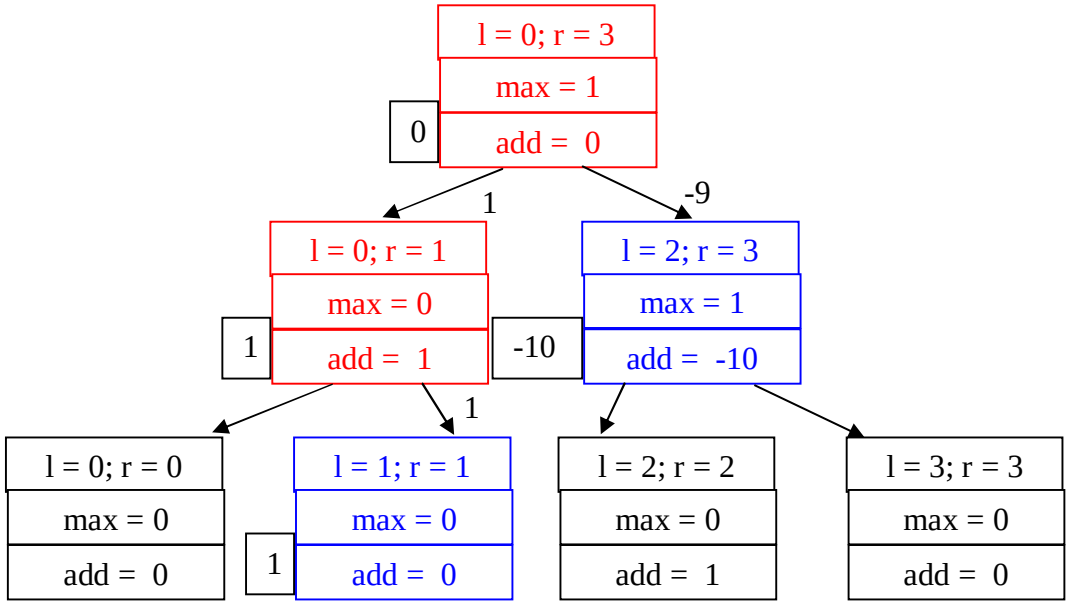
Приведем несколько примеров. Пусть у нас есть дерево для 4 элементов (нумерация от 0 до 7). Красным мы будем помечать узлы, в которых произошел пересчет поля max (накрылись неполностью), а синим – те узлы, в которых пересчитывалось поля add (накрылись полностью). Пусть мы вызвали функцию изменения `modify(tree, 0, 2, 1)` (l = 0, r = 5, value = 1):



Осуществим еще один вызов функции modify(tree, 2, 3, -10):



Теперь вызовем функцию поиска максимума `findmax(tree, 1, 3, 0)`. Синие узлы просто вернули свое значение, красные накрывались частично. Рядом с каждым узлом будем указывать значение `sumadd` для этого узла. Рядом со стрелочкой будем указывать,



что было возвращено из этого узла.

Возвращенное значение будет 1. Приведем текст функций для работы с RMQ:

```
#define infinity 214783648

int min(int a, int b)
{
     return a<b?a:b;
}

int max(int a, int b)
{
     return a>b?a:b;
}

typedef struct _rmq
{
     int max, add, l, r;
     struct _rmq *left, *right;
} rmq;

rmq* creatermq(int l, int r)
{
     rmq* root = (rmq*) malloc(sizeof(rmq));
     root->l = l;
     root->r = r;
     root->max = 0;
     root->add = 0;
     if (l < r)
     {
          root->left = creatermq(l, (l+r)/2);
          root->right = creatermq((l+r)/2+1, r);
     } else {
          root->left = NULL;
          root->right = NULL;
     }
     return root;
}


void modify(rmq *root, int l, int r, int val)
{
     if (root->l == l && root->r == r) root->add += val;
     else
     {
          if (l <= root->left->r) modify(root->left, l, min(r,
root->left->r), val);
          if (r >= root->right->l) modify(root->right, max(l,
root->right->l), r, val);
          root->max = max(root->left->max + root->left->add, root-
>right->max + root->right->add);
     }
}
```

```
int findmax(rmq *root, int l, int r, int sumadd)
{
     if (root->l == l && root->r == r) return sumadd+root->max;
     else
     {
          int res = -infinity;
          if (l <= root->left->r) res = max(findmax(root->left, l,
min(r, root->left->r), sumadd + root->left->add), res);
          if (r >= root->right->l) res = max(findmax(root->right,
max(l, root->right->l), r, sumadd + root->right->add), res);
          return res;
     }
}
```

Описанный пример будет выглядеть так:

```
rmq *tree = creatermq(0, 3);
modify(tree, 0, 2, 1);
modify(tree, 2, 3, -10);
int max13 = findmax(tree, 1, 3, tree->add);
```

На первом шаге мы можем попасть на два неполных отрезка, на всех остальных шагах у нас будет получаться не более одного неполного отрезка. Высота дерева составляет logN, таким образом, мы и получаем сложность O(logN) для обеих операций.

Дерево максимумов можно организовывать и на массиве (по аналогии с реализацией кучи), однако в этом случае требуется особенная внимательность при пересчете индексов.

Рассмотрим некоторые модификации RMQ.

Дерево отрезков.

Дерево отрезков – структура, над которой можно выполнять две операции: `modify(tree, l, r, value)` – полностью аналогична операции над деревом максимумов и операция `findcover(tree, l, r)`, которая возвращает количество ячеек, равных нулю на отрезке [l, r].

Фактически, мы можем добавлять отрезки, убирать их и считать количество непокрытых отрезками точек. При этом следует следить, чтобы мы не убирали отрезки оттуда, где их нет. Необязательно, чтобы отрезки совпадали, главное чтобы не было ячеек, покрытых отрицательным количеством отрезков.

Для реализации дерева отрезков мы будем использовать дерево минимумов, у которого в каждом узле будет добавлено еще одно поле – count, которое будет хранить количество минимальных элементов.

Также как и в обычном дереве минимумов у нас должна быть функция, инициализирующая все нулями. В функцию добавления после пересчета минимума необходимо вставить строки, пересчитывающие количество минимальных элементов. А именно:

```
root->count = 0;
if (root->min == root->left->min + root->left->add)
     root->count += root->left->count;
if (root->min == root->right->min + root->right->add)
     root->count += root->right->count;
```

В функции подсчета изменений еще меньше. В случае если отрезок покрыт полностью, надо проверить, что минимум равен 0, и вернуть в таком случае count, иначе 0. Если же отрезок покрыт не полностью, то следует просуммировать значения, возвращенные рекурсивными вызовами для предков.

Дерево сумм (RSQ).

Рассмотрим еще одну модификацию RMQ, которая будет возвращать сумму всех элементов на отрезке. Суть структуры не меняется, однако вместо поля max должно присутствовать поле sum, для которого будут иные операции пересчета.

Сначала рассмотрим изменения в операции `modify`. Если отрезок накрывается полностью, то никаких изменений вносить не надо. В случае же если он накрывается не полностью, необходимы изменения. Если для пересчета максимумов нам было достаточно просто получить значения из сыновей, то для суммы необходимо также знать количество элементов запроса в этих отрезках. Это количество легко считается, как разность правой и левой границы интервала плюс единица. После этого мы просто заменяем поле sum данного узла на сумму полей sum для обоих сыновей и сумму произведений сыновних полей `add` на количество элементов в каждом сыне.

В операции `findsum` изменения по сравнению с `findmax` совсем незначительны – вместо выбора максимального элемента мы считаем их сумму.

Обратите внимание, что ни на каком из отрезков значение суммы не должно превосходить максимальной вместительности используемого типа переменных!

Система непересекающихся множеств.

В олимпиадных задачах довольно часто требуется разбить набор объектов на непересекающиеся множества (т.е. каждый объект может лежать только в одном множестве, но в одном множестве может находиться несколько объектов).

Пусть объекты пронумерованы от 0 до N-1. В качестве идентификатора множества будем использовать также числа от 0 до N-1. При инициализации, обычно, все множества состоят из одного элемента, т.е. объект с номером i лежит во множестве с номером i.

Для системы непересекающихся множеств определены две операции: fset(x), возвращающая номер множества, в котором лежит элемент x и операция sunion(x, y), объединяющая множества, содержащие элементы x и y в одно. Первая операция используется, обычно, для проверки того, лежат ли два элемента в одном множестве или в разных.

Эффективность различных структур будем оценивать как количество операций, необходимое для объединения N множеств в одно в наихудшем для данной структуры случае.

Самым простым способом реализации является одномерный массив, где индекс задает номер объекта, а значение – номер множества, в котором этот объект находится. Проверка будет просто возвращать значение из запрошенной ячейки, а объединение должно проходить по всему массиву и заменять все числа x на y (или наоборот). В худшем случае каждый раз мы будем добавлять по одному элементу, таким образом, нам потребуется N проходов по массиву и сложность составит $O(N^2)$.

Рассмотрим более эффективную реализацию, где кроме массива будет храниться также список элементов множества (для списка мы будем хранить указатели на начало и конец). Этот способ требует O(N) дополнительной памяти. При проверки мы будем также возвращать значение из массива, а при модификации проходить по одному из списков и менять значения в массиве, а потом прикреплять этот список к концу другого. Если мы будем делать это бездумно, то сложность также составит $O(N^2)$ – это случай, когда каждый раз длинный список будет прикрепляться к концу списка длины 1. Если ввести для каждого списка такое поле, как длина списка (оно легко пересчитывается при объединении) и приписывать каждый раз более короткий список, то для последовательного объединения всех списков длины 1 сложность получится O(N). Однако в случае, если каждый раз длины списков будут равны (худший для нас случай), мы получаем сложность O(NlogN). На каждом шаге будут объединяться все пары списков равной длины, каждый раз длина списка будет увеличиваться вдвое, следовательно, общее количество шагов будет составлять logN.

Рассмотрим также реализацию на стягивающихся корневых деревьях. Для каждого множества выберем элемент, которого назовем «представителем». Представитель указывает сам на себя и сначала все элементы указывают сами на себя. Для каждого дерева введем такое понятие, как высота, которое сродни длине списка и более «низкое» дерево следует прикреплять к более высокому, а в случае, если высоты были равны, то высота результирующего дерева увеличится на единицу. Требования к дополнительной памяти составят O(N).

Функция поиска должна идти наверх, пока не дойдет до представителя, а функция объединения вызывает две функции поиска, а затем уже объединяет деревья.

Казалось бы, такой способ не дает никаких особых преимуществ перед списками, однако можно модифицировать функцию поиска так, чтобы при проходе по всем элементам пути до представителя она, получив указатель на представителя, для всех элементов на пути устанавливала указатель непосредственно на представителя. Это требует еще O(logN) дополнительной памяти (эта оценка сильно завышена). Приведем реализацию системы непересекающихся множеств:

```
typedef struct syst_
{
     int rank, p;
} syst;

void init(syst *a, int n)
{
     int i;
     for (i = 0; i<n; i++)
     {
          a[i].p = i;
          a[i].rank = 0;
     }
}

int fset(syst *a, int x)
{
     if (x != a[x].p)
          a[x].p = fset(a, a[x].p);
     return a[x].p;
}

void sunion(syst *a, int x, int y)
{
     if (a[x].rank < a[y].rank)
          a[x].p = a[y].p;
     else
     {
          a[y].p = a[x].p;
          if (a[x].rank == a[y].rank) a[x].rank++;
     }
}
```

Сложность объединения всех множеств в одно практически линейна. Оставим без доказательства этот факт и точную оценку, однако интуитивно понятно, почему операция `fset` будет выполняться с каждым разом все быстрее.