Построение и анализ алгоритмов – Лекция 10. Двоичные поисковые деревья

Олег Смирнов
oleg.smirnov@gmail.com

8 декабря 2011 г.

Содержание

Цель лекции

- Двоичные поисковые деревья.
- Обход дерева и варианты записи дерева.
- Связь с Quicksort.

1 Двоичные и N-арные деревья

В теории графов, дерево – это связный (ориентированный или неориентированный) граф, не содержащий циклов. Т.е. для любой вершины есть один и только один способ добраться до любой другой вершины.

В программировании наиболее часто используются бинарные деревья, в которых число исходящих рёбер не превосходит 2, и N-арные деревья с произвольным количеством исходящих ребер.

В памяти компьютера деревья обычно представляют в виде связной структуры, где каждый узел помимо ключа (key) хранит указатели на дочерние узлы и иногда на родительский. Для хранения N-арных деревьев используют структуру с левым дочерним и правыми сестринским узлами (left-child, right-sibling representation). В этом случае вместо указателя на дочерние узлы каждый узел x хранит два указателя:

- в $left\_child[x]$ – указатель на крайний левый дочерний узел узла x;
- в $right\_sibling[x]$ – указатель на узел, расположенный на одном уровне с x справа от него.

1.1 Свойства деревьев

1. Дерево не имеет кратных ребер и петель.
2. Любое дерево с n узлами содержит $n-1$ ребро. Более того, конечный связный граф является деревом тогда и только тогда, когда $B - P = 1$, здесь B – число узлов, P – число рёбер графа.
3. Граф является деревом тогда и только тогда, когда любые два различных его узла можно соединить единственным элементарным путём.

4. Любое дерево однозначно определяется расстояниями (длиной наименьшей цепи) между его концевыми (степени 1) узлами.

5. Любое дерево является двудольным графом. Любое дерево, содержащее счётное количество вершин, является планарным графом.

2 Двоичные (бинарные) поисковые деревья

Двоичным или бинарным поисковым деревом называется бинарное дерево, для каждого узла x которого выполняется следующее свойство:

- если узел y лежит в левом поддереве узла x, то $key[y] < key[x]$;
- если узел y лежит в правом поддереве узла x, то $key[y] \geqslant key[x]$.

"Хорошими" считаются сбалансированные бинарные деревья с высотой порядка $O(\lg n)$. У несбалансированных бинарных деревьев высота может достигать n. Время прохода по бинарному дереву пропорционально его высоте. Цель – построить бинарное дерево с высотой порядка $O(\lg n)$ в большинстве случаев. Один из способов – рандомизация – рассматривается в данной лекции.

2.1 Сортировка массива

Бинарные деревья поиска можно использовать для сортировки массива:

```
1  T ← ∅
2  for i ← 1 to n
3        do Tree_Insert(T, A[i])
4  Infix_Traverse(root[T])
```

Время работы алгоритма складывается из частей:

- $O(n)$ для обхода Infix_Traverse.
- $\Omega(n \lg n)$ для Tree_Insert в среднем и в лучшем случае (идеально сбалансированное бинарное дерево).
- $T = \sum_{x \in T} depth(x) = \Theta(n^2)$ для Tree_Insert в худшем случае (массив уже отсортирован).

Поведение алгоритма похоже на поведение сортировки Хоара – Quicksort.

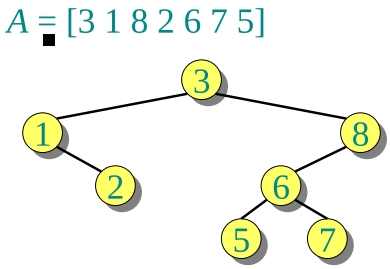

Рис. 1: Дерево работы BST sort

3 Бинарное поисковое дерево и сортировка Хоара

Алгоритмы BST sort и Quicksort выполняют одинаковое количество сравнений, но в различном порядке.

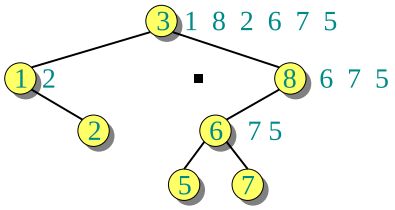

Рис. 2: Дерево работы Quicksort

Полученное дерево в точности совпадает с построенным в предыдущем примере.

Анализируя работу, можно увидеть, что алгоритм Quicksort в начале делает сравнение всех элементов с первым опорным (3), генерируя первое разбиение. BST sort также сравнивает каждый элемент в порядке добавления с корнем дерева (3). Аналогичные сравнения происходят для каждого элемента массива. Элемент, который в Quicksort становится опорным, в BST sort становится корнем поддерева.

3.1 Рандомизированный BST sort

1. Случайная перестановка элементов массива
2. Сортировка BST sort(A)

Время работы совпадает со временем рандомизированного Quicksort. Совпадает и мат. ожидание:

$$E[\text{time}] = E[\text{Randomized\_Quicksort}] = \Theta(n \lg n)$$

Нет смысла использовать BST sort для сортировки массива, т.к. время не отличается от Quicksort. Поиск по BST также не дает преимуществ по сравнению с бинарным поиском в просто отсортированном массиве. Полезность BST заключается в возможности добавлять элементы в структуру динамически, сохраняя ожидаемое время работы.

Ожидаемое время работы рандомизированного алгоритма BST sort $T(n)$ будет $\Theta(n \lg n)$. Время работы равно сумме глубины всех узлов дерева:

$$T(n) = \sum_x depth\ x$$

4 Анализ высоты BST

Интуитивно ясно, что ожидаемая высота дерева должна быть $\Theta(\lg n)$.

Ожидаемая *средняя* высота будет равна: $E[\frac{1}{n} \sum_{x \in T} depth\ x] = \frac{\Theta(n \lg n)}{n} = \Theta(\lg n)$.

Ожидаемая средняя высота дерева $\Theta(\lg n)$ не означает, что высота всего дерева также будет $\Theta(\lg n)$. Например, если в дереве есть один из путей длинной $\sqrt{(}n) > \lg n$, а остальные пути $\lg n - \sqrt{(}n)$, средняя высота, тем не менее, будет равна:

$$\leqslant \frac{1}{n}(n \lg n + \sqrt{(}n)\sqrt{(}n)) = O(\lg n)$$

Теорема: $E[$высота рандомизированного BST$] = O(\lg n)$

Доказательство теоремы позволит показать, что в рандомизированном BST можно производить поиск за (ожидаемое) логарифмическое время.

Схема доказательства:

1. Неравенство Йенсена для выпуклой функции f: $f(E[X]) \leqslant E[f(X)]$.
2. Вместо анализа X_n (случайная величина высоты BST) анализ $Y_n = 2^{X_n}$.

3. Доказательство $E[Y_n] = O(n^3)$.

4. Поиск границы $E[2^{X_n}] = E[Y_n] = O(n^3)$.

5. В соответствии с неравенством Йенсена $2^{E[X_n]} \leqslant E[2^{X_n}]$.

6. После логарифмирования получим $E[X_n] \leqslant \lg O(n^3) = 3 \lg n + O(1)$.

4.1 Ожидаемая высота

Пусть X_n – случайная величина высоты рандомизированного BST для n узлов, а $Y_n = 2^{X_n}$ – выпуклая функция.

Анализ высоты дерева похож на анализ алгоритма Quicksort в том смысле, что после выбора корня дерева r остальные элементы исходного массива разделяются на две части – меньшие r (пусть k элементов), которые попадут в левое поддерево и большие r $(n-k-1)$, которые попадут в правое поддерево. Каждое из поддеревьев также является случайным рандомизированным BST, что приводит к рекурсивному анализу алгоритма.

Если корень дерева r имеет ранг k, то $X_n = 1 + \max(X_{k-1}, Xn - k)$, а $Y_n = 2max(Y_{k-1}, Y_{n-k})$.

Можно показать, что

$$E[X_n] \leqslant 3 \lg n + O(1)$$

5 Алгоритмы работы с бинарными деревьями

Пусть высота бинарного дерева равна h. Тогда алгоритм поиска элемента k в дереве с корнем x выполняется за время $O(h)$:

```
TREE_SEARCH(x, k)
1  if x = NIL | k = key[x]
2      then return x
3  if k < key[x]
4      then return Tree_Search(left[x], k)
5      else  return Tree_Search(right[x], k)
```

Процедуру можно превратить в итеративную с помощью хвостовой рекурсии.

5.1 Обход дерева

Поиск элемента в дереве является частным случаем процедуры обхода дерева. Существует несколько принципиально разных способов обхода:

1. Обход в прямом порядке, когда каждый узел посещается до того, как посещены его потомки (*prefix traverse*). Для корня дерева рекурсивно вызывается следующая процедура:

   (a) Посетить узел.

   (b) Обойти левое поддерево.

   (c) Обойти правое поддерево.

   Такой обход используется, например, в решение задачи методом деления на части и в стратегии "разделяй и властвуй" (сортировка слиянием, быстрая сортировка, одновременное нахождение максимума и минимума последовательности чисел, умножение длинных чисел и т.д.).

2. Симметричный обход, когда сначала посещается левое поддерево, затем узел, затем правое поддерево (*infix traverse*). Для корня дерева рекурсивно вызывается следующая процедура:

   (a) Обойти левое поддерево.

   (b) Посетить узел.

   (c) Обойти правое поддерево.

3. Обход в обратном порядке (*postfix traverse*), когда узлы посещаются "снизу вверх" по следующей процедуре:

   (a) Обойти левое поддерево.

   (b) Обойти правое поддерево.

   (c) Посетить узел.

   Такой обход используется в анализе игр с полной информацией, в динамическом программировании и для вычисления выражений в постфиксной форме.

Все три варианта обхода в глубину можно реализовать итеративно.

Для бинарных и для произвольных (N-арных деревьев) существует обход в ширину, когда узлы посещаются уровень за уровнем (k-й уровень дерева – множество узлов с высотой k). Каждый уровень обходится слева направо. Для его реализации используется структура *queue* (очередь).

5.2 Вставка и удаление элемента

Алгоритм вставки узла z, у которого $key[z] = v$, $left[z] = NIL$, $right[z] = NIL$ в бинарное дерево T:

```
TREE_INSERT(T, z)
 1  y ← NIL
 2  x ← root[T]
 3  while x ≠ NIL
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else  x ← right[x]
 8  p[z] ← y
 9  if y = NIL
10     then root[T] ← z //Дерево T – пустое
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
```

Цикл в начале процедуры перемещает указатели вниз по дереву в зависимости от сравнения ключей $key[x]$ и $key[z]$, до тех пор, пока x не станет равным NIL. Это значение находится именно в той позиции, куда следует вставить узел z.

Процедура Tree_Delete рассматривает три возможных случая:

1. Если у узла z нет дочерних узлов, он просто удаляется из дерева.
2. Если у узла один дочерний узел, он “склеивается” с родительским для z.
3. Если дочерних узла два, то в дереве находим следующий за z узел y, у которого нет левого дочернего узла, убираем его из позиции, где он находился ранее и заменяем им узел z.

Можно показать, что если у узла BST два дочерних узла, то у предшествующего узла нет правого дочернего, а у последующего – левого.

Если в дереве T существуют два узла a и b, $a < b$, такие, что $rank(a) = k$, $rank(b) = k + 1$. Т.е. узел b является последующим за a. Тогда левым дочерним узлом узла b может являться только элемент $x < b$. Но из условия предшествия следует, что $x < a$ и $rank(x) < rank(a)$. Т.к. узлы a и b уже размещены в дереве, алгоритм Tree_Insert поместит x в левое поддерево элемента a, но не b.

Процедура Tree_Successor возвращает следующий элемент за аргументом в отсортированной последовательности.

```
TREE_DELETE(T, z)
 1  if left[z] = NIL | right[z] = NIL
 2     then y ← z
 3     else  y ← Tree_Successor(z)
 4  if left[y] ≠ NIL
 5     then x ← left[y]
 6     else  x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else  if y = left[p[y]]
12              then left[p[y]] ← x
13              else  right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16          //Копирование сопутствующих данных в z
17  return y
```

Очевидно, что балансировку дерева можно нарушить, вставляя или удаляя специально подобранные элементы. Для борьбы с таким поведением существуют специальные структуры данных и соответствующие алгоритмы: красно-чёрные деревья, AVL-деревья, декартовы деревья (Treap) и т.п.