Лабораторная работа №6 по дисциплине

“Типы и структуры данных”

Сорокин Антон ИУ7-32Б

Номер по списку – 24

Вариант 4

***Условие задачи***

В текстовом файле содержатся целые числа. Построить ДДП из чисел файла. Вывести его на экран в виде дерева. Сбалансировать полученное дерево и вывести его на экран. Построить хеш-таблицу из чисел файла. Использовать закрытое хеширование для устранения коллизий. Осуществить удаление введенного целого числа в ДДП, в сбалансированном дерево, в хеш-таблицу и в файл. Сравнить время поиска и удаления, объем памяти и количество сравнений при использовании различных (4-х) структур данных. Если количество сравнений в хеш-таблице больше указанного, то произвести реструктуризацию таблицы, выбрав другую функцию.

***Техническое задание***

*Исходные данные*

На вход программе подаётся текстовый файл с целыми числами.

*Результат*

Вывод полученных структур данных и данные о времени работы поиска и удаления при различных структурах данных. Дерево двоичного поиска также представляется графически в формате .png.

*Описание задачи*

Программа считывает из файла последовательность целых чисел, из которой составляет дерево двоичного поиска и хэш-таблицу. Программа осуществляет операции поиска и удаления в данных структурах данных, для дерева двоичного поиска выполняет его балансировку, сравнивает времена работы соответствующих операций при использовании данных структур

*Аварийное завершение работы программы*

Программа завершается аварийно, если

* библиотеки или отдельные функции библиотек не загрузились;
* не подан файл в качестве аргумента на вход программе;
* файл невозможно открыть;
* файл пустой;
* в файле некорректные данные (символы, вещественные числа и т. д.);
* случилась ошибка выделения памяти при чтении из файлы в различные структуры.

***Обращение к программе***

Исполняемый файл app.exe создается путем автоматической сборки проекта с помощью файла makefile. Для выполнения работы следует запустить данный исполняемый файл с указанием имени файла с исходными данными в качестве аргумента. Для запуска модуля анализа работы функций по времени и объёму памяти собирается исполняемый файл test.exe. Для выполнения его работы дополнительных аргументов указывать не требуется.

***Алгоритм***

Поиск в ДДП:

* Если корня не существует, то элемента нет в дереве, поиск завершить.
* Если искомый ключ равен ключу вершины, то поиск завершить, вернуть указатель на вершину.
* Если искомый ключ меньше ключа вершины, то рекурсивно выполнить поиск ключа в левом поддереве.
* Если искомый ключ больше – в правом поддереве.

Удаление из ДДП / АВЛ-дерева:

* Если корня не существует, то элемента нет в дереве, поиск завершить.
* Если искомый ключ меньше ключа вершины, то рекурсивно выполнить удаление вершины из левого поддерева.
* Если искомый ключ больше – из правого поддерева.
* Если искомый ключ равен ключу вершины, то:
  + Если вершина является терминальной, то вернуть пустой указатель.
  + Если нет одного из поддеревьев, то вернуть указатель на противоположный.
  + Иначе найти максимальный элемент левого поддерева:
    - Двигаться по правым ветвям левого поддерева до тех пор, пока правая ветвь не завершится.
    - Переставить максимальный элемент левого поддерева на место удаляемой вершины.
    - Вернуть указатель на эту вершину.
  + В случае АВЛ-дерева перед возвратом указателя на нового вершину предварительно сбалансировать данный узел.

Балансировка узла АВЛ-дерева:

* Если корня не существует, то вернуть пустой указатель.
* Определить фактор балансировки: найти разность высот правого и левого поддерева.
* Если фактор равен -2, то:
  + Найти фактор балансировки для левого поддерева.
  + Если фактор баланс. левого поддерева больше 0, то выполнить поворот влево левого поддерева.
  + Повернуть дерево вправо.
  + Вернуть указатель на новый корень.
* Если фактор равен 2, то:
  + Найти фактор балансировки для правого поддерева.
  + Если фактор баланс. правого поддерева меньше 0, то выполнить поворот вправо правого поддерева.
  + Повернуть дерево влево.
  + Вернуть указатель на новый корень.
* Иначе вернуть указатель на корень.

Поиск в хэш-таблице:

* Определить значение хэш-функции по ключу.
* В цикле до 4 сравнений:
  + Если найдена пустая или очищенная ячейка (определяется по значению флага), ключа нет – закончить поиск.
  + Если ключ найден, то вернуть индекс.
  + Иначе двигаться на следующую ячейку в таблице.
    - При достижении границы выделенной области памяти перейти в начало таблицы.
* После 4х сравнений считается, что ключа в таблице нет.

Удаление из хэш-таблицы:

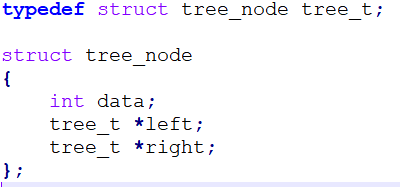
* Выполнить поиск ключа в таблице.
* Если ключ найден, то по индексу обозначить ячейку очищенной (изменить значение флагу).

***Тестовые данные***

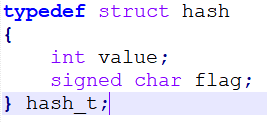
|  |  |  |
| --- | --- | --- |
| Вход | Выход | Класс |
|  |  |  |

***Внутренние структуры данных***

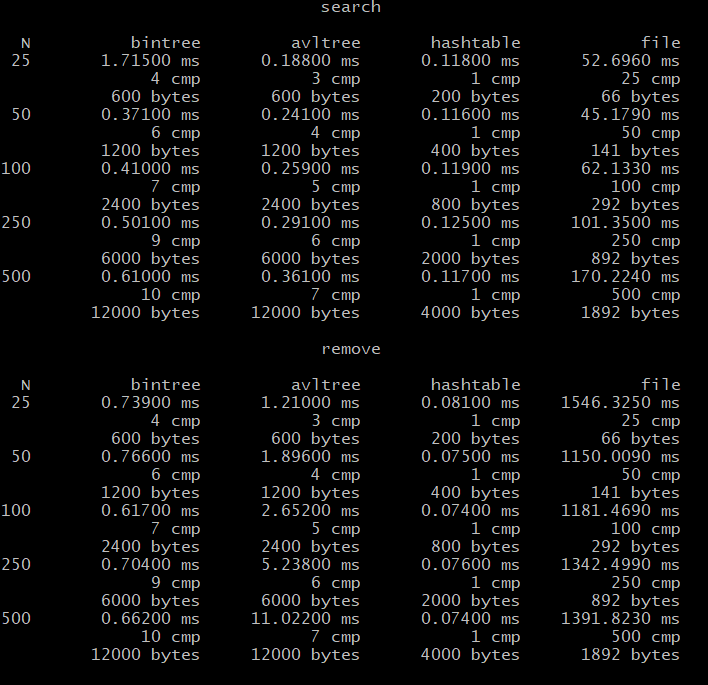
Для хранения деревьев двоичного поиска используется такая структура данных, в которой каждый элемент дерева представляет собой запись с тремя полями: значение узла (целое число), указатели на левое и на правое поддеревья.



Для хранения хэш-таблицы используется статический массив с ограничением в 1000 элементов. Элементами массива является структура из двух полей: ключа и флага, отвечающего за заполненность ячейки. Текущий размер таблицы регулируется отдельной целочисленной переменной.



***Оценка эффективности***



Исходя из полученных данных, можно сделать вывод о том, что по времени эффективнее всего использовать хэш-таблицу: при поиске или удалении производится всего 1 сравнение, которое занимает малый промежуток времени (не более десятой доли миллисекунды). Выигрыш во времени зависит от количества элементов.

Однако следует помнить, что при возможном рехешировании заново перестраивается вся таблица, что затрачивает большое кол-во времени.

Файл использовать эффективнее по памяти, чем любые другие структуры, в 5-10 раз, однако обработка файлов серьёзно проигрывает по времени – в 10000 раз, на это уходит около одной секунды при большом количестве данных, в то время как при обработке других структур уходят доли миллисекунд.

По времени поиска сбалансированное дерево выигрывает у несбалансированного в среднем на 40%: необходимо произвести меньшее количество сравнений. Однако АВЛ-дерево существенно проигрывает обыкновенному ДДП по времени удаления (в зависимости от количества элементов в 2 – 20 раз), поскольку после удаления требуется сбалансировать поддерево.

***Вывод***

Если не производится частого добавления элементов, то эффективнее всего использовать хэш-таблицу, поскольку она требует небольшой размер памяти (поле данных и поле флага) и тратит десятые доли миллисекунд на поиск.

В противном случае, хэш-таблица будет тратить много времени на рехеширование. В этом случае лучше воспользоваться деревьями двоичного поиска. Для сокращения времени поиска можно изначально строить сбалансированное ДДП.

***Ответы на вопросы***

**1. Что такое дерево?**

Дерево – это нелинейная структура данных, используемая для представления иерархических связей, имеющих отношение «один ко многим».

**2. Как выделяется память под представление деревьев?**

Под дерево выделяется память на каждый узел. Узел представляют собой структуру из трёх полей: данные, указатель на левое поддерево, указатель на правое.

**3. Какие стандартные операции возможны над деревьями?**

Над деревьями возможны операции: обхода, поиск, включения и исключения.

**4. Что такое дерево двоичного поиска?**

Дерево двоичного поиска – это такое дерево, в котором все левые потомки моложе предка, а все правые – старше.

**5. Чем отличается идеально сбалансированное дерево от АВЛ дерева?**

Критерий сбалансированности идеально сбалансированного заключается в том, чтобы число вершин в поддеревьях отличается не более, чем на единицу. Критерий же для АВЛ-деревьев определяет разницы высот поддеревьев, а не общее количество вершин.

**6. Чем отличается поиск в АВЛ-дереве от поиска в дереве двоичного поиска?**

Алгоритм поиска не отличается, но отличается эффективность, поскольку высота дерева может быть значительно меньше, чем у обычного ДДП.

**7. Что такое хеш-таблица, каков принцип ее построения?**

Хэш-таблица – массив, заполненный в порядке, определённым хэш-функцией, которая определяет по ключу индекс, по которому следует записать данный элемент.

Эта функция должна:

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

**8. Что такое коллизии? Каковы методы их устранения.**

Коллизией называется ситуация, когда разным ключам соответствует одно значение хэш-функции. Существует несколько методов их устранения. Самыми распространёнными являются методы открытого и закрытого хеширования. При открытом к занятой ячейке присоединяется связный список элементов с одинаковыми значениями хэш-функции. При закрытом просматриваются следующие за определённой хэш-функцией ячейки до тех пор, пока не будет найдена пустая или не будет определено, что хэш-таблица заполнена.

**9. В каком случае поиск в хеш-таблицах становится неэффективен?**

Поиск неэффективен при возникших коллизиях: максимальное количество сравнений для нахождения элемента увеличивается до количества возникших коллизий. Это говорит о неэффективной хэш-функции.

**10. Эффективность поиска в АВЛ деревьях, в дереве двоичного поиска и в хеш-таблицах**

Самый быстрый поиск выполняется в хэш-таблицах, если хэш-функция достаточно хорошая. Если в результате заполнения таблицы возникли коллизии, то эффективность поиска падает. Но в общем случае сложность составляет O(1).

В ДДП поиск выполняется со сложностью O(log2n). Если дерево сбалансированно (АВЛ-дерево), то его высота стремится к наименьшей, что укорачивает значительно поиск.