Лабораторная работа № 7

Методические указания

Деревья, хеш-таблицы

Цель работы – построить и обработать хеш-таблицы, сравнить эффективность поиска в сбалансированных деревьях, в двоичных деревьях поиска и в хеш-таблицах.

Краткие теоретические сведения

Рассмотрим алгоритм построения двоичного дерева. Если при добавлении узлов в дерево мы будем их равномерно располагать слева и справа, то получится дерево, у которого число вершин в левом и правом поддеревьях отличается не более, чем на единицу. Такое дерево называется **идеально сбалансированным**.

Для построения идеально сбалансированного дерева используется рекурсия. Для дерева из **n** узлов, где **nl** - количество узлов в левом поддереве, **nr** - количество узлов в правом поддереве, алгоритм построения идеально сбалансированного дерева описывается следующим образом:

1. выбрать одну вершину в качестве корня;
2. рекурсивно построить левое поддерево с **nl = n div 2** узлами;
3. рекурсивно построить правое поддерево с **nr = n – nl 1** узлами.

Идеальная балансировка дает наименьшую высоту дерева, а так как высота дерева определяет длину пути поиска в нем, то, следовательно, и укорачивает поиск. Но поддержание идеальной сбалансированности дерева при включении или исключении элемента – это достаточно сложная процедура.

Адельсон-Вельский и Ландис сформулировали менее жесткий критерий сбалансированности таким образом: двоичное дерево называется сбалансированным, если у каждого узла дерева высота двух поддеревьев отличается не более чем на единицу. Такое дерево называется *АВЛ-деревом.*

Использование этого критерия приводит к легко выполняемой балансировке. При этом средняя длина поиска остается практически такой же, как и у идеально сбалансированного дерева.

При включении узла в сбалансированное дерево возможны 3 случая: (рассматриваем включение в левое поддерево):

1) левое и правое поддеревья становятся неравной высоты, но критерий сбалансированности не нарушается;

2) левое и правое поддерево приобретают равную высоту и, таким образом, сбалансированность даже улучшается;
3) критерий сбалансированности нарушается, и дерево надо перестраивать.

Алгоритм включения и балансировки существенно зависит от способа хранения информации о сбалансированности дерева. Одно из решений – хранить в каждой вершине показатель ее сбалансированности. В этом случае сбалансированность будет определяться как разность между высотой правого и левого поддеревьев. Например, Bal =1, если правое поддерево выше левого, Bal=0, если правое и левое поддеревья равной высоты и Bal= –1, если левое поддерево выше правого.

Процесс включения узла фактически состоит из трех последовательно выполняемых подзадач:

1) проход по пути поиска (пока не убедимся, что элемента с таким ключом в дереве нет);
2) включение нового узла и определение показателя сбалансированности (Bal);
3) «отступление» (возврат) по пути поиска, с проверкой показателей сбалансированности для каждой вершины, и если необходимо, то проведение балансировки.

Этот метод требует некоторой избыточной проверки, но зато его можно реализовать с помощью простого расширения алгоритма поиска с включением. На каждом шаге при этом необходима информация о высоте дерева.

Операция по балансировке состоит только из последовательных переприсваиваний ссылок. Фактически ссылки циклически меняются, что приводит к одно- или двукратному повороту двух или трех участвующих в процессе балансировки узлов. Кроме вращения, необходимо должным образом изменять и показатели сбалансированности этих узлов.[1]

Алгоритм удаления узла из сбалансированного дерева основан на алгоритме удаления из дерева (на замене удаляемого узла на самого левого потомка из правого поддерева или на самого правого потомка из левого поддерева) с учетом операции балансировки, то есть тех же поворотов узлов.[1]

Использование деревьев для поиска информации достаточно эффективно (трудоемкость – $O(\log_2 n)$). Можно ли создать еще более эффективную структуру или метод, позволяющий лучше осуществлять поиск информации? Для этого было бы хорошо по значению ключа сразу определять индекс элемента массива, в котором хранится информация. То есть необходимо создать такую функцию, по которой можно вычислить этот индекс. Такая функция называется *хеш-функцией* (от англ. to hash -

крошить, рубить) и она ставит в соответствие каждому ключу k_i индекс ячейки j, где расположен элемент с этим ключом, таким образом:

$$h(k_i) = j, \text{ если } j=(1,m),$$

где j принадлежит множеству от 1 до m, а m. – размерность массива.

Массив, заполненный в порядке, определенным хеш-функцией, называется *хеш-таблицей*. Минимальная трудоемкость поиска в хеш-таблице равна O(1)!

Принято считать, что хорошей является такая функция, которая удовлетворяет следующим условиям:

- функция должна быть простой с вычислительной точки зрения;
- функция должна распределять ключи в хеш-таблице наиболее равномерно.

Итак, первое свойство хорошей хеш-функции зависит от характеристик компьютера, а второе – от значений данных. Если бы все данные были случайными, то хеш-функции были бы очень простые (несколько битов ключа, например). Однако на практике случайные данные встречаются крайне редко, и приходится создавать функцию, которая зависела бы от всего ключа.

Может возникнуть ситуация, когда разным ключам соответствует одно значение хеш-функции, то есть, когда h(K1)=h(K2), в то время как K1 ≠ K2. Такая ситуация называется *коллизией*. В этом случае, очевидно, необходимо найти новое место для хранения ключей, претендующих на одну и ту же ячейку хеш-таблицы. Причем количество коллизий необходимо минимизировать. Таким образом, хорошая хеш-функция должна удовлетворять еще одному требованию, а именно: она должна минимизировать число коллизий.

Существует два основных типа хеширования, один из которых основан на делении, а другой на умножении. Впрочем, это не единственные методы, которые существуют, более того, они не всегда являются оптимальными.

Если ключей меньше, чем элементов массива, то в качестве хеш-функции можно принять вычисление остатка от деления целочисленного ключа на размерность массива (m), то есть:

$$h(k_i) = (k_i \bmod m),$$

при n<m, где n – количество ключей.

Данная функция очень проста, хотя и не самая лучшая. Вообще, можно использовать любую размерность (константу m), но она должна быть такой, чтобы минимизировать число коллизий. Для этого лучше использовать простое число. В большинстве случаев подобный выбор вполне удовлетворителен. Для символьной

строки ключом может являться остаток от деления, например, суммы кодов символов строки на m.

На практике, метод деления – самый распространенный

Существует несколько возможных вариантов разрешения коллизий, которые имеют свои достоинства и недостатки.

Первый метод – внешнее (открытое) хеширование (метод цепочек)

В случае, когда элемент таблицы с индексом, который вернула хеш-функция, уже занят, к нему присоединяется связный список. Таким образом, если для нескольких различных значений ключа возвращается одинаковое значение хеш-функции, то по этому адресу находится указатель на связанный список, который содержит все значения. Поиск в этом списке осуществляется простым перебором, так как при грамотном выборе хеш-функции любой из списков оказывается достаточно коротким.

Другой путь решения проблемы, связанной с коллизиями – внутреннее (закрытое) хеширование (открытая адресация). Оно, состоит в том, чтобы полностью отказаться от ссылок. В этом случае, если ячейка с вычисленным индексом занята, то можно просто просматривать следующие записи таблицы по порядку (с шагом 1), до тех пор, пока не будет найден ключ K или пустая позиция в таблице. При этом, если индекс следующего просматриваемого элемента определяется добавлением какого-то постоянного шага (от 1 до n), то данный способ разрешения коллизий называется линейной адресацией. Для вычисления шага можно также применить формулу:

$$h = h + a^2,$$

где a – это номер попытки поиска ключа. Этот вид адресации называется квадратичной или произвольной адресацией.

При любом методе разрешения коллизий необходимо ограничить длину поиска элемента. Если для поиска элемента необходимо более 3–4 сравнений, то эффективность использования такой хеш-таблицы пропадает и ее следует реструктуризировать (т.е. найти другую хеш-функцию), чтобы минимизировать количество сравнений для поиска элемента

Задание

Построить хеш-таблицу по указанным данным. Сравнить эффективность поиска в сбалансированном двоичном дереве, в двоичном дереве поиска и в хеш-таблице. Вывести на экран деревья и хеш-таблицу. Подсчитать среднее количество сравнений для поиска данных в указанных структурах. Произвести реструктуризацию хеш-таблицы, если среднее количество сравнений больше указанного. Оценить

эффективность использования этих структур (по времени и памяти) для поставленной задачи.

Указания к выполнению работы

При разработке интерфейса программы следует предусмотреть:

- указание типа, формата и диапазона вводимых данных;
- указание действий, производимых программой;
- наличие пояснений при выводе результата;
- вывод деревьев осуществить в графическом виде (или предложить иную визуализацию в виде дерева);
- вывод на экран хеш-таблицы;
- возможность изменения хеш-функции при необходимости реструктуризации таблицы;
- вывод времени и количества сравнений при поиске одних и тех же данных в различных структурах данных.

При тестировании программы необходимо:

- проверить правильность ввода и вывода данных (то есть их соответствие требуемому типу и формату), обеспечить адекватную реакцию программы на неверный ввод данных;
- обеспечить вывод сообщений при отсутствии входных данных («пустой ввод»);
- проверить правильность выполнения операций;
- предусмотреть вывод сообщения при необходимости реструктуризации хеш-таблицы;
- проверить поиск существующих и поиск несуществующих данных;

Содержание отчета

В отчете по лабораторной работе должны быть сделаны выводы о том, применение какой структуры (дерева двоичного поиска, сбалансированного дерева или хеш-таблицы) целесообразно для решения поставленной задачи, какие преимущества дает использование той или иной структуры. Выводы следует подтвердить результатом числовых сравнений расходования памяти и времени выполнения программы, а также количеством сравнений при поиске данных. В отчете

также следует указать, в каком случае необходима реструктуризация хеш-таблицы и на что нужно обратить особое внимание при тестировании программы

В отчете по лабораторной работе должны быть даны ответы на следующие вопросы:

1. Чем отличается идеально сбалансированное дерево от АВЛ дерева?
2. Чем отличается поиск в АВЛ-дереве от поиска в дереве двоичного поиска?
3. Что такое хеш-таблица, каков принцип ее построения?
4. Что такое коллизии? Каковы методы их устранения.
5. В каком случае поиск в хеш-таблицах становится неэффективен?
6. Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска и в хеш-таблицах

Отчет представляется в электронном или печатном виде.

Список рекомендуемой литературы

1. *Вирт Н.* Алгоритмы и структуры данных: Пер. с англ. СПб.: Невский диалект, 2001. С.261 – 274, 324–336.
2. *Ахо А.*, *Хопкрофт Д.*, *Ульман Д.* Структуры данных и алгоритмы: Пер. с англ. М.: Издат. дом «Вильямс», 2000. С. 77–99.
3. *Кнут Д.* Искусство программирования, Т. 3. Сортировка и поиск: Пер. с англ. М.: Издат. дом «Вильямс», 2001. С. 492–507, 549–586.
4. *Кормен Т.*, *Лейзерсон Ч.*, *Ривест Р.*, Алгоритмы: построение и анализ: Пер. с англ. М.: МЦНМО, 2001. С. 213–235.