МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**

Кафедра теории функций и стохастического анализа

ОТЧЕТ ПО ПРОИЗВОДСТВЕННОЙ (БАЗОВОЙ) ПРАКТИКЕ

студента 4 курса 451 группы
направления 38.03.05 — Бизнес-информатика

механико-математического факультета
Чайковского Петра Ильича

Место прохождения: завод "Тантал"
Сроки прохождения: с 29.06.2019 г. по 26.07.2019 г.
Оценка:

Руководитель практики от СГУ

доцент, к. ф.-м. н. _____ Н. Ю. Агафонова

Руководитель практики от организации

ведущий программист _____ Д. Э. Кнутов

Саратов 2019

Тема практики:«Правила оформления курсовых и дипломных работ»

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

При рассмотрении таких структур данных, как последовательности и списки, особое внимание привлекает простота их определения: последовательность (список) с базовым типом T - это либо:

1) пустая последовательность (список); либо

2) конкатенация элемента типа T и некоторой последовательности с базовым типом T. Для определения принципов построения, а именно следования или итерации, здесь используется рекурсия. Следования и итерация встречаются настолько часто, что их считают фундаментальными образами строения данных и поведения программ, однако всегда следует помнить, что их можно определять только с помощью рекурсии (обратное неверно), в то время как рекурсии можно эффективно употреблять для определения более сложных структур. Хорошо известным примером служат деревья, которые могут быть определены по следующему принципу: дерево с базовым типом T - это либо:

1) пустое дерево; либо

2) некоторая вершина типа T с конечным числом связанных с ней отдельных деревьев с базовым типом T, называемых *поддеревьями.*

Из сходства рекурсивных определений последовательностей и деревьев ясно, что последовательность (список) есть дерево, в котором каждая вершина имеет не более одного поддерева.

1 Постановка задачи. Описание метода её решения.

Для более корректной постановки задачи сформулируем ещё несколько обязательных определений:

Упорядоченное дерево - это дерево, у которого рёбра (ветви), исходящие из каждой вершины, упорядочены.

Степень вершины - число непосредственных потомков внутренней вершины.

Заметим, что особо важную роль при рассмотрении деревьев играют *упорядоченные* деревья *второй степени.* Такие деревья называются *бинарными.* Определим упорядоченное двоичное дерево как конечное множество элементов (вершин), которое либо пусто, либо состоит из корня (вершины) с двумя отдельными двоичными деревьями, которые называются *левым* и *правым поддеревом* этого корня.

Обращаясь же к проблеме представления деревьев, будем описывать как переменные с фиксированной структурой сами вершины, таким образом, их тип будет зафиксирован, и степень дерева будет определять число ссылочных компонент, указывающих на вершины поддеревьев. Ссылки на пустые деревья будут обозначаться значением *NIL.* Таким образом, компоненты дерева имеют такой вид:

```
1 TYPE Ptr = POINTER TO Node;
2 TYPE Node = RECORD op: CHAR;
3               left, right: Ptr
4             END
```

Листинг 1: Компоненты дерева.

Будем считать, что надо строить дерево минимальной глубины, состоящее из n вершин. В таком случае минимальная высота при заданном числе вершин достигается, если на всех уровнях, кроме последнего, помещается максимальное число вершин. Такого можно добиться, размещая приходящие вершины поровну слева и справа от каждой вершины.

Правило равномерного распределения для известного числа вершин n можно сформулировать, используя рекурсию:

1. Взять одну вершину в качестве корня.

2. Построить тем же способом левое поддерево с nl = n DIV 2 вершинами.

3. Построить тем же способом правое поддерево с nr = n - nl - 1 вершиной.

Такому правилу соответствует следующая процедура построения *идеально сбалансированного дерева* (причём будем исходить от следующего определения: дерево называется *идеально сбалансированным*, если число вершин в его левых и правых поддеревьях отличается не более, чем на 1):

```
1  PROCEDURE tree(n: INTEGER): Ptr:
2    VAR newnode: Ptr;
3      x, nl, nr: INTEGER;
4   BEGIN
5    IF n = 0 THEN newnode := NIL
6    ELSE nl := n DIV 2; NR := n - nl - 1;
7      ReadInt(x); ALLOCATE(newnode, SIZE(Node));
8      WITH newnode DO
9        key := x; left := tree(nl); right := tree(nr)
10     END
11   END
12   RETURN newnode
13  END tree;
```

Листинг 2: Процедура построения идеально сбалансированного дерева.

1.1 Поиск и включение элемента для дерева.

Рассмотрим ситуацию, в ходе процесса которой меняется структура дерева, то есть дерево будет расти или сокращаться в ходе выполнения программы.

Рассмотрим случай, когда дерево постоянно растёт, но не убывает. Типичным примером такой задачи будет являться построение частотного словаря.

Сформулируем задачу следующим образом: в заранее заданной последовательности надо определить частоту вхождения каждого из слов. Это означает, что любое слово надо искать в дереве, причём вначале дерево пустое. Если слово найдено, то счётчик его вхождений увеличивается, если нет - это слово включается в дерево с единичным значением счётчика. Будем называть такую задачу *поиском по дереву с включением.*

Приведём код процедуры поиска с включением:

```
1  PROCEDURE search(x: INTEGER; VAR p: WPtr):
2  BEGIN
3    IF p = NIL THEN
4      ALLOCATE(p.SIZE(Word));
5      WITH p DO
6        key := x; count := 1; left := NIL; right := NIL
7      END
8    ELSIF x < p.key THEN search(x, p.left)
9    ELSIF x > p.key THEN search(x, p.right)
10   ELSE p.count := p.count + 1
11   END
12 END SEARCH
```

Листинг 3: Процедура поиска элемента с его включением в дереве.

1.2 Исключение из деревьев.

Теперь рассмотрим задачу *исключения*. Стоит заметить, что процедура исключения элемента из дерева оказывается простой только в том случае, если исключаемый элемент - терминальная вершина или вершина с одним потомком. Трудность возникает в том случае, когда нужно удалить элемент с двумя потомками. В этом случае удаляемый элемент нужно заменить либо на самый правый элемент его левого поддерева, либо на самый левый элемент его правого поддерева, причём они должны иметь как максимум одного потомка.

Поэтому процедура удаления (исключения) элемента из дерева должна различать три случая:

1. Компоненты с ключом, равным x, нет.

2. Компонента с ключом x имеет не более одного потомка.

3. Компонента с ключом x имеет двух потомков.

Приведём код рассматриваемой процедуры:

```
1  PROCEDURE delete(x: INTEGER; VAR p: Ptr):
2    VAR q: Ptr;
3
4    PROCEDURE del(VAR r: Ptr):
5    BEGIN
6      IF r.right # NIL THEN del(r.right)
7      ELSE q.key := r.key; q.count := r.count;
8        q := r; r := r.left
9      END
10   END del;
11
12   BEGIN
13     IF p = NIL THEN ( * слова в дереве нет * )
14     ELSIF x < p.key THEN delete(x, p.left)
15     ELSIF x > p.key THEN delete(x, p.right)
16     ELSE ( * исключение p * ) q := p;
17       IF q.right = NIL THEN p := q.left
18       ELSIF q.left = NIL THEN p := q.right
19       ELSE del(q.left)
20       END;
21   END
```

```
22 END delete
```

Листинг 4: Процедура исключения элемента из дерева.

Вспомогательная процедура *del* начинает работать только в случае 3. Она спускается вдоль правой ветви левого поддерева элемента q, который нужно исключить, и заменяет существенную информацию (ключ и счётчик) в q на соответствующие значения из самой правой компоненты r левого поддерева, после чего от q можно освободиться.

2 Вычисление сложности проблемы.

2.1 Построение идеально сбалансированного дерева.

К сожалению, не существует быстрых алгоритмов, для выполнения базовых операций для идеально сбаланированных деревьев. Однако, определение идеально сбаланированного дерева, фактически, дает нам алгоритм его построения. Для построения идеально сбаланированного дерева по набору из n элементов упорядочим этот набор. После этого алгоритм построения дерева сводится к разбиению полученной последовательности a_i, $i = 1, \ldots, n$ на последовательности a_i, $i = 1, \ldots, [n/2]$ и a_i, $i = [n/2] + 2, \ldots, n$. Эти последовательности либо имеют равную длину (для нечетных n), либо их длина отличается не более, чем на единицу (для четных n). В корень создаваемого дерева помещаем элемент $a[n/2] + 1$, а левое и правое поддеревья строим таким же алгоритмом, соответственно, для последовательностей $a_i\,, i = 1, \ldots, [n/2]$ и $a_i\,, i = [n/2] + 2, \ldots, n$.

Таким образом, приведенный алгоритм доказывает утверждение о том, что идеально сбалансированное дерево поиска, состоящее из n вершин, можно построить за время, равное $O(n \cdot \log_2 n)$.

2.2 Поиск элемента в дереве поиска.

Рассмотрим сложность некоторых основных операций, которые можно выполнять над деревом. В качестве таких основных операций рассмотрим *поиск* в дереве поиска, поиск по дереву с включением и исключение из дерева.

Сначала рассмотрим поиск в дереве поиска. *Деревом поиска* будем называть дерево, организованное по следующему правилу: если для каждой вершины t_i справедливо, что все ключи левого поддерева t_i меньше ключа t_i, а все ключи правого поддерева t_i больше его, то такое дерево будем называть *деревом поиска*. В таком дереве можно найти произвольный ключ - достаточно, начав с корня, двигаться по правому или левому поддереву на основании сравнения с ключом текущей вершины.

Поскольку из n элементов можно организовать идеально сбалансированное двоичное дерево с высотой не более $\log n$, то поиск в таком дереве среди его n элементов выполняется максимум за $\log n$ сравнений, поэтому временная сложность поиска элемента в дереве поиска будет $O(\log n)$, где n - количество узлов в дереве.

Приведём код алгоритма поиска в дереве поиска:

```
1 PROCEDURE locate(x: INTEGER; t: Ptr): Ptr:
2 BEGIN
3   WHILE (t # NIL) & (t.key # x) DO
4     IF t.key < x THEN t := t.right ELSE t := t.left END
5   END;
6   RETURN t
7 END locate
```

Листинг 5: Процедура поиска элемента в дереве поиска

2.3 Поиск и включение элемента для дерева.

Предположение о том, что при вставке новых элементов в дерево, оно будет сохранять структуру идеально сбалансированного дерева, является идеализированным. Скорее всего, такая структура не будет сохранена, и число сравнений в дереве, сформированном нашим алгоритмом, будет больше. Остаётся узнать насколько.

Начнём с рассмотрения худшего случая. Будем предполагать, что все поступающие ключи идут в строго возрастающем (убывающем) порядке. Тогда каждый ключ присоединяется справа (слева) от его предшественника, и в результате мы получим полностью вырожденное дерево, вытянутое в линейный список. В этом случае средние затраты на поиск равны $n/2$ сравнениям. Такой вариант приводит к неудовлетворительной производительности, но необходимо узнать, насколько он вероятен, а именно: необходимо узнать усреднённую длину пути для всех n ключей и для всех $n!$ деревьев, полученных в результате $n!$ перестановок из этих n исходных ключей. Проведём анализ сложности алгоритма:

Пусть даны n различных ключей со значениями $1, 2, \ldots, n$, причём поступают они в случайном порядке. Вероятность того, что первый ключ, который становится корневым узлом, имеет значение i, равна $1/n$. Его левое поддерево будет состоять из $i-1$ вершин, а правое - из $n-i$ вершин. Обозначим среднюю длину пути в левом поддереве через a_{i-1}, а в правом поддереве - через a_{n-i}. Будем вновь предполагать, что все возможные перестановки оставшихся $n-1$ ключей равновероятны. Средняя длина пути в дереве с n вершинами равна сумме произведений уровня каждой вершины на вероятность обращения к ней.

Также мы можем разделить вершины на три класса:

1. $i-1$ вершин в левом поддереве имеют среднюю длину пути a_{i-1};
2. корень имеет длину пути 0;
3. $n-i$ вершин в правом поддереве имеют среднюю длину пути a_{n-i}.

Тогда средняя длина пути с n вершинами может быть найдена как:

$$a_n^{(i)} = \frac{((i-1)\cdot a_{i-1} + (n-i)\cdot a_{n-i})}{n}$$

Искомое значение a_n - среднее значение $a_n^{(i)}$ для всех $i = 1, \ldots, n$, то есть для всех деревьев с ключами в корне, равными $1, 2, \ldots, n$:

$$a_n = \frac{\sum_{i=1}^{n}((i-1)\cdot a_{i-1} + (n-i)\cdot a_{n-i})}{n^2} = 2\cdot\frac{\sum_{i=1}^{n}(i-1)\cdot a_{i-1}}{n^2} =$$

$$2\cdot\frac{\sum_{i=1}^{n-1} i\cdot a_i}{n^2}.$$

Это уравнение представляет собой рекуррентное соотношение вида $a_n = f_1(a_1, a_2, \ldots, a_n)$. Из него мы получаем более простое рекуррентное соотношение вида $a_n = f_2(a_{n-1})$. Сначала получаем (1), раскладывая последнее произведение, и (2), подставляя $n-1$ вместо n:

(1) $a_n = 2\cdot(n-1)\cdot a_{n-1}/n^2 + 2\cdot(\sum_{i=1}^{n-1} i\cdot a_i)/n^2$
(2) $a_{n-1} = 2\cdot(\sum_{i=1}^{n-2} i\cdot a_i)/(n-1)^2$

Умножая (2) на $\frac{(n-1)^3}{n^2}$, получаем:

(3) $2\cdot\frac{\sum_{i=1}^{n-2} i\cdot a_i}{n^2} = \frac{a_{n-1}\cdot(n-1)^2}{n^2}$

и, подставив правую часть (3) в (1), найдём:

Заметим, что a_n можно представить в нерекурсивном замкнутом виде с помощью гармонической функции:

$H_n = 1 + 1/2 + 1/3 + \cdots + 1/n \implies a_n = 2\cdot(H_n\cdot(n+1)/n - 1)$

Из формулы Эйлера (используя константу $\gamma = 0.577\ldots$):

$H_n = \gamma + \ln n + 1/12n^2 + \ldots$

получаем для больших n соотношение $a_n = 2\cdot(\ln n + \gamma - 1)$. Так как средняя длина пути в идеально сбалансированном дереве приблизительно равна $a'_n = \log n - 1$, то, опуская слагаемые, которые при больших n становятся малыми, получаем:

$$\lim_{n\to\infty}\frac{a_n}{a'_n} = 2\cdot\frac{\ln n}{\log n} = 2\cdot\ln(2) = 1.386\ldots$$

Этот результат означает, что стараясь всегда строить идеально сбалансированное дерево вместо случайного, мы можем ожидать средний выигрыш в длине пути поиска не более 30% при предположении, что все ключи распределены равномерно.

Цифра 39% накладывает ограничение на объём дополнительных ухищрений для перестройки структуры дерева по мере поступления ключей. При этом имеет существенное значение отношение частоты обращений (поисков) к вершинам к частоте включений (коррекций дерева). Чем больше это отношение, тем больше выигрыш от процедуры перестройки. Однако цифра 39% достаточно низка, и в большинстве приложений улучшение простого алгоритма поиска с включением не оправдывается, если не предусматривается большого числа вершин и большего отношения доступ/включение.

2.4 Исключение элемента из дерева.

Поскольку при удалении вершины из дерева, оно может разбалансироваться, это нужно учитывать в алгоритме удаления элемента из дерева. Таким образом, процедура удаления вершины v из сбалансированного дерева поиска сводится к следующему:

1) Нахождение вершины v, которую надо удалить

2) Её удаления из дерева поиска (с помещением на её место некоторой вершины v')

3) Для каждой вершины ветви дерева от v' до корня необходимо проверить условие балансировки - если оно нарушилось, то нужно провести балансировку соответствуюшего поддерева.

Таким образом, в силу построения алгоритма удаления вершины из сбалансированного дерева, верно утверждение о том, что из сбалансированного дерева поиска, состоящего из n вершин, *можно* удалить одну вершину за время $O(\log_2 n)$.

3 Программа решения на языке C++.

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ