Деревья: бинарные, декартовы, сбалансированные

Наумов Д.А., доц. каф. КТ

Алгоритмы и структуры данных, 2021

Содержание лекции

1. Абстракция *отображение*
2. Бинарные деревья поиска
3. Дерево отрезков

Отображение

абстракция, устанавливающая направленное соответствие между двумя множествами (множеством ключей и множеством данных) и реализующая над ними определённые операции.

Рис.: Пример отображения: каждому городу соответствует численность его населения

Абстракция *отображение* есть аналог дискретной функции:

Функция

есть отображение множества D на множество E.

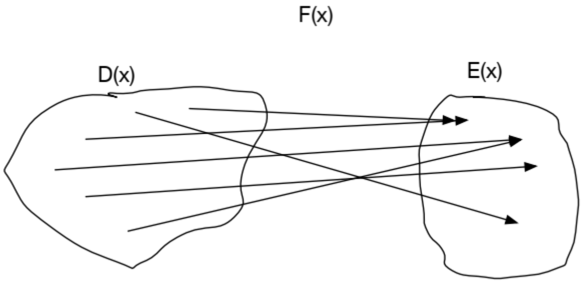


Рис.: Отображение множества D на множество E

Наша цель – реализовать некий *словарь*, в котором мы можем добавлять, искать и удалять *словарные статьи*.

C++, наряду с другими современными языками, предоставляет возможность использовать отображения как обобщение понятия массив с помощью синтаксиса индексации:

```
map m<string,int>;
m["Шанхай"] = 24150000;
m["Карачи"] = 23500000;
m["Пекин"] = 21150000;
m["Дели"] = 17830000;
...
int BeijingPopulation = m["Пекин"];
...
for (auto x: m) {
  printf("Population of %s is %d\n", x.first, x.second);
}
```

Интерфейс абстракции *отображение*

Интерфейс абстракции отображение есть частный случай абстракции хранилища CRUD (create, read, update, delete):

- insert(key, value) – добавить элемент с ключом key и значением value.
- Item find(key) – найти элемент с ключом key и вернуть его.
- erase(key) – удалить элемент с ключом key.
- walk – получить все ключи (или все пары ключ/значение) в каком-либо порядке.

В дальнейшем под термином ключ мы понимаем пару *ключ+значение*, в которой определена операция сравнения по ключу.

Абстракцию *множество* можно рассматривать как частный случай абстракции отображение.

- множество ключей с прикрепленными данными;
- отображение набора ключей на логические переменные.

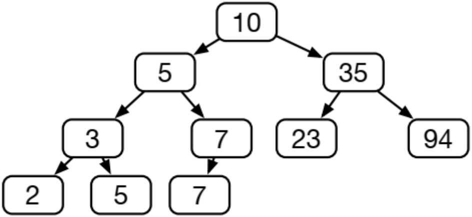
Бинарным деревом поиска (Binary SearchTree, BST)

называется бинарное дерево, в котором все узлы, находящиеся справа от родителя, имеют значения, не меньшие значения в родительском узле, а слева — не большие этого значения.

Задача.

- На вход подаётся последовательность чисел.
- Выходом должно быть двоичное дерево поиска.

{10, 5, 35, 7, 3, 23, 94, 2, 5, 7}



Алгоритм поиска элемента в BST, содержащего ключ X

1. Делаем текущий узел корневым.
2. Переходим в текущий узел C.
3. Если X = C:Key, то алгоритм завершён.
4. Если X > C:Key и C имеет потомка справа, то делаем текущим узлом потомка справа. Переходим к п. 2.
5. Если X < C:Key и C имеет потомка слева, то делаем текущим узлом потомка слева. Переходим к п. 2.
6. Ключ не найден. Конец алгоритма.

Алгоритм прост и эффективен, и его сложность определяется только наибольшей высотой дер

Алгоритм *наивного* построения BST

1. Делаем текущий узел корневым.
2. Переходим в текущий узел C.
3. Если $X = C{:}Key$, то алгоритм завершён, вставка невозможна.
4. Если $X > C{:}Key$ и C имеет потомка справа, то делаем текущим узлом потомка справа. Переходим к п. 2.
5. Если $X > C{:}Key$ и C потомка справа не имеет, то создаём правого потомка с ключом X и завершаем алгоритм.
6. Если $X < C{:}Key$ и C имеет потомка слева, то делаем текущим узлом потомка слева. Переходим к п. 2.
7. Если $X < C{:}Key$ и C потомка слева не имеет, то создаём левого потомка с ключом X и завершаем алгоритм.

Алгоритм *наивного* построения BST

- первый поступивший элемент последовательности формирует корневой узел дерева – и каждый последующий элемент занимает соответствующее место после неудачного поиска (если поиск удачен, то дерево уже содержит ключ).
- Неудачный поиск всегда заканчивается на каком-то узле, у которого в нужном направлении нет потомка.

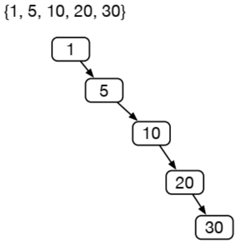


Рис.: Пример вырожденного BST

Случайное бинарное дерево T размера n

дерево, получающееся из пустого бинарного дерева поиска после добавления в него n узлов с различными ключами в случайном порядке, при условии, что все n! возможных последовательностей добавления равновероятны.

Средняя глубина случайного бинарного дерева: $d(N) = 2ln(N)$.

Средние времена выполнения операций вставки, удаления и поиска в случайном бинарном дереве есть $\Theta(log_2(N))$.

Поиск минимального/максимального ключа

Так как все потомки слева от узла имеют значения ключей, меньшие значения ключа самого узла, то наименьший элемент всегда находится в самом низу левого поддерева.

Аналогично, наибольший элемент всегда находится в самом низу правого поддерева.

```
tree * minNode(tree *t) {
        if (t == nullptr) return nullptr;
        while (t->left != nullptr) {
                t = t->left;
        }
        return t;
}
```

Методы поиска и вставки, описанные ранее, достаточно просты.

Удаление узла

Процедура удаления – сложнее, требуется рассмотреть три случая:

1. У удаляемого узла нет потомков – достаточно удалить этот узел у родителя.
2. Имеется один потомок – переставляем узел у родителя на потомка.
3. Имеется два потомка – находим самый левый лист в правом поддереве и им замещаем удаляемый.

Первый случай. Удаление терминального узла 6.

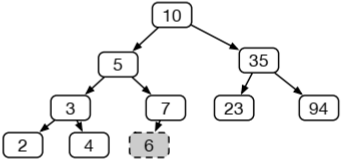


Рис.: До удаление

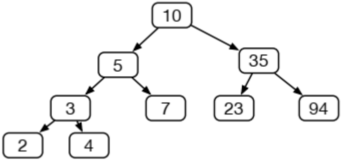


Рис.: После удаления

Второй случай. Удаление узла 7, имеющего одного потомка.

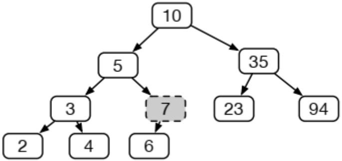


Рис.: До удаление

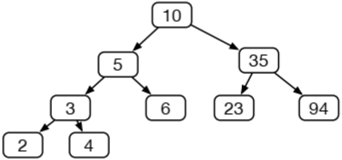


Рис.: После удаления. Единственный потомок занял место удаляемого узла

Третий случай. Удаление узла 10, имеющего двух потомков.

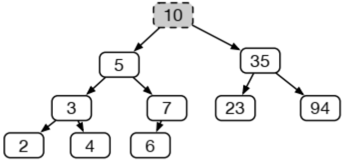


Рис.: До удаление

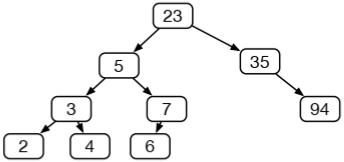


Рис.: После удаления. Самый левый потомок занял место удаляемого

Структура хранилища	вставка	удаление	поиск
Бинарное дерево поиска (наихудшее)	$O(N)$	$O(N)$	$O(N)$
Бинарное дерево поиска (среднее)	$O(\log N)$	$O(\log N)$	$O(\log N)$

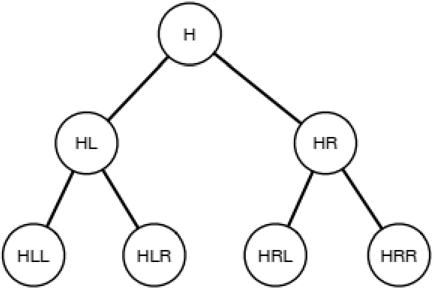
Рис.: Сложность операций для бинарного дерева поиска

- Сложность всех алгоритмов в бинарных деревьях поиска определяется средневзвешенной глубиной.
- Операции вставки/удаления могут привести к дисбалансу и ухудшению средних показателей.
- Для борьбы с дисбалансом применяют *рандомизацию* и *балансировку*.

- Создание BST из упорядоченной последовательности чревато крайне несбалансированным деревом. Возникает
- Вопрос: а что будет, если вставлять новые элементы не в терминальные узлы дерева, а заменять ими корень?
- Последствия таковы: если вставляемый элемент больше корня, то старый корень сделаем левым поддеревом, а его правое поддерево – нашим правым поддеревом.
- Аналогично рассуждаем для случая, когда вставляемый элемент меньше корня. К сожалению, и в том, и в другом случае может нарушиться упорядоченность.

Чтобы нарушений не происходило, требуется сохранять инвариант упорядоченности. Для этого введём понятие **поворота**, не изменяющего упорядоченные свойства дерева, но меняющего его структуру, то есть высоту поддеревьев.

- H есть сокращение от слова Head,
- L – от Left
- R – от Right.



После поворота направо дерево будет выглядеть так:

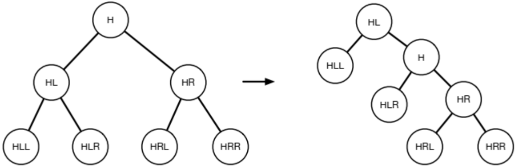


Рис.: BST: дерево после правого поворота

```
void rotateRight(node* &head) {
        node *temp = head->left;
        head->left = temp->right;
        temp->right = head;
        head = temp;
}
```

После поворота налево дерево будет выглядеть так:

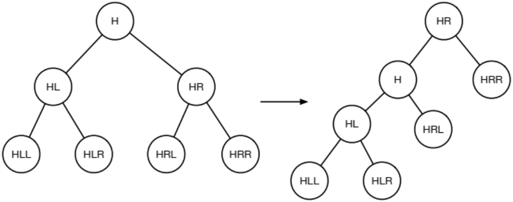


Рис.: BST: дерево после левого поворота

```
void rotateLeft(node* &head) {
        node *temp = head->right;
        head->right = temp->left;
        temp->left = head;
        head = temp;
}
```

Функция вставки нового узла в корень дерева:

```
void insert(node* &head, item x) {
        if (head == nullptr) {
                head = new node(x);
                return;
        }
        if (x.key < head->item->key) {
                insert(head->left, x);
                rotateRight(head);
        } else {
                insert(head->right, x);
                rotateLeft(head);
        }
}
```

Рандомизированное дерево

- Мы знаем, что вставка узла в терминальный узел приводит к вырожденному дереву для упорядоченной последовательности. Впрочем, если попробуем эту же последовательность вставлять в корень, то получим подобный результат.
- Будем случайным образом выбирать, куда мы собираемся вставить очередной ключ.
- Операция вставки в корень дерева значительно сложнее операции вставки в терминальный узел, так как она требует $O(\log N)$ операций поворота.
- Для дерева, содержащего N вершин, вставка очередного узла в корень производится с вероятностью $1/(N+1)$, в противном случае вставку оставляем обыкновенной, то есть в узел.
- В этом случае свойства любого дерева будут соответствовать свойствам случайного дерева.

Сбалансированные деревья поиска

- Добиться хороших оценок времени исполнения операций с BST можно и без использования случайности.
- Для этого требуется при операциях избегать тех преобразований структуры деревьев, которые приводят к их вырождению.
- Это можно сделать, измеряя высоты поддеревьев, и, делая повороты при необходимых условиях, балансировать деревья.

Сбалансированные деревья поиска

Поставим более жёсткую задачу: реализовать операции с деревьями, имеющие время в худшем $\Theta(logN)$.

- Для этого требуется сохранять высоту дерева H в определённых границах.
- Чтобы сравнить различные стратегии балансировки, будем сравнивать высоту H, выраженную формулой $H < A \cdot log_2(N) + B$; где A и B – некоторые фиксированные константы.
- Если обозначить через H_{ideal} высоту идеально сбалансированного дерева, а через H_{algo} – наибольшую из возможных высот при реализации выбранной стратегии балансировки, то в первую очередь нас будет интересовать константа $A = H_{algo}/H_{ideal}$ – отношение этих высот.

Сбалансированное дерево №1. Идеально сбалансированное дерево

Для любого узла количество узлов в левом и правом поддереве N_l, N_r отличается не более, чем на 1.

$$N_r \leq N_l + 1; N_l \leq N_r + 1$$

Если N – нечётно и равно $2M + 1$, тогда левое и правое поддеревья должны содержать ровно по M вершин.

$$Hideal(2M + 1) = 1 + Hideal(M)$$

Если N – чётно и равно $2M$. Тогда

$$H_{ideal}(2M) = 1 + max(H_{ideal}(M - 1); H_{ideal}(M))$$

Так как $H_{ideal}(M)$ – неубывающая функция, то

$$H_{ideal}(2M) = 1 + H_{ideal}(M); H_{ideal}(N) \leq log_2(N)$$

Это означает, что ключевой коэффициент A равен 1.

Сбалансированное дерево №2. Примерно сбалансированное дерево

Для любого узла количество подузлов в левом и правом поддеревьях удовлетворяют условиям:

$$N_r \leq 2N_l + 1; N_l \leq 2N_r + 1$$

Максимальная высота сбалансированного дерева со свойством 2:

$$H(N) > log_{3/2}N + 1 \approx 1.71 log_2(N) + 1$$

Сбалансированное дерево №3. Примерно сбалансированное АВЛ-дерево

Для любого узла количество подузлов в левом и правом поддеревьях удовлетворяют условиям:

$$N_r \leq N_l + 1; N_l \leq N_r + 1$$

Название, *АВЛ-деревья* – взято из первых буквы фамилий их изобретателей: Георгия Максимовича Адельсона-Вельского и Евгения Михайловича Ландиса.
Максимальная высота сбалансированного дерева со свойством 2:

$$H(N) \approx log_{\Phi}(N - 1) + 1 \approx 1.44 log_2(N) + 1$$

Сбалансированное дерево №4. Красно-чёрное дерево

Красно-чёрное дерево, RBT

это сбалансированное бинарное дерево поиска, которое в качестве критерия балансировки использует цвет узлов.

1. Вершины разделены на красные и чёрные.
2. Каждая вершина хранит поля ключ и значение.
3. Каждая вершина имеет указатель left, right, parent.
4. Отсутствующие указатели помечаются указателями на фиктивный узел nil.
5. Каждый лист nil – чёрный.
6. Если вершина – красная, то её потомки – чёрные.
7. Все пути от корня root к листьям содержат одинаковое число чёрных вершин. Это число называется чёрной высотой дерева, black height, bh(root).

Сбалансированное дерево №4. Красно-чёрное дерево

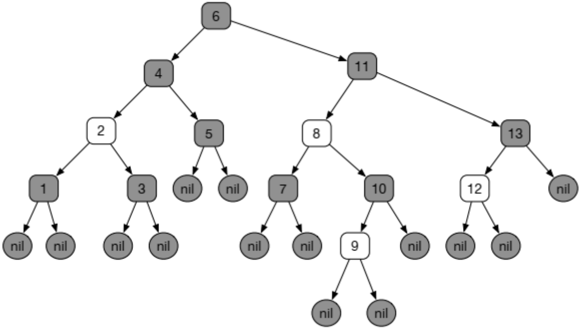


Рис.: Красно-чёрное дерево, пример. Красные узлы на рисунке изображены отсутствием заливки

$$H_{rb} \leq 2 \cdot log_2(N) + 1$$

Сбалансированное дерево №4. Красно-чёрное дерево

	RB-tree	AVL-tree
Средняя высота	до 1.38H	H
Поиск/вставка	до 1.38t	t
Поворотов при вставке	до 2	до 1
Поворотов при удалении	до 3	до $\log N$
Дополнительная память	1 бит	1 счётчик

Пусть нам надо решить задачи:

- многократное нахождение максимального значения на отрезках массива;
- многократное нахождение суммы на отрезке массива.

Мы умеем совершать эти действия за время $O(N)$, где $N = R - L + 1$. При определённой подготовке можно сократить время на каждую из операций до $O(logN)$.

Для примера возьмём массив:

Рис.: Массив – основа дерева отрезков

Попарно соединим соседние вершины, поместив в узел-родитель значение функции max(left,right). Родитель каждого узла называется *доминирующим узлом*.

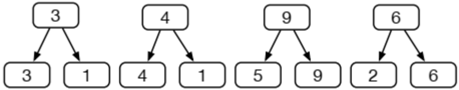


Рис.: Дерево отрезков: построение. Добавление доминирующих узлов первого уровня

Проделаем эту же операцию с получившимися узлами:

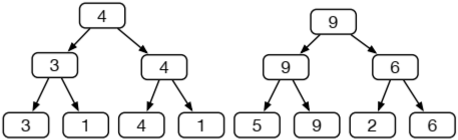
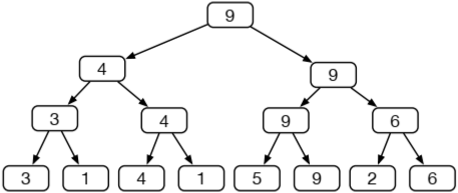


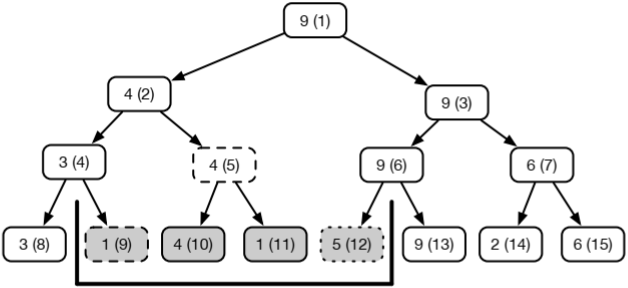
Рис.: Дерево отрезков: построение. Добавление доминирующих узлов второго уровня

Итог:



После построения такого дерева задачу нахождения максимума на отрезке можно решить за $O(logN)$.

Дерево отрезков: поиск максимума на отрезке



- Идея вычисления проста – если на отрезке присутствует пара элементов, имеющая общий доминирующий узел, то результатом вычисления функции для этой пары будет предвычисленное значение доминирующего узла.
- Элементы, не входящие в такие пары, могут находиться либо строго на левом конце отрезка, либо строго на правом. Их придётся явно учесть отдельно – и это мы сделаем, когда будем реализовывать соответствующие операции.

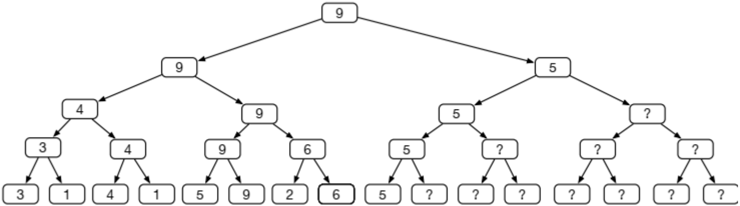
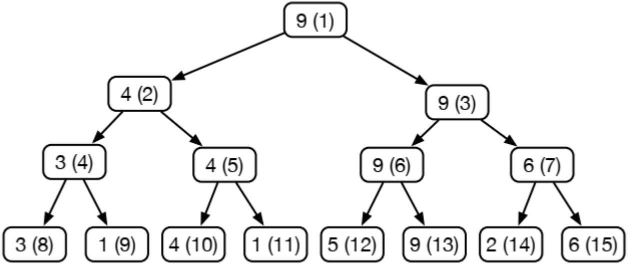


Рис.: Дерево отрезков: дополнение числа элементов до степени двойки

Что должно находиться в узлах, отмеченными знаками вопроса?

- Так как все значения в доминирующих узлах вычисляются с помощью функции $P = \max(L; R)$, то то же самое должно происходить с элементом '?'.
- Это означает, что операция $\max(L; ?)$ должна возвращать L. То есть элемент '?' есть ∞.
- Для функции max ∞ есть нейтральный элемент.

При создании дерева отрезков (Create(size)) создаётся бинарная куча, инициализированная нейтральными элементами. C есть номер первого элемента в нижнем ряду, представляющем заданный набор, над которым будут далее производиться операции. $C = min(2^k) : C > size$



Каждая операция вставки элемента по сути заменяет нейтральный элемент, который находился в месте вставки, на нужное значение.

```
Insert/Replace(i, val):
        body[i+C]=val;
        propagate(i);
```

Операция *propagate(i)* рекурсивно обновляет все доминирующие узлы.

Операция нахождения значения функции на интервале [left; right] тоже рекурсивна:

```
Func(left,right):
        Res = E;
        if (left % 2 == 1)
                Op(Res, body[left++]);
        if (right % 2 == 0)
                Op(Res, body[right--]);
        if (right > left)
                Op(Res, Func(left/2, right/2));
```

Операция создания дерева отрезков требует в худшем случае до $4N$ памяти, а остальные операции имеют логарифмическую сложность.

- Требуемая память: $min = O(2N)..max = O(4N)$
- Операция **Insert/Replace**: $O(logN)$
- Операция **Func** на любом подотрезке: $O(logN)$