Основные алгоритмы, комментарии по шестому листку (Структуры данных)

Шибаев Иннокентий

March 16, 2021

1 По поводу теории

1.1 О сортировках в этом курсе

У нас было уже достаточно много разных сортировок, в Таблице 1 ниже приведена основная информация о них. В начале надо сказать несколько вещей по поводу них

- Во-первых, алгоритмы сортировок изученные нами можно разделить на две важные группы, в зависимости от того основаны ли они на сравнениях. Первые 5 сортировок в таблице – это примеры сортировок сравнениями, последние две основаны на других идеях, к примеру в сортировке подсчетом делается дополнительное предположение о том что в массиве есть не более k различных значений, а в *RadixSort* предполагается что длина сравниваемых элементов (строк, или чисел) не превышает k.

- Во-вторых, затраты по памяти в таблице указаны в смысле дополнительной памяти помимо той что нужна для хранения массивов (а это всегда $O(n)$).

- В-третьих, на одном из прошлых семинаров была задача (ее разбор есть в 4 листке) следующего вида «Докажите, что любую сортировку сравнениями можно сделать устойчивой сохранив асимптотическое время работы». Поэтому «Нет» в последнем столбце, это скорее указание на то что данные сортировки не являются устойчивыми без проведения дополнительных манипуляций, а конкретно – без расширения ключа по которому будет проводится сортировка (имеется в виду добавление к изначальным параметрам по которым будет производится сравнение еще и номера в исходном массиве, по которому будет производится сравнения в случае прочих равных). Но это требует $O(n)$ дополнительной памяти, и в таком случае в смысле асимптотик эти алгоритмы уже ничем не отличаются от *MergeSort*.

  В случае *CountingSort* фактически тоже требуются дополнительные действия (т.к. если просто посчитать частоты элементов относительный порядок будет утрачен), но это не сортировка сравнениями, поэтому для нее в таблице это расписано отдельно.

- В-четвертых, для быстрой сортировки оценки сложности сильно зависят от метода выбора опорного элемента. Ее можно реализовать так чтобы у нее была сложность по времени $O(n \log n)$ и $O(\log n)$ по памяти. Кстати, сложность по памяти берется из того что нам нужно хранить стек в рекурсивных вызовах, при итеративной реализации тоже будет получаться $O(\log n)$ дополнительной памяти (почему?).

Таблица 1: Общая информация о сортировках в этом курсе

Сортировка	Сложность по времени	Сложность по памяти	Идея	Стабильность
BubbleSort, Сортировка пузырьком	$O(n^2)$	$O(1)$	Идем по массиву и меняем местами пары соседних элементов если их порядок неправильный. Повторяем n раз.	Да, достаточно не менять местами равные элементы
InsertionSort, Сортировка вставками	$O(n^2)$	$O(1)$	Вставляем элементы в подходящее место среди уже упорядоченных	Да, если размещать новый элемент **после** равных ему
MergeSort, Сортировка слиянием	$O(n \log n)$	$O(n)$	Сортируем левую и правую половину массива, сливаем за $O(n)$	Да, при слиянии равных в начале записываем элементы справа
QuickSort, Быстрая сортировка	$O(n \log n)$ в среднем, при этом в худшем случае $O(n^2)$	$O(\log n)$ (в худшем случае $O(n)$)	Применяем *Partition* (делим массив относительно на две части – меньшая и большая опорного элемента, применяем сортировку к каждой)	Нет[1]
HeapSort, Пирамидальная сортировка	$O(n \log n)$	$O(1)$	Превращаем массив в *MaxHeap*, меняем местами корень и последний элемент в куче, обновляем, повторяем n раз	Нет[1]
CountingSort, Сортировка подсчетом	$O(n+k)$ где k – кол-во различных элементов	$O(k)$ если просто считать, $O(n+k)$ если нужна стабильность	Считаем частоту вхождений элементов. Если требуется стабильность то либо сохраняем списки, либо создаем массив с частотам вхождений элементов меньших либо равных данному (с лекции)	Да, если запоминать последовательность подсчитываемых элементов
RadixSort, Поразрядная сортировка	$O\left(w \cdot (n+k)\right)$ где w – длина ключей[2]	$O(n+k)$	Сортируем элементы по разрядам, начиная с младшего, используя устойчивую сортировку	Да

Нельзя в общем случае сказать какая сортировка лучше сработает для данного массива. К примеру массив к которому добавляется новый элемент не нужно сортировать за $O(n \log n)$ – достаточно просто вставить новый элемент на свое место за $O(n)$ (или за $O(\log n)$ если мы используем структуру вроде сбалансированного бинарного дерева поиска). Некоторые сортировки комбинируются чтобы достигать каких-либо дополнительных свойств (к примеру в stdlib вроде используется комбинация *QuickSort*, *HeapSort* и *InsertionSort* (т.н. *IntroSort*), а в *Python* в качестве стандартной используется *TimSort* который является комбинацией *MergeSort* и *InsertionSort*.

[1] Если нельзя использовать дополнительную память. Можно расширить ключ добавив номер в исходном массиве, но это уже $O(n)$ доп. памяти

[2] По поводу *RadixSort*. Тут есть уже три различных сущности, n – число сортируемых элементов, k – кол-во различных значений которые могут быть в каждом разряде (к примеру для десятичной системы это $\{0, 1, \ldots, 9\}$ – всего 10 значений, а при сортировке строк из букв латинского алфавита будет уже $k = 26$ или $k = 52$ если надо хранить регистр). Эти два параметра были еще в *CountingSort*,

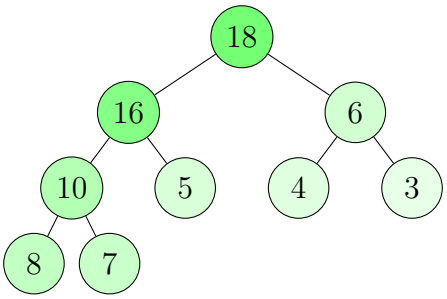
1.2 Структура данных *Heap* (куча, пирамида)

Перейдем к структурам данных. Первая структура которую мы рассмотрим, *Heap*, обладает следующими свойствами

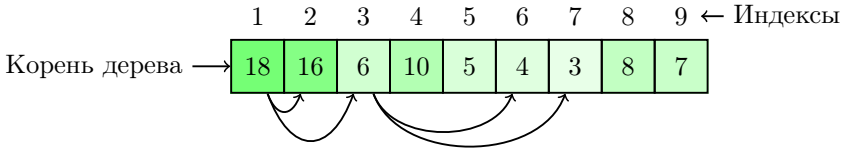
- Это бинарное дерево, у которого глубина всех листьев отличается не более чем на один
- Последний слой заполняется слева направо
- Для каждого узла выполнено свойство максимальности – значение в узле больше либо равно значениям в дочерних вершинах.

Из этих двух свойств следует, к примеру, что в корневой вершине лежит наибольший элемент. В одной из дочерних вершин корневой также лежит второй максимум, про третий так уже сказать нельзя.

Рассмотрим, к примеру, кучу с семинара

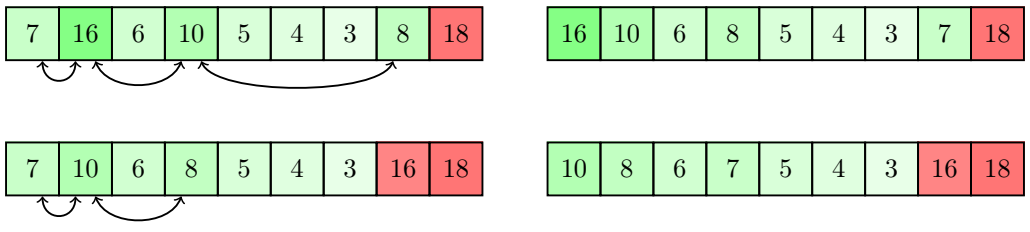


ее можно удобно представить в виде массива



в котором для элемента с индексом i индексы дочерних вершин вычисляются как $2 \cdot i$ и $2 \cdot i + 1$ (т.е. к примеру для 6 стоящей на позиции $i = 3$ дочерние вершины со значениями 4 и 3 стоят на позициях $6 = 2 \cdot i$ и $7 = 2 \cdot i + 1$).

Эта структура позволяет сортировать массив. Для этого надо на каждом шаге доставать из корня максимум и менять его с концом массива в котором сейчас лежит куча. После ее обновления этот процесс повторяется.



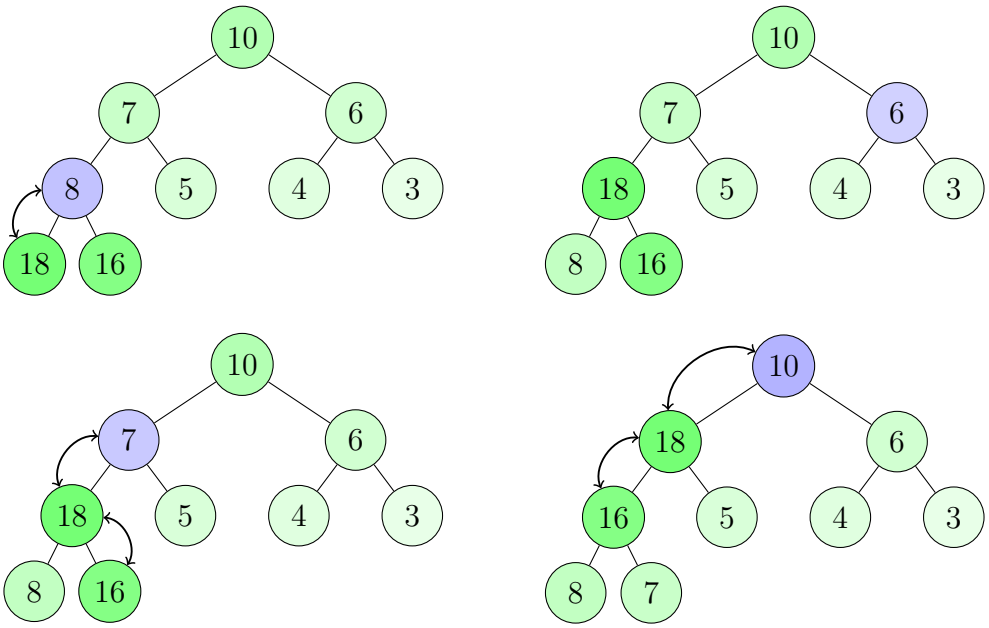
И так далее, таким образом массив заполняется с конца максимальными значениями неотсортированной части массива. В таком смысле у него есть общие

но добавляется еще один – w, максимальная длина ключей (т.е. число разрядов). И таким образом, применяя тот же *CountingSort* (сложность которого $O(n+k)$) для каждого из w разрядов мы получим $O(w(n+k))$

черты с алгоритмом сортировки пузырьком (просто сравниваются не ближайшие элементы, а по степеням двойки причем в подходящем для этого множестве).

Вторая вещь которую здесь стоит обсудить это то как вообще можно построить данную структуру данных. Идея в рекурсивном ее построении: если оба поддерева у родительской вершины уже являются кучами, то если значение в этой вершине больше чем в дочерних то это уже куча. Иначе надо выбрать максимум из дочерних вершин, поменять его с родительской вершиной, после чего произвести те же манипуляции с выбранным поддеревом (т.к. после обновления корня в нем оно могло перестать быть кучей).

Поэтому алгоритм идет начиная с вершин которым соответствуют самые маленькие кучи (из 2-3 элементов), и постепенно обновляет все более сложные. Процесс представлен ниже (синим выделена вершина в которой алгоритм сейчас строит кучу)



Осталось только обсудить сложность этого процесса. Алгоритм начинает с вершин у которых наименьшее кол-во дочерних (если куча хранится в массиве то это соответствует тому что он идет от вершин с индексами $\lfloor \frac{n}{2} \rfloor$), и для каждой вершины ищет максимум значений между значением в самой вершине и дочерними. Если наибольшее значение находится в дочерней вершине то происходит обмен, после чего рекурсивно обновляется поддерево соответствующее этой вершине. Обозначим число действий для одного обновления за C, тогда если мы вызываем обновление от вершины находящейся на уровне k (от листьев) на ней произойдет не более Ck действий.

Теперь, заметим что в дереве $\lfloor \frac{n}{2} \rfloor$ вершин имеют высоту большую либо равную 1, $\lfloor \frac{n}{4} \rfloor$ вершин – высоту большую либо равную 2 и т.д. Таким образом получаем, что алгоритм совершит

$$C \cdot \left(\left\lfloor \frac{n}{2} \right\rfloor - \left\lfloor \frac{n}{4} \right\rfloor \right) + 2C \cdot \left(\left\lfloor \frac{n}{4} \right\rfloor - \left\lfloor \frac{n}{8} \right\rfloor \right) + 3C \cdot \left(\left\lfloor \frac{n}{8} \right\rfloor - \left\lfloor \frac{n}{16} \right\rfloor \right) + \dots$$

операций, и, переупорядочив это, получим

$$C \cdot \left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lfloor \frac{n}{4} \right\rfloor + \left\lfloor \frac{n}{8} \right\rfloor + \dots \right) \leqslant Cn$$

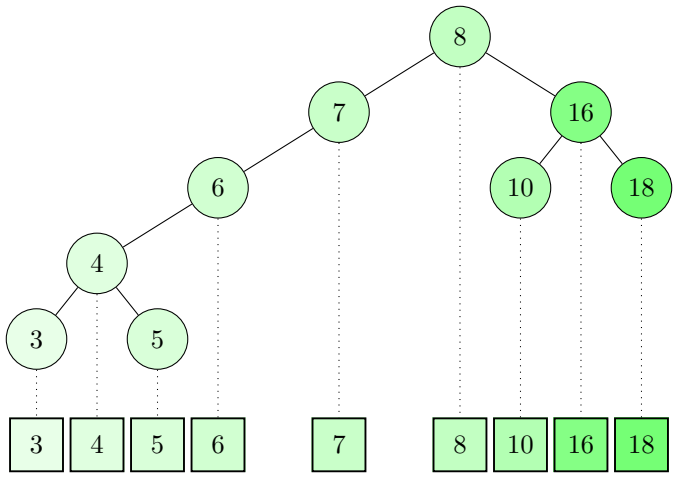
таким образом построение кучи работает за $O(n)$.

1.3 Двоичное дерево поиска (*BST*, *Binary Search Tree*)

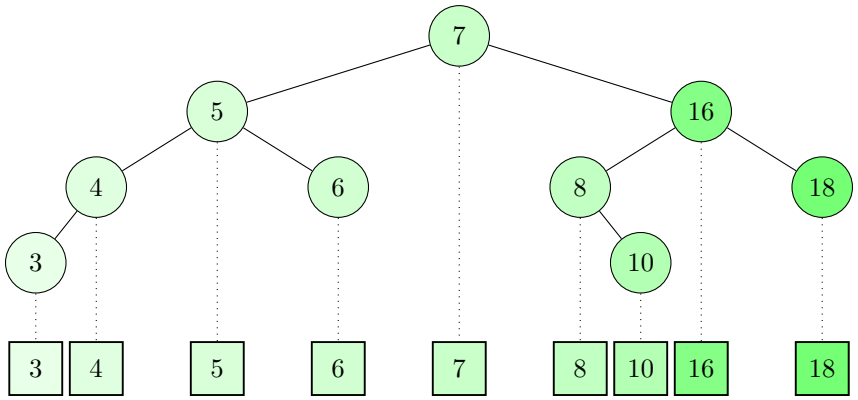
Вторая структура которую мы рассмотрим, *BST*, обладает следующими свойствами

- Это бинарное дерево
- Значения в левом поддереве вершины не превышают значений в самой вершине, а значения в правом поддереве больше либо равны значению в вершине.

Что дают эти свойства мы обсудим ниже, после того как посмотрим на пример (на том же массиве что и в прошлом разделе):



- Во-первых, данное дерево может быть несбалансированным. В принципе ему ничего не мешает быть графом-путем.
- Во-вторых, по этой структуре можно получить упорядоченный массив – достаточно выписывать элементы графа в следующем порядке – в начале элементы левого поддерева, потом корень, и, наконец, элементы правого поддерева. Такой порядок обхода называется In-order, есть и другие.
- Наконец, в-третьих, эта структура позволяет отвечать на запросы о наличии элемента в ней. А именно, для элемента a мы сравниваем его с вершиной, и идем в поддерево в котором a может лежать (или останавливаемся если значение в вершине равно a). При этом, если дерево сбалансировано (т.е. для любой вершины выполнено что число элементов в левом и правом поддеревьях отличается не более чем на 1) то этот поиск будет выполнятся за $O(\log n)$, где n – число вершин. Пример (один из возможных) сбалансированного дерева для того же массива представлен ниже:

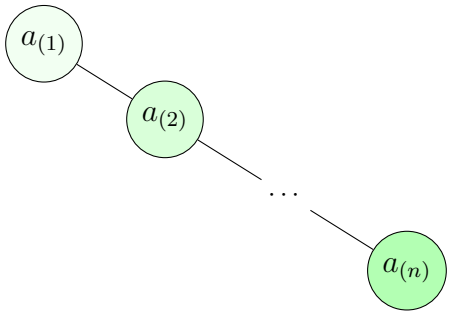


2 По поводу задач

Задача 2.1 (Задача 5 из листка 6)**.** Один программист решил хранить двоичное дерево поиска в массиве,используя ту же схему,что и для хранения кучи (во втором издании Кормена).Сколько ячеек массива понадобится ему в худшем случае для хранение дерева с n вершинами?

Решение. Выше мы уже видели, что BST может быть довольно плохим, оно не всегда сбалансировано. При этом длина массива (если хранить его так же как и кучу) определяется как раз высотой дерева – в случае кучи дерево в некотором смысле сбалансировано, поэтому там получается что при числе вершин n высота дерева есть $\lfloor \log_2 n \rfloor$, и, значит, массив будет иметь длину $l \leqslant 2^{\lfloor \log_2 n \rfloor + 1} - 1 < 2n$ (и это если хранить пустые поля для элементов которые могут появится на последнем слое).

Итак, чтобы занять как можно больший массив нам надо построить дерево наибольшей возможной высоты. При этом нам нужно лишь чтобы оно было бинарным, и сохранялась упорядоченность. Тогда мы можем сделать так:



при этом высота дерева h будет равна $n - 1$, и тогда размер массива будет $2^{h+1} - 1 = 2^n - 1$. □